

- JMS and Multi-threading.
- JMS and Security
- Summary

Part II: JMS Messaging

Chapter 5: The Nuts and Bolts of JMS Messages???????????????????????. 48

- Introducing the JMS Messages
- The Message interface
- The Message Header and its components
 - o The Message Properties
 - o Standard properties
 - o Application specific properties
 - o Provider specific properties
- The Message Body
 - o Text Message
 - o Object Message
 - o Stream Message
 - o Map Message
 - o Bytes Message
- Message Selection
 - o The syntax rules with examples
- Summary

Chapter 6: JMS Messaging Models?????????????????????..?????... 69

- Point to Point Messaging
 - o The Components
 - o An Example: Creating a JMS Phone
- Publish-Subscribe Messaging
 - o The Components
 - o An Example: Creating a JMS based Chat
- Request/Reply Messaging
 - o What is it?
 - o Modifying the point-to-point example to use Request/Reply
 - o Simulating Synchronous calls with Request/Reply
 - o An Example: A Simple Compute Server based on Request/Reply
- Summary

Part III: JMS in the Real World

Chapter 7: Using XML with JMS?????????????????????..?????. 124

- Why use XML with JMS
- An XML refresher
 - o Why is XML so important?
 - o Manipulating XML programmatically
- Back to JMS
 - o The JmsXMLHelper class
 - o Using the JmsXMLHelper class
- Summary

Chapter 8: Space-based Programming with JMS?????????????????..?????. 137

- The need for Spaces: A common problem in distributed computing
- An introduction to "Space-based" programming
- Using a JMS compliant queue to create a homegrown space – QSpace.
- Testing QSpace

- Creating a client/server Compute Server based on QSpace.
- Summary

Chapter 9: Creating a JMS protocol Handler 181

- An overview of the Java Protocol Handler architecture
- The JMS Protocol Handler
 - o The JmsURLConnection class
- Creating Programs that use the JMS protocol handler
 - o A Sender
 - o A Receiver
- Summary

Chapter 10: Custom JSP tags for JMS 210

- The need for custom tags
- The Custom Tags
 - o The write tag
 - o The read tag
- Testing it out
- Summary

Chapter 11: Using JMS with EJB 1.1 223

- Introduction
- JMS As A Resource
- Asynchronous EJB
 - o The "Wrong" Architecture
 - o A "Correct" Alternative Architecture
 - § The AsyncDelegator
 - o The Architecture in Action
 - § The Backup EJB
 - § An example client
 - § Compiling and Running the pieces

Chapter 12: An introduction to the new MessageDriven Bean in EJB 2.0 251

- The Basics of Message-Driven beans
- Creating a Message-driven bean
- The Container Contract
 - o Details of the deployment descriptor
- The Lifecycle of a message-driven bean
- Summary

Appendix A: The JMS Exception Family 261

- Understanding the JMS Exceptions
 - o Standard Exceptions
- Summary

Appendix B: A list of JMS Providers 267

Appendix C: Java Naming and Directory (JNDI) 269

Index 270

To
*my mother for her moral support,
my wife Mala for accepting my computer in our lives,
my son Sagar for being born,
and
God for everything else.*

Acknowledgements

Although only the author's name appears on the cover of a book, in reality a book is the combination of the direct and indirect efforts of many people. At this time, I would like to take the opportunity to acknowledge at least a few of those people.

I would like to give special thanks to Charlie Flowers, Chief Technology Officer at Online Insight, the company I work for. He has been instrumental throughout the entire project by providing me tons of encouragement and excellent technical feedback. He has provided very valuable and unbiased opinions that have helped shape the contents of the book. Thank you, Charlie. I would also like to thank all my coworkers, especially Kurt Rush and Greg Corley for as they put it, "doing all the work, while I wrote the book."

Thanks to the many people at Manning who made my vision of this book a reality. Thanks to Susan Whittaker for guiding me through the initial proposal evaluation phase. Special thanks to Marjan Bace for very patiently explaining to me the meaning of pedagogical writing. I would like to thank Ted Kennedy, Mary Pierges, Lianna Wlasiuk, Syd Brown, and the entire editing team for doing such a great job of making this book what it is today. I would also like to thank all the reviewers for taking the time and effort of reviewing this book and providing extremely valuable and unbiased feedback that has been instrumental in raising the quality of this book.

Finally, I would like to acknowledge my mother and wife for taking such good care of me throughout this entire project, with little things such as reminding me to eat, or take a bath, or even backup my work. My 2¹/₂ year old has been especially understanding as well even though all he knew was "pappa is studying."

About This Book

It is a well-known fact that organizations face tremendous business challenges in today's Internet age. The pace at which technology changes has become unmanageable and it is anybody's guess as to what the next hot technology is going to be. In a time of such uncertainty and opportunity, businesses are willing to invest a lot of time and money to make sure that the work they do today does not have to be thrown away tomorrow. The focus has shifted from portability (although still important) to the much larger issue of interoperability both with legacy applications within the organization and competing/cooperating applications external to the organization. Amongst all this chaos, message-based products have proven to be a boon to software architects/developers tasked with creating such interoperable software.

The introduction of JMS (Java Message Service, current version 1.0.2) by Sun represents a revolution in the world of messaging. JMS allows message queue vendors to expose their features in a portable way and hence increase their market size and at the same time reduces the consumer's risk of being tied to a specific vendor. Thus, JMS enables a win-win proposition for both vendors and consumers. This is in tune with Sun's "portability/interoperability" message. JMS has gained support from industry leaders such as IBM, Oracle, Novell, Sybase, and many more as a result of which there are tons of vendors offering JMS compliant message queuing products. Even IBM offers JMS compliant classes to interface with their world-class message queue, MQSeries.

According to Sun Microsystems,

"JMS is a strategic technology for J2EE. JMS will work in concert with other technologies to provide reliable, asynchronous communication between components in a distributed computing environment."

This places even more urgency on the enterprise developer to learn about this key technology. Unfortunately [for the enterprise developer], there are not even a handful of books available on the market that cover JMS in sufficient detail. I hope to fill that void with this book.

I have organized this book into three parts. The first part entitled "Getting Started" consists of 4 chapters that covers many different aspects of JMS. Chapters 1 and 2 provide a gentle introduction to JMS that will be useful to a wide range of people; from the highly technical to the merely curious, while chapter 3 goes into detail about the various architectural pieces that make up JMS and explain how they fit together. Chapter 4 explains complex issues such as multithreading, transactions, security, etc.

The second part focuses on JMS messaging. Chapter 5 covers a central concept of JMS (and of messaging systems in general) – messages. It goes into details about the JMS message structure and the different types of messages. Chapter 6 covers the three messaging styles of JMS in detail.

Finally, part three puts JMS in the context of the real world. Chapter 7 goes into the details of using JMS with XML. Chapter 8 introduces the concept of space-based programming and explains how it can be used to solve many of the problems associated with distributed programming. I also go into details of using any JMS compliant product to create your own space implementation. Chapter 9 uses the façade design pattern to help system architects ease the transition of their development organizations to using JMS. Instead of creating a regular library, I present an alternative technique based on Java's protocol handler architecture. Chapter 10 takes this façade one step further for JSP developers by creating JSP custom tags based on the JMS protocol created in chapter 9. Chapters 11 and 12 focus on using JMS with EJB. Chapter 11 creates an entire framework for using JMS with EJB 1.1, while chapter 12 introduces the new Message-driven beans in EJB 2.0.

My goal in this book is two fold. First, to educate the reader about the JMS specification and second, to show how to use this knowledge to create architectural pieces/applications that are vendor independent. Therefore, I will not go into any details about vendor specific features that are not directly related to how JMS works. For example, one such feature is how different JMS providers implement load balancing and fault tolerance. Every vendor implements this differently and there are simply too many vendors out there to give justice to any one implementation. Instead, my goal is to give you enough knowledge so as to let you make an informed decision while evaluating and selecting a vendor. At the same time, there are aspects of load balancing and fault tolerance that can be achieved by JMS alone irrespective of the vendor. I will concentrate on these aspects. For example, the discussion in chapter 8 about space-based programming using JMS presents an architecture that can be used for creating fault tolerant and naturally load balanced distributed systems and will work with any JMS provider.

And finally, I hope that you enjoy reading this book as much as I have enjoyed writing it.

Chapter 1

An Introduction to the Java Message Service (JMS)

1. Setting the Stage

With the advent of the Internet, distributed computing has become even more important to organizations seeking to create flexible and scalable enterprise applications. A distributed system implies that different parts of the system can be distributed across different machines. These machines may be in the same room or may be in different countries across the globe. The machines are located where they are needed and the different parts of the distributed system are on the machine that is most suited for that part of the system. Creating distributed systems is hard. Think about how hard it is to get a (non-distributed i.e. single executable) complex application to work on a single machine. Now think about the numerous factors that get introduced when the same application is broken up into pieces and installed on multiple machines. Factors such as disparate machine architectures (e.g. Intel Vs Alpha), disparate operating systems, network bandwidth (i.e. the speed at which data can be transferred from one machine to the other), and the multitude of reasons due to which the network can fail, all have to be considered now. In short, the complexity of a distributed system is exponentially higher than the equivalent non-distributed one.

A distributed system itself can be logically divided into at least two pieces: the actual business/functional code and the infrastructure/plumbing code. The business code pertains to the actual function that you are trying to achieve and is independent of whether the system is distributed or not. On the other hand, the infrastructure code is very dependent on whether the system is distributed. If the system is not distributed this code almost disappears. In a distributed system though, the infrastructure code can be extremely complex and may be even larger (in proportion) than the actual business code. The primary objective of this infrastructure code is to transfer data back and forth from one part of the distributed application to another. How this transfer actually takes place depends on how the infrastructure code is implemented.

Note that the infrastructure code does not accomplish any business objective. Process reengineering folks would use the term "non-value adding" work for this type of code (i.e. code that does not provide value to the end user per se) and recommend eliminating this code completely. As software developers we know that we cannot eliminate this code, so the next best alternative is "code reuse". Luckily, since this infrastructure code is only dependent on the distribution and not the business aspect of the system, it is very reusable. Not too ago, software scientists spent a lot of time and effort creating their own distributed libraries (using sockets, for example). These libraries required a lot of maintenance, debugging, and testing, and were a cause of a lot of frustration in software organizations. The software community has matured a lot since then and we now have standards such as DCOM from Microsoft, RMI from Sun, and CORBA from OMG for creating distributed systems.

The infrastructure/plumbing code in the discussion above is commonly referred to as middleware, which is what I will be calling it as well from now on. Based on the prior discussion, middleware can be formally defined as:

"The wide range of software layered between applications and an operating system that provide specialized services and interoperability between distributed applications."

Or, in simpler terms, middleware is the software used to connect software applications to one another. There are *two* fundamentally different types of middleware based on the approach used

by the middleware to transfer the data between the distributed software applications. These are Remote Procedure Call (RPC) based middleware and Message-oriented Middleware (MOM). Let's take a more detailed look at each one.

1.1 RPC Based Middleware

Consider two [good mannered] people talking over the phone. One person starts talking while the other listens until the one talking finishes. The other person processes the information and responds to the first person. All this time the first person has been patiently waiting for the second person's response. If you understand this scenario, you understand the gist of RPC based middleware. That is, the software application that uses RPC based middleware to transfer data to another software application has to wait (i.e. block, in technical terms) until the latter application is done processing the data. Thus with this type of middleware, the communication proceeds in a lock step, synchronized manner and the communicating processes are tightly coupled to one another. Examples of such middleware include Java RMI, OMG's CORBA or Microsoft DCOM.

1.2 The Almighty MOM?

Now let's take a look at a radically different type of middleware, popularly known as MOM. Assume that you've just written a letter to your friend and send it to him via snail mail (such as the U.S. postal service). After sending the letter you obviously do not wait to receive his response to your letter before doing anything else. Instead, you go on with your life. At some point you may get a response from him, but in the meantime your actions did not depend upon this response. If you followed this example, you understand the basics of how a MOM works. The idea behind a MOM is extremely simple, which also leads to its power and widespread popularity. Basically, between any two distributed parts of a system that need to communicate i.e. transfer data, you install a third "intermediary" system. Now instead of transferring data between each other, these distributed parts transfer data to and from the intermediary. This intermediary is the MOM. In more technical terms we have decoupled the communicating applications from one another. This is referred to as asynchronous communication. Thus MOM enables asynchronous communication. There is another aspect of MOM that makes it even more critical to successful distributed applications and hence deserves attention. If for any reason the latter application is unreachable, such as due to a network problem or simply because the application is currently not running, the MOM will take on the burden of keeping track of the undeliverable messages, most likely by maintaining them in a queue, and later deliver these messages when it becomes possible. *This allows the applications in a distributed system using MOM to have completely disjointed lifetimes.* This is an important point and deserves further explanation via an example.

Consider the following scenario: A salesperson is filling in customer orders on his laptop while flying back to the corporate office in an airplane. The laptop is not connected to the central computer and so the orders cannot be processed at that time. However, both the laptop and the central computer are equipped with MOM software. So even though the laptop cannot communicate with the central computer at the time the orders are being entered, the MOM on the laptop keeps track of these "order messages". Once the salesperson is back in his office and hooks up to the corporate intranet, the MOM on the laptop fires off these stored messages to the MOM software on the central computer. The central computer receives and processes these messages. This is a classic example of where the facilities provided by the MOM are needed. The MOM takes care that the messages are not lost, or delivered out of sequence, or duplicated.

Remember

The salesperson example illustrates a major difference between RPC and MOM based middleware. In RPC, applications are coupled in time i.e. the applications must have overlapping lifecycles in order to be able to communicate, while in the case of MOM, the applications can/may have completely disjointed lifecycles and in most cases don't have any knowledge of one another.

The additional flexibility and fault-tolerance offered by leveraging a MOM does not come for free though, as it involves more work for the application developers. Having said that, the extra effort is well worth it in *most* cases. Note that I say *most*, not all cases since the choice of which middleware to use is very application/domain specific. For example, if the application absolutely cannot proceed without a response then RPC based middleware makes more sense. Not only may this reduce programming complexity, but it also may perform better by avoiding the overhead associated with MOM.

2. Introducing JMS

2.1 What is it?

So where does JMS fit into all this? It will all be clear in a moment, but first let me present a definition of JMS.

"JMS is a specification that defines a set of interfaces and associated semantics, which allow applications written in Java to access the services of any JMS compliant Message MOM product."

Quite a mouthful!

Dissecting this definition reveals the following:

- JMS is only a specification and not an actual product implementation.
- Since JMS is a specification, it only defines the interfaces and their semantics.
- These interfaces are used to interface with any JMS compliant MOM product. Thus, JMS is only useful if there are any compliant MOMs out there. Fortunately there are plenty of compliant MOM products available in the market, including MQSeries from IBM, SonicMQ from Progress Software, FioranoMQ from Fiorano, and many more. A JMS compliant MOM is known as a JMS provider.
- Only applications written in Java can use JMS to access the JMS compliant MOM.

Thus, put more simply,

"JMS is an API used to access the facilities of a MOM from a Java application."

2.2 And why was it created?

With all the power that messaging systems have to offer (as described in section 1.2), it is not surprising that there are many such systems available in the market, each one with its own set of advantages and disadvantages and some more popular than others. Examples of the more popular messaging products include IBM's MQSeries and TIBCO's Rendezvous. That's the good

news, since competition is supposed to bring out the best in terms of price and quality. Here's the sad news: each product has its own interfaces and APIs and behaves slightly differently than others, even for [supposedly] similar features.

Consider the following scenario:

A client developing an enterprise application identifies the need for a messaging product and creates a list of requirements that must be met by a vendor. After evaluating several of the popular choices available in the market, the client selects a vendor that best meets their requirements. The client integrates the messaging product into its offering. Then, a year later, another vendor with a better messaging product comes along or a new requirement comes up that the existing messaging product will not [easily] satisfy. Does this sound familiar? In most cases, this is just considered tough luck (or is this Murphy's Law?). The client cannot change vendors because it is too tightly coupled with the vendor, but this is also why a vendor cannot get new clients. Which came first, the chicken or the egg?

This is where JMS comes into the picture. JMS attempts to solve these problems by offering a uniform set of interfaces and associated semantics for messaging systems. In essence, JMS allows clients to get a uniform view of all JMS compliant messaging product providers, or JMS providers, as shown in figure 1.

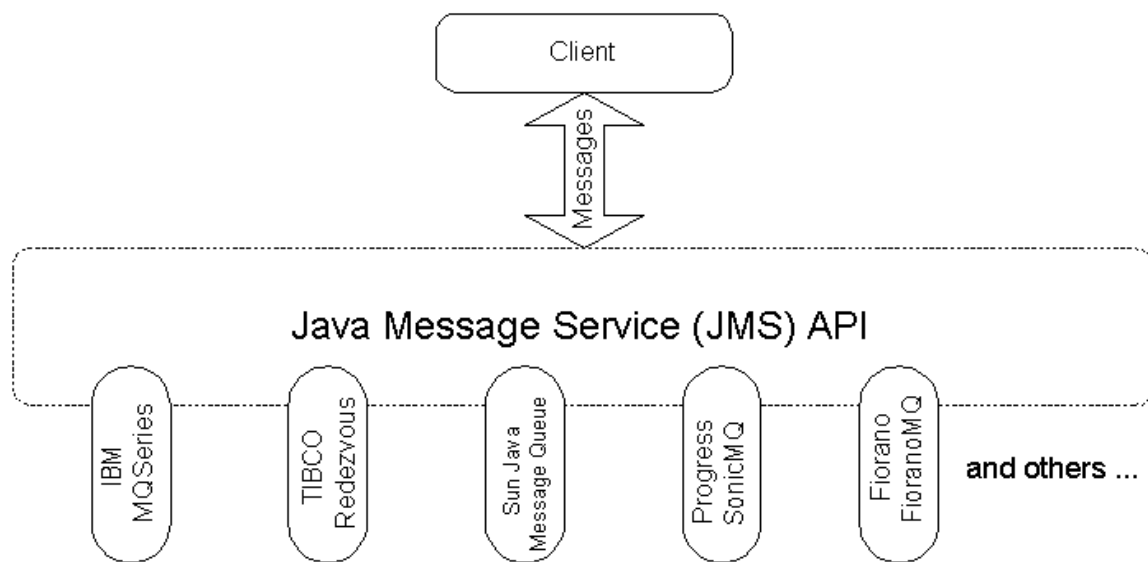


Figure 1: The JMS Architecture¹

Figure 1 illustrates how the JMS allows message queue vendors to expose their features in a portable way and hence increase their market size and at the same time reduces the consumer's risk of being tied to a specific vendor. In this figure the client can use MQSeries, Rendezvous, SonicMQ, or any other JMS compliant provider without any change at all. *Thus, JMS enables a win-win proposition for both vendors and consumers.*

It is important to point out that JMS is not a specification that a few "geeks" at Sun Microsystems came up with one late night after lots of coffee. On the contrary, JMS is the result of many industry

¹ TIBCO Rendezvous does not currently support JMS, but plans are underway to support JMS in early 2001 along with support for EJB2.0

leaders working together over an extended period, with a common goal in mind: *to achieve a uniform enterprise level messaging API*. A number of important industry players, such as Allaire, BEA systems, Fiorano Software, Progress Software, etc. initially collaborated with Sun to define the first draft of the JMS specification. In addition, many comments were received from other companies, government and educational organizations, and others during the three-month public review period. That is one of the key reasons of JMS's widespread acceptance; the other being that it is really well defined, as you'll see in this book. Currently many industry leaders, such as Oracle, Sybase, Novell, IBM, etc. endorse the JMS specification. See appendix B for a complete list of vendors providing JMS compliant messaging products.

It is also important to understand that JMS does not represent [or claim to be] the union of all features available across all messaging products in the market. That would be too impractical. The specification would be too bulky and cumbersome for any one vendor to support. Plus, it would be too complicated for many developers to [quickly] comprehend in this "Internet" time. Most importantly, it would result in message products that are extremely heavyweight and that support too many features for most common applications.

On the other hand, JMS is not merely an intersection of all common features of existing messaging products in the market either. Instead JMS defines a common set of enterprise messaging concepts and facilities that are crucial to implementing sophisticated messaging applications. In this respect JMS is analogous to JDBC. Just as JDBC allows uniform access to many different relational databases, JMS allows uniform access to many different messaging products.

2.3 JMS is part of the Java 2 Enterprise Edition (J2EE)

JMS itself is a very powerful specification, but as part of the J2EE platform its power increases exponentially. Remember the adage "*The sum is greater than the parts*"? Well, it applies here as well. When Java first came out in the mid 90's, it was simply a language that enabled software developers to create applications that could run on any platform that had a Java virtual machine. Since then Java has evolved tremendously, so much so that it is no longer just a language but a platform that supports application development. This platform, called the Java 2 Platform, Enterprise Edition (J2EE), enables the creation of solutions for developing, deploying and managing multi-tier server-centric applications. J2EE utilizes the Java 2 Platform, Standard Edition to extend a complete, scalable, stable, secure, and fast Java platform to the enterprise level. It delivers value to the enterprise by enabling a platform, which significantly reduces the cost and complexity of developing multi-tier solutions, resulting in services that can be rapidly deployed and easily enhanced. Without going into too much detail, J2EE provides the following benefits:

- ? A unified platform for building, deploying and managing enterprise-class software without locking users into a vendor specific-architecture. This results in less maintenance and upgrade headache for IT and is key to the success of any enterprise-class application.
- ? A platform that will allow enterprise-class application the ability to run anywhere. This is because the platform is based on Java and the "*Write Once, Run Anywhere*" philosophy².
- ? A platform with a complete range of readily available enterprise-class services. Think of an enterprise application developer as an expert craftsman with his toolkit of available tools. That's what J2EE provides – a toolkit to the enterprise developer. As I'll discuss below JMS is one of these tools/services.
- ? A single easy-to-learn blueprint programming model for J2EE. The J2EE Blueprints are the best practices philosophy for the design and building of J2EE-based applications. These best practices are derived from years of experience from the best developers in the industry. These design guidelines provide two important things.
 - o First, they provide the philosophy of building n-tier applications on the Java 2 platform.

² It should probably be "Write Once, Test Everywhere, and then Run Anywhere", but I guess that doesn't have quite the same ring to it, does it?

- Second, they provide a set of design patterns for designing these applications, as well as a set of examples or recipes on how to build the applications.

What's most relevant to us though, is the fact that *JMS is a strategic component of the J2EE platform*. Developers creating applications within the J2EE platform can leverage the power of JMS along with other powerful and strategic J2EE technologies, which include Enterprise JavaBeans (EJB), JavaServer Pages (JSP), Java Naming and Directory Interface (JNDI), Java Transaction API (JTA), Java Database Connectivity (JDBC), J2EE Connector, Servlets, XML, and CORBA. The strategic role that JMS plays in the J2EE platform is even more evident with the EJB 2.0 specification, which supports the integration of JMS in the following two ways:

- As a resource available to beans
This capability has actually existed since EJB 1.1. Both session and entity beans are RPC-based components, which is an excellent architecture for assembling transactional components. In some cases, however, as we've already discussed earlier, the synchronous nature of RPC becomes a "handicap", which is precisely why JMS was made available as a resource to these beans. Using JMS providers, EJB developers could overcome this handicap and simulate a bean with asynchronous calls. I will discuss this in great detail in chapter 11.
- As a `MessageDrivenBean`.
As is common with all initial specifications, the creators of the EJB specification have addressed the "synchronous handicap" identified above in the EJB 2.0 specification. In order to provide a standard solution to this problem, EJB 2.0 has introduced a completely new enterprise bean type, the `MessageDrivenBean`, which is designed specifically to handle incoming JMS messages. I will discuss Message-driven beans in detail in chapter 12.

2.4 Common Misconceptions about JMS

Till now I've discussed what JMS is and why it is so important to any enterprise level software developer striving to create applications with the J2EE platform or otherwise. Equally important however, is a discussion on *what JMS is not*. Like all other acronyms in the technology field, JMS has acquired its own set of myths and misconceptions over time. It is very important that these misconceptions get cleared/busted early on so as to facilitate the learning of the advanced concepts in the later chapters. In fact, I think it is important enough to warrant a separate section in this chapter. In this section, I will attempt to clear up nine of the most common misconceptions associated with the JMS specification.

? Misconception #1: JMS is just another Mail API.

By now it must be obvious that JMS is *not* another Mail API. Sun already has that covered with the JavaMail API. The JavaMail API provides a set of abstract classes that models a mail system, which is meant to provide a platform independent and protocol independent framework to build Java-based mail and messaging applications. That does not mean that you could not develop a mail-based application, such as pine, by using a JMS compliant messaging product, but the benefits to be gained by doing so are questionable. After all why go through all the trouble of doing with JMS what the JavaMail API already does?

? Misconception #2: JMS is an actual messaging product.

This is a big one. JMS is a specification and *not* an actual product. A JMS provider such as IBM, Progress Software, or even Sun provides a messaging product that implements the specification. As discussed previously, JMS allows MOM vendors to expose their features in a portable way and hence increase their market size and at the same time reduces the consumer's risk of being tied to a specific vendor.

? Misconception #3: JMS specifies a distributed version of the Java event model.

Java defines a very elegant event model that can be used by objects within the same VM. JMS is not a distributed version of this Java event model. In fact, I would not consider JMS an event model at all, although it can be used to simulate one, with some work of course.

? Misconception #4: JMS offers synchronous messaging and notification of message delivery.

JMS does not standardize synchronous message delivery and/or notification of delivery, such as by defining a set of system messages. Let's think about synchronous delivery for a moment. If synchronous delivery is *really* required, using RPC may be a more desirable solution. After all that is *exactly* what RPC does. A JMS compliant messaging product can be made to simulate synchronous delivery, but again through a lot of hard work that bypasses the benefits bestowed by JMS in the first place. As far as notification of delivery, applications using JMS providers are free to define their own application specific acknowledgement messages, if they desire. Such a feature is defined in the specification itself.

? Misconception #5: JMS is a replacement for the CORBA Notification service.

JMS is not a replacement for the CORBA Notification service. For example, JMS does not offer subscription notification. This is a feature available in the CORBA Notification service. If this feature *were* available in the JMS, it *would* allow publishers of messages to know if there are interested subscribers for the message. Intelligent publishers could then use this information to only publish the message if there were any interested subscribers. However, using this feature may create problems in many situations involving the use of messaging products. Let's go back to our example of the salesperson filling in orders offline. *Since the salesperson is offline, there would be no subscribers for the "order" messages, but yet they still need to be published.* So instead of complicating matters by specifying a subscription notification feature, the JMS specification leaves it up to each JMS provider to minimize the overhead associated with messages for unsubscribed topics. In a similar vein, the JMS does not specify a repository for storing message type definitions, as available in with the CORBA Notification service. A detailed discussion of the CORBA notification service is beyond the scope of this book. Please refer to the references at the end of the book for more information about this service.

? Misconception #6: JMS specifies elaborate load balancing and fault-tolerance schemes.

JMS does not specify any load balancing/fault-tolerance schemes. These features are left up to each individual JMS provider. Therefore, providers are not required to support these features. Most likely such providers will not survive long in today's competitive market without including these features in their product. The point to remember here is that *these features will be vendor specific.*

? Misconception #7: JMS defines a complete API for administering a messaging product

JMS does not provide an API for administering a messaging product. One possible reason is that providers may have their own unique features and setup requirements, which JMS cannot predict. However, JMS does include two administrable objects, which I will discuss in chapter 2. Yet no API for administering these objects is included either; the reason for which will become clear in chapter 2.

? Misconception #8: JMS defines a protocol for secure access to messages.

JMS does not specify an API for controlling the privacy or integrity of messages. It does not attempt to define [yet another] access control/authentication/authorization protocol. Instead,

individual providers are free to implement their own security features. Once again, most vendors will implement such features in their product, even if it is just to be competitive in the market. The point to remember is that as in the case of load balancing and fault tolerance features, *these features will also be vendor specific*.

? Misconception #9: JMS defines a format in which the message is transmitted over the wire.

JMS does not define the wire protocol for messaging. For example, while the OMG has specified the General Inter-ORB Protocol (GIOP) and its mapping on TCP/IP, the Internet Inter-ORB Protocol (IIOP) for CORBA, Sun has done no such thing for JMS.

3. Summary

In this chapter, we looked at what the JMS specification is and what it's not. Remember, JMS is not an actual product implementation, rather it is simply an API specification that other MOMs comply with. These compliant MOMs are called JMS providers. *Furthermore, JMS selectively specifies only the most critical pieces required for interoperability between messaging products.* This encourages vendors to embrace the JMS specification allowing them to differentiate themselves from others based on features such as load balancing, fault-tolerance, security, and administrative ease.

We also looked at how the JMS is related to other key Java technologies and why it's so important. To recap, JMS is compelling for *four* main reasons:

- ? It is the first enterprise messaging API that has achieved wide industry support.
- ? It simplifies the development of enterprise applications by providing standard messaging concepts and conventions that apply across a wide range of enterprise messaging systems.
- ? It leverages existing, enterprise-proven messaging systems.
- ? It is an integral part of the J2EE platform where it will work in concert with other key Java technologies such as EJB, JSP, JNDI, JTA, JDBC, J2EE Connector, Servlets, XML, and CORBA to provide reliable, asynchronous communication between components in a distributed computing environment.

Finally, we also busted nine common misconceptions about JMS. It is important that these misconceptions get cleared early on to facilitate the learning of the advanced concepts in later chapters.

In the next chapter I will discuss the basic pieces of the JMS architecture and how they fit together. We will also get our hands dirty with some sample applications that use the different messaging styles in JMS.

Additional Material for this chapter (Note this is not a heading)

Sidebar: Characteristics of a Message based system

A message-based system i.e. a system that uses a MOM exhibits three key characteristics, which are identified and explained below:

- Scalability
Most messaging products will buffer messages until they can be delivered or the receiver is ready to receive the message. Thus, receivers can service the requests at their own [steady] pace without getting overloaded with work all at once.
- Reliability
In most case message based systems are more reliable than synchronous RPC based systems, because as mentioned above most messaging products will store messages until they are successfully delivered.
- Real-time enough
Unless you're designing a system that prevents nuclear meltdowns, it is more than likely that a message-based system will be real-time enough to suit your needs. Especially with Internet based e-commerce applications where the network lag largely overshadows any but the most significant processing times, messaging systems offer more than adequate performance.

Si

A popular application of MOM includes integrating "legacy" systems with one another to facilitate efficient and almost real-time sharing of data among these systems. This process of tying together multiple enterprise-level applications to support the flow of information is known as Enterprise Application Integration, or EAI. EAI is about integration and interoperability, two key buzzwords of this Internet era. It's not so difficult to see why. Pick your favorite e-commerce (B2B or B2C) site and think about all it does (or should do). Ideally, it should take you through the entire shopping experience. For example, it should

- Help you determine your true needs and requirements
- Based on these needs and requirements, it should search for all available items are appropriate for you i.e. make product recommendations.
- Help you compare and evaluate your different options.
- Take you through the process of placing your order.
- Let you pay the bill in a way that is convenient to you.
- Let you track the order status
- Provide after sales customer support.

How the site integrates these activities is a major part of the site's value proposition to the customer (and market). Integrating these activities is not an easy task. Traditionally, each activity has been supported by its own application with its own database. Some of these systems have been in use for many years. Now vendors are forced to integrate these systems, or at least the data available from these systems to better serve the customer in an increasingly customer-focused market. The Internet has put the power back into the consumer's hands where the competitor is only one click away.

One approach to EAI is to create a point-to-point solution, where each system knows how to communicate with all the systems that it needs to share data with. It is easy to see that such a system would become unmanageable with a large number of systems. A better solution is to use a message brokering architecture as shown in figure 1.

As seen in figure 2, the message broker is essentially a MOM with built in intelligence for routing and transforming messages. A message broker may have a sophisticated rules engine allowing for the creation of workflow type information routing sequences. Each system publishes "data events"

in a well-known format³ that other systems can subscribe to. The events that each system publishes are documented and made available with the system documentation. Any number of other systems may subscribe to such events. Such an architecture is much more manageable with a large number of systems.

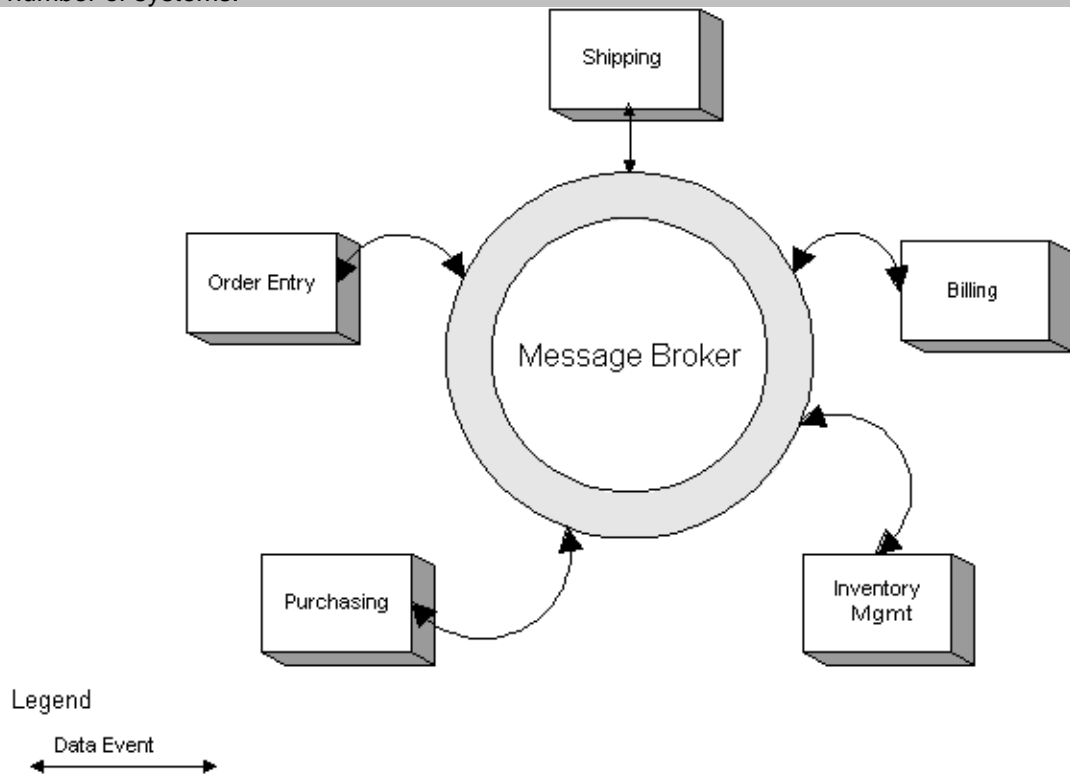


Figure 2: EAI using a message brokering architecture

³ This is where XML comes into the picture. I will talk more about XML in chapter 7.

Chapter 2

Getting Down and Dirty with JMS

Chapter 1 introduced you to the very basics of JMS that would allow you to talk intelligently about JMS in a cocktail party. In this chapter, I am going to take you a step deeper by getting your hands dirty with some actual JMS code.

1. Middleware revisited

In the previous chapter, I spent a great deal of time discussing middleware. Let's take another look at it. Remember there are two types of middleware depending on how the data gets transferred. The middleware of interest to us in this book is message-oriented middleware (MOM) because as I discussed in the previous chapter that's the type of middleware that JMS defines access to. Figure 1 shows that there are two different types of MOM: point-to-point and publish-and-subscribe.

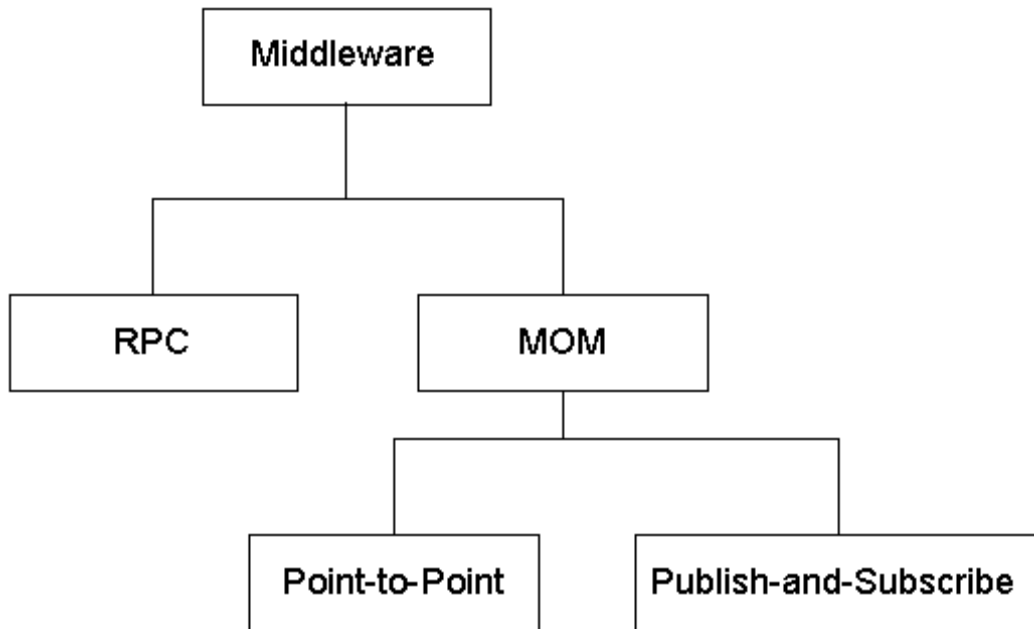


Figure 1: Middleware types

Both types of MOM are popular in the market. Hence, JMS supports both these types. JMS refers to these as messaging styles. Thus, I can say that JMS supports two messaging styles: point-to-point and publish-and-subscribe. Let's take a more detailed look at both of these styles.

1.1 The Point-to-point messaging style

In this model, a MOM is used by two applications to communicate with each other, often as an asynchronous replacement for remote procedure calls (RPC). What exactly do I mean by an asynchronous replacement for RPC? Remember, from our discussion in chapter 1 that RPC is the form of middleware in which all communication between the two applications occurs in a synchronized, lock step manner. But, what if the two applications do not or cannot communicate in this manner. The alternative is to use a MOM that supports the point-to-point messaging style in

place of RPC. The two applications still communicate with each other, but this communication is asynchronous. The example of writing a letter to a friend in chapter 1 was an example of a point-to-point messaging style.

1.2 The Publish-and-subscribe messaging style

In this model multiple applications connect to the MOM as either publishers, which are producers of messages, or subscribers, which are consumers of messages. An important point of difference between the two styles is that a point-to-point system is typically either a one-to-one system, which means one message sender talking to one message receiver, or it is a many-to-one system, which means more than one senders are talking to one receiver. On the other hand, publish-and-subscribe systems are typically many-to-many systems, which means that there could be one or more publishers talking to one or more subscribers at a time.

Publish-and-subscribe systems are very popular in "event-based" systems. An event is an indication of an interesting occurrence in a system that is "published" by that system. Other software applications can "subscribe" for this event. In an event-based system, publishers and subscribers are unaware of and independent of each other. Therefore, a major application of such event-based system is in integrating "legacy" systems to "next generation" systems, i.e. in enterprise application integration (EAI). I discussed the role of messaging in EAI in chapter 1 in the sidebar "MOM and EAI".

1.3 JMS support of the messaging styles

As I mentioned above JMS supports both these styles. However, not all MOMs support both these messaging styles. Therefore, JMS provides a separate domain for each of these styles and defines compliance for each domain. *This means that a MOM can be JMS compliant even if it does not support both messaging styles.* I will discuss how JMS supports both these styles in detail chapter 6. JMS makes a clear distinction between the point-to-point and publish-and-subscribe messaging styles. This means that before a client can use a JMS provider to send and receive messages, the client must decide which messaging style it wants to use. This decision shapes nearly every aspect of how the client system interacts with JMS from then on.

2. Understanding the Players in JMS

Before, we actually look at some basic programs that use JMS, let's look at the primary concepts that all JMS programmers need to know. In the following discussion I will refer to programs that use a JMS provider as JMS clients.

2.1 "Connections" and "Connection Factories"

At the very core of JMS, there is the concept of a "connection." A connection represents a logical connection to the JMS provider. It is intuitively obvious that one of the first things a JMS client must do is obtain a connection to the JMS provider. To obtain this connection each JMS provider provides a connection factory⁴. The connection factory is interesting since even though it is an integral part of JMS, JMS does not standardize what information the connection factory encapsulates or how a client gets the connection factory from a JMS provider. I will discuss the reason for this in the next chapter. There are two types of connection factories: one for point-to-point and another for publish-and-subscribe. Based on the desired messaging style, the client obtains the appropriate connection factory and connects to the JMS provider (all of which can be referred to as "obtaining a connection").

2.2 "Sessions"

Once the client has a connection to the JMS provider the next step is to start a new session. A "session" is a client's own private view of the connection. Each connection may have many sessions in progress at the same time. Just as a connection is necessary to communicate with a

⁴ As in the physical world, a factory is a producer of well known "things". For more details on the factory design pattern refer to "Elements of Reusable Object-Oriented Software" by Eric Gamma, et al.

JMS provider, a session is necessary to communicate with the connection. A simple analogy might help here. Consider the Connection analogous to the main telephone line that serves your entire neighborhood. A "session" would then correlate to your specific phone call that goes across that line.

2.3 "Destinations"

In any message-based system, regardless of the messaging style being used, each message has to be sent somewhere. This "somewhere" is known as a "destination" in JMS lingo. Since messages are sent to a destination, messages are received from a destination as well. JMS does not standardize what information a destination encapsulates. The next chapter will discuss why. JMS does specify how a destination is obtained by a user of the messaging system, which is through a session (discussed above). There are two types of destinations. The type of destination used to send and receive messages depends on the messaging style being used. For point-to-point messaging the destination is called a "queue" and for publish-and-subscribe messaging it is called a "topic". Accordingly, a session that was created for the point-to-point messaging style can only be used to get a queue. Similarly a session that was created for the publish-and-subscribe messaging style can only be used to get a topic.

Once we have obtained a Connection, used it to obtain a Session, and then used the Session to obtain a Destination, we are ready to actually send and receive messages. However, the session itself cannot be used to send and receive messages. Instead, the Session acts as a factory that can be used to create senders and receivers that are used for sending and receiving messages. The next section will help clarify these concepts with the help of some example programs.

3. Hello World – JMS Style?

Although the process of using a JMS provider is fairly straightforward, the number of concepts involved can be overwhelming at first. In order to give you a better feel of what's involved in creating a JMS client and using a JMS provider, I will go through two extremely simple "Hello World" examples – one for each messaging style. The JMS provider that I will be using is Sun Microsystems's Java Message Queue product. An evaluation version is available for download at Sun's website. I have installed this product in my "E:\Program Files" directory.

3.1 A "Hello World" point-to-point example

A point-to-point system consists of one or more sender programs and at most one receiver.

3.1.1 *The Hello World Sender*

First let's look at a sender program. All JMS related classes are located in the `javax.jms` package. Since, we are interested in the point-to-point messaging style, I obtain Sun's connection factory for this style, which I then use to create a connection. This is shown below.

```
// Get a connection factory for the point-to-point style
// i.e. a queue connection factory.
QueueConnectionFactory myConnectionFactory =
new com.sun.messaging.QueueConnectionFactory();

// Use myConnectionFactory to get a queue connection
QueueConnection myConnection =
myConnectionFactory.createQueueConnection();
```

Notice that the class names for the connection factory and connection begin with "Queue." JMS follows a naming convention in which the name of any class used with the point-to-point messaging style begins with "Queue". Similarly the name of any class used with the publish-and-subscribe style begins with "Topic".

The connection is used to create a session as follows.

```
// Use myConnection to create a queue session
QueueSession mySession =
    myConnection.createQueueSession(false,1);
```

The session is used to get the queue called "HelloWorldQueue" as shown below.

```
// Use mySession to get the queue
Queue myQueue = mySession.createQueue("HelloWorldQueue");
```

Finally, the session is also used to create a sender that will be used to send a message. When the sender is created, it is told which queue to send the messages to. This is shown below.

```
// Use mySession to create a sender
QueueSender mySender = mySession.createSender(myQueue);
```

The "Hello World" message is sent using the send method on the sender as follows:

```
// Create the HelloWorld message
TextMessage m = mySession.createTextMessage();
m.setText("Hello World");
// Use mySender to send the message
mySender.send(m);
```

Note that the session is also used to create an empty message. In other words, in addition to being a factory for creating senders and receivers, the session is a factory for producing messages as well.

The complete implementation of the sender program follows:

```
// The Hello World Sender Program: HelloSender.java
import javax.jms.*;

public class HelloSender {
    public static void main(String[] args) throws Exception
    {
        try {
            // JMS setup work.

            // Get a connection factory for the point-to-point style
            // i.e. a queue connection factory.
            QueueConnectionFactory myConnectionFactory =
                new com.sun.messaging.QueueConnectionFactory();

            // Use myConnectionFactory to get a queue connection
            QueueConnection myConnection =
                myConnectionFactory.createQueueConnection();

            // Use myConnection to create a queue session
            QueueSession mySession =
                myConnection.createQueueSession(false,1);

            // Use mySession to get the queue
            Queue myQueue = mySession.createQueue("HelloWorldQueue");

            // Use mySession to create a sender
            QueueSender mySender = mySession.createSender(myQueue);

            // Start the connection
            myConnection.start();

            // Create the HelloWorld message
            TextMessage m = mySession.createTextMessage();
            m.setText("Hello World");
```

```

        // Use mySender to send the message
        mySender.send(m);

        // Done.
        // Need to clean up
        mySession.close();
        myConnection.close();
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
}
}

```

3.1.2 The Hello World Receiver

The receiver is very similar to the sender program. In fact all the work required to obtain a connection factory, the connection, the session, and the queue is exactly the same. In this program, however, the session is used to create a receiver instead of a sender. This receiver is told which queue to receive messages from when it is created. This is shown below:

```

// Use mySession to create a receiver
QueueReceiver myReceiver = mySession.createReceiver(myQueue);

```

To actually receive a message, the program calls the receive method as follows:

```

TextMessage m = (TextMessage)myReceiver.receive();

```

Once a message is received, its contents are printed to standard out. The complete program is shown below. The differences between this program and the sender are highlighted in bold face.

```

// The Hello World Receiver Program: HelloReceiver.java
import javax.jms.*;

public class HelloReceiver {
    public static void main(String[] args) throws Exception
    {
        try {
            // JMS setup work.

            // Get a connection factory for the point-to-point style
            // i.e. a queue connection factory.
            QueueConnectionFactory myConnectionFactory =
                new com.sun.messaging.QueueConnectionFactory();

            // Use myConnectionFactory to get a queue connection
            QueueConnection myConnection =
                myConnectionFactory.createQueueConnection();

            // Use myConnection to create a queue session
            QueueSession mySession =
                myConnection.createQueueSession(false,1);

            // Use mySession to get the queue
            Queue myQueue = mySession.createQueue("HelloWorldQueue");

            // Use mySession to create a receiver
            QueueReceiver myReceiver = mySession.createReceiver(myQueue);

            // Start the connection
            myConnection.start();

            // Wait for the Hello World message
            // Use the receiver and wait forever until the
            // message arrives

```

```

    TextMessage m = (TextMessage)myReceiver.receive();

    // Display the message
    System.out.println("Received the message: " + m.getText());

    // Done.
    // Need to clean up
    mySession.close();
    myConnection.close();
}
catch( Exception e ) {
    e.printStackTrace();
}
}
}

```

3.2 A "Hello World" publish-and-subscribe example

Next, it's time to see how to use the publish-and-subscribe message style in JMS. As you'll discover, it's not very different than using the point-to-point style. A publish-and-subscribe system consists of one or more publisher and subscriber programs.

3.2.1 The Hello World Publisher

First let's look at a publisher program. As before, all JMS related classes are located in the `javax.jms` package. Since in this case we're interested in the publish-and-subscribe messaging style, I obtain Sun's connection factory for this style, which I then use to create a connection. This is shown below.

```

// Get a connection factory for the publish-and-subscribe style
// i.e. a topic connection factory.
TopicConnectionFactory myConnectionFactory =
new com.sun.messaging.TopicConnectionFactory();

// Use myConnectionFactory to get a Topic connection
TopicConnection myConnection =
myConnectionFactory.createTopicConnection();

```

Earlier I discussed the naming convention followed by JMS w.r.t class names. Observe that the class names for the connection factory and connection begin with "Topic."

The connection is used to create a session as follows.

```

// Use myConnection to create a Topic session
TopicSession mySession =
myConnection.createTopicSession(false,1);

```

The session is used to get the topic called "HelloWorldTopic" as shown below.

```

// Use mySession to get the Topic
Topic myTopic = mySession.createTopic("HelloWorldTopic");

```

Finally, the session is also used to create a publisher that will be used to publish the "Hello World" message. When the publisher is created, it is told which topic to publish the messages to. This is shown below.

```

// Use mySession to create a publisher for myTopic
TopicPublisher myPublisher = mySession.createPublisher(myTopic);

```

The "Hello World" message is published using the `publish` method as follows:

```

// Create the HelloWorld message
TextMessage m = session.createTextMessage();

```

```

m.setText("Hello World");
// Use myPublisher to publish the message
myPublisher.publish(m);

```

The complete implementation of the publisher program follows:

```

// The Hello World Publisher Program: HelloPublisher.java
import javax.jms.*;

public class HelloPublisher {
    public static void main(String[] args) throws Exception
    {
        try {
            // JMS setup work.

            // Get a connection factory for the
            // publish-and-subscribe style
            // i.e. a topic connection factory.
            TopicConnectionFactory myConnectionFactory =
                new com.sun.messaging.TopicConnectionFactory();

            // Use myConnectionFactory to get a Topic connection
            TopicConnection myConnection =
                myConnectionFactory.createTopicConnection();

            // Use myConnection to create a Topic session
            TopicSession mySession =
                myConnection.createTopicSession(false,1);

            // Use mySession to get the Topic
            Topic myTopic = mySession.createTopic("HelloWorldTopic");

            // Use mySession to create a publisher for myTopic
            TopicPublisher myPublisher = mySession.createPublisher(myTopic);

            // Start the connection
            myConnection.start();

            // Create the HelloWorld message
            TextMessage m = mySession.createTextMessage();
            m.setText("Hello World");

            // Use myPublisher to publish the message
            myPublisher.publish(m);

            // Done.
            // Need to clean up
            mySession.close();
            myConnection.close();
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
    }
}

```

3.2.2 The Hello World Subscriber

The subscriber is very similar to the publisher program. In fact all the work required to obtain a connection factory, the connection, the session, and the topic is exactly the same. In this program, however, the session is used to create a message subscriber instead of a message publisher. This message subscriber is told which topic to subscribe messages from when it is created. This is shown below:

```

// Use mySession to create a subscriber
TopicSubscriber mySubscriber =

```



```
mySession.createSubscriber(myTopic);
```

To actually receive a message, the program calls the `receive` method. Yes, it is same method that the message receiver called in the receiver program in the point-to-point example above. There is no `subscribe` method. This is shown below.

```
TextMessage m = (TextMessage)mySubscriber.receive();
```

Once a message is received, its contents are printed to standard out. The complete program is shown below. The differences between this program and the publisher are highlighted in bold face.

```
// The Hello World Subscriber Program: HelloSubscriber.java
import javax.jms.*;

public class HelloSubscriber {
    public static void main(String[] args) throws Exception
    {
        try {
            // JMS setup work.

            // Get a connection factory for the
            // publish-and-subscribe style
            // i.e. a topic connection factory.
            TopicConnectionFactory myConnectionFactory =
                new com.sun.messaging.TopicConnectionFactory();

            // Use myConnectionFactory to get a Topic connection
            TopicConnection myConnection =
                myConnectionFactory.createTopicConnection();

            // Use myConnection to create a Topic session
            TopicSession mySession =
                myConnection.createTopicSession(false,1);

            // Use mySession to get the Topic
            Topic myTopic = mySession.createTopic("HelloWorldTopic");

            // Use mySession to create a subscriber
            TopicSubscriber mySubscriber =
                mySession.createSubscriber(myTopic);

            // Start the connection
            myConnection.start();

            // Wait for the Hello World message
            // Use the receiver and wait forever until the
            // message arrives
            TextMessage m = (TextMessage)mySubscriber.receive();

            // Display the message
            System.out.println("Received the message: " + m.getText());

            // Done.
            // Need to clean up
            mySession.close();
            myConnection.close();
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
    }
}
```

4. Compiling and Running the Programs

4.1 Setting up the environment

Copy the following into a batch file called setenv.bat.

```
REM Setup the classpath for Java Message Queue
set JMQ_HOME=E:\Program Files\JavaMessageQueue1.0
set
CLASSPATH=%CLASSPATH%;%JMQ_HOME%\lib\jms.jar;%JMQ_HOME%\lib\jmq.jar;%JMQ_HOME%\lib\jmqadmin.jar
```

Remember, I have installed Sun's Java Message Queue in the "E:\Program Files" directory. You must adjust your JMS_HOME environment variable to reflect your installation directory.

4.2 Compiling the pieces

From a dos prompt in the directory that contains all four programs (HelloSender.java, HelloReceiver.java, HelloPublisher.java, and HelloSubscriber.java) execute the following commands:

```
setenv
javac *.java
```

Here setenv is the same batch file created above.

4.3 Start the Java Message Queue Router

From another dos box in the bin directory of the Java Message Queue installation, start the Java Message Queue router as shown below. The router is a component that is specific to Sun's Java Message Queue and is responsible for routing the messages, providing fault tolerance, security, load balancing, etc.

```
set JAVA_HOME=C:\Program Files\jdk1.2.2
set JMQ_HOME=E:\Program Files\JavaMessageQueue1.0
irouter
```

Once again adjust the environment variables JAVA_HOME and JMQ_HOME to reflect your JDK and Java Message Queue installation directories.

4.4 Running the Sender and Receiver

From a dos box in the same directory as the class files, start a receiver as follows:

```
setenv
set CLASSPATH=%CLASSPATH%;.
java HelloReceiver
```

Now from another dos box in the same directory, start a sender as follows

```
setenv
set CLASSPATH=%CLASSPATH%;.
java HelloSender
```

At this point the receiver dos window should display the message "Received the message: Hello World." If you try starting up a second receiver, while another receiver is waiting for a message, you will see the following error message:

```
javax.jms.JMSEException: javax.jms.JMSEException: Unable to create
receiver for queue as it is already in use. Please close this object
and try again. at
modulus.iagent.jms.IAQueueSession.createReceiver(IAQueueSession.ja
va:101) at HelloReceiver.main(HelloReceiver.java:30)
```

This is because JMS does not allow multiple receivers in the point-to-point messaging style.

4.5 Running the Publisher and Subscriber

From a dos box in the same directory as the class files, start a subscriber as follows:

```
setenv
set CLASSPATH=%CLASSPATH%;.
java HelloSubscriber
```

Now from another dos box in the same directory, start a publisher as follows

```
setenv
set CLASSPATH=%CLASSPATH%;.
java HelloPublisher
```

At this point the subscriber dos window should display the message "Received the message: Hello World." Try starting up multiple subscribers and then run a publisher. Not only do the multiple subscribers run without a problem, but all the subscribers will receive the "Hello World" message from the publisher. This is because the publish-and-subscribe model is a many-to-many model that can have multiple subscribers and publishers.

5. Summary

The simple "Hello World" examples in this chapter serve to illustrate an important point, which is regardless of the messaging style being used the client always follows the same sequence of steps. This sequence is summarized below:

1. Get a JMS provider specific connection factory.
2. Use the connection factory to get a connection to the JMS provider.
3. Use the connection to create a new session. Remember, the type of session created depends on the messaging style.
4. Use the session to get a destination for the messages. A session that was created for the point-to-point messaging style can only be used to get a queue . Similarly a session that was created for the publish-and-subscribe messaging style can only be used to get a topic.
5. Use the session to create a sender that can be used to send messages to the destination created in step 4. The session is also used to create receivers that are used to receive messages from the destination.

In this chapter, I purposefully refrained from getting into the architectural details of JMS while at the same time trying to give you an idea of what a typical JMS client looks like. If it seems fairly simple to you, then that's great – we have an excellent foundation for moving forward. Keep in mind, however, that we have glossed over many complex topics, such as security, message-reliability, message-delivery, transactions and thread-safety. All of these topics are explained in detail in the remainder of this book. In the next chapter, I will start delving into the details of JMS.

Chapter 3

The Basics of JMS

In chapter 2, I gave you your first taste of JMS, but in doing so I glanced over many of the details of JMS. To use JMS more effectively and efficiently it is important to have a good grasp of these details, which is the goal of this chapter.

The Concept of Administrable Objects

In chapter 2, we saw two integral concepts of JMS – the destination and the connection factory – that are not standardized by JMS. I will now tell you why this is so. As I discussed in chapter 1, JMS selectively specifies *only* the most critical pieces required for interoperability between messaging products. As a result, each JMS provider has their own procedures of installing and administering their product and its unique characteristics. However, since the central idea behind JMS is client portability, the JMS specification must somehow isolate these unique characteristics of the individual messaging product from client software. For this purpose, JMS has defined the concept of *Administrable Objects*. These are standard objects that are created and customized by the JMS provider and used by client software to gain access to the messaging product. Both the destination and connection factory are examples of administrable objects. JMS only defines the interfaces for these objects that allow clients to use these objects. These interfaces provide the contract between the client and the JMS provider. As long as this contract is not violated JMS will guarantee client portability and interoperability. *Clients should use these objects only through JMS specified interfaces to guarantee portability across all JMS compliant providers.* Once again remember, it is up to the JMS provider to actually provide the objects that implement these interfaces.

Gaining Access to JMS Administrable Objects

JMS does not specify the method in which clients are to gain access to these administered objects. That then once again becomes a vendor's personal preference, and hence a point at which portability is at stake. For example, in all the examples in chapter 2, I used Sun's proprietary method of accessing the connection factories. This would not work with another vendor such as Progress Software's JMS provider. To help alleviate such problems, JMS does make the following recommendations to JMS providers with respect to administrable objects:

1. JMS providers should make these administered objects available in a JNDI namespace. Refer to Appendix C for a very brief introduction to JNDI and a list of references for more information about this useful API.
2. JMS providers should provide the tools an administrator needs to create and configure administered objects in a JNDI namespace.
3. Implementations of administered objects by JMS providers should be both `javax.jndi.Referenceable` and `java.io.Serializable` so that they can be stored in all JNDI naming contexts. In addition, it is recommended that these implementations follow the JavaBeans design patterns.
4. 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. This allows clients to think of such objects as local Java objects without worrying about locking up resources and jumping through hoops and hurdles to use them.

Unfortunately these are merely [currently not enforced] recommendations and so many commercial JMS providers currently do not follow all four of these recommendations. In chapter 8,

I will show you a technique that I use to gain access to the administrable objects in a portable way that can be used regardless of whether or not a JMS provider follows these recommendations. This is in contrast to the technique used in chapter 2, which was provider (i.e. Sun's Java Message Queue) specific.

Now let's take a more detailed look at the two administrable objects defined by JMS, starting with the destination object.

Destination

JMS does not define a standard address format/syntax. The reason for this is that there are simply too many established enterprise messaging products with different enough addressing formats to make even the attempt to bridge the gap between these a daunting task. JMS does define the concept of a `Destination` object though, which is meant to encapsulate all of the provider specific addressing information. Since this is an administrable object, JMS providers will provide proprietary ways, such as via programmatic interfaces, a GUI, or both, of configuring this information. To the client, this is an opaque structure, the contents of which are not important. All the client knows is that it has access to a destination object that implements the `Destination` interface. The `Destination` interface is shown below:

```
public interface Destination {  
}
```

Basically, it is just a "marker" interface i.e. it is used to identify a valid destination object. In practice (and as seen in chapter 2), one seldom uses this interface directly. Rather depending on the messaging style, one uses either the `Queue` (or `TemporaryQueue`) or the `Topic` (or `TemporaryTopic`) interface. These interfaces are derived from the `Destination` interface and correspond to the queues and topics I introduced in chapter 2. This relation is shown in figure 1. I will discuss these interfaces in detail in chapter 6.

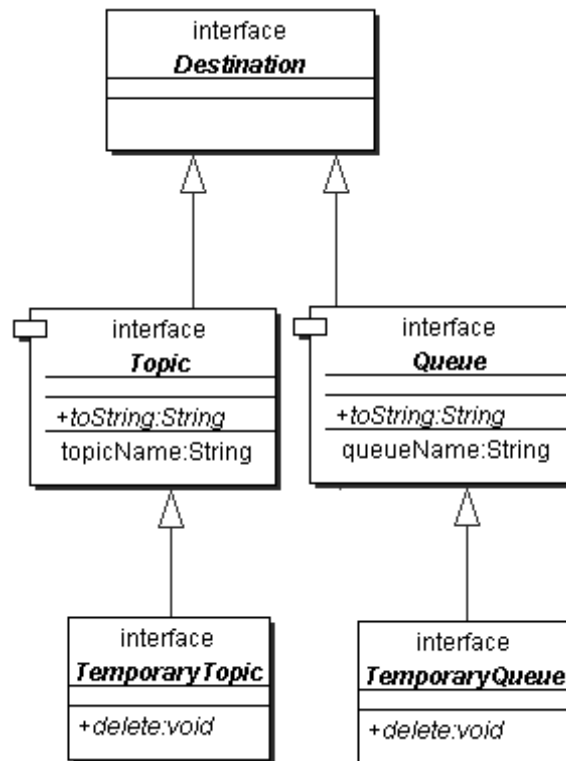


Figure1: The Destination and related interfaces

Now let's take a look at the other Administrable Object – the connection factory

The Connection Factory

Once again, instead of defining a set of standard connection parameters that all JMS providers must use to specify information to connect with that provider, JMS defines the connection factory administrable object that encapsulates all the provider specific connection information. This object implements the `ConnectionFactory` interface, which is shown below:

```
public interface ConnectionFactory {  
}
```

It is also just a "marker" interface. In practice (and as seen in chapter 2), one seldom uses this interface directly. Rather depending on the messaging model, one uses either the `QueueConnectionFactory` or the `TopicConnectionFactory` interface, which are derived from this interface. I will discuss these interfaces in chapter 6. This relation is shown in figure 2

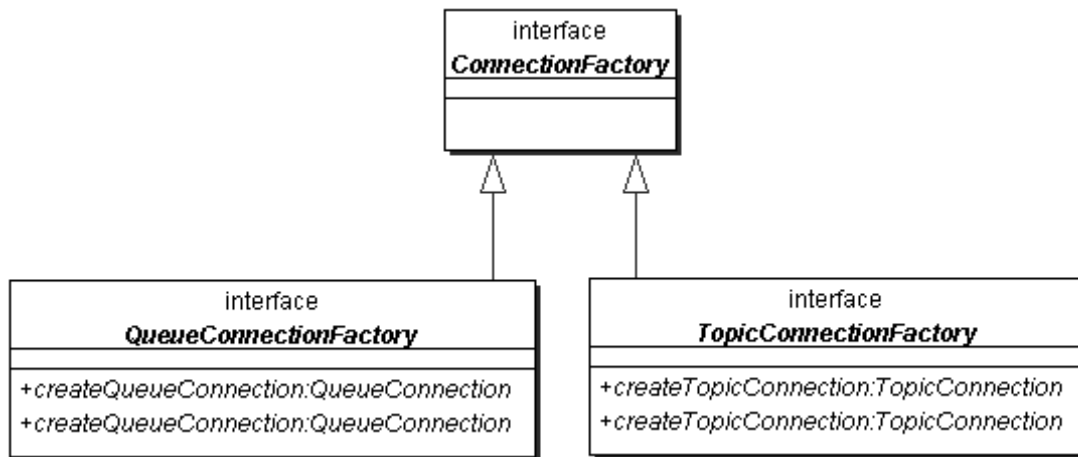


Figure2: The ConnectionFactory and related interfaces

Connecting to the JMS Provider

As we saw in chapter 2, once a client has access to a connection factory i.e. an object that implements a `ConnectionFactory` interface, it can get an actual connection to the JMS provider, in the form of a connection object. The JMS specification states the following about a connection object:

- It encapsulates an open connection with a JMS provider. This may involve the use of resources outside the local Java virtual machine.
- It can specify a unique client identifier. I will touch on this again later.
- If client authentication needs to be done, it should be done during connection setup. This being said, JMS does not define what this authentication means or how it's done, so it is provider specific. It may be as simple as the user specifying a name and password or using the user–login information from the underlying operating system. In any case, the authentication process may be fairly involved and hence the connection should be viewed as an expensive/heavyweight object.

- No messages are delivered by a connection until it has been started. JMS Providers must insure that this is the case because clients that cannot handle asynchronous message delivery depend upon this. I will cover asynchronous message delivery in detail in chapter 4.

A connection object implements the `Connection` interface, which is shown below:

```
public interface Connection {
    String getClientID() throws JMSEException;
    void setClientID(String clientID) throws JMSEException;
    ConnectionMetaData getMetaData() throws JMSEException;
    ExceptionListener getExceptionListener()
        throws JMSEException;
    void setExceptionListener(ExceptionListener listener)
        throws JMSEException;

    void start() throws JMSEException;
    void stop() throws JMSEException;
    void close() throws JMSEException;
}
```

In practice (and as seen in chapter 2), one seldom uses this interface directly. Rather depending on the messaging model, one uses either the `QueueConnection` or the `TopicConnection` interface, which are derived from this interface. I will discuss these interfaces in chapter 6. This relation is shown in figure 3

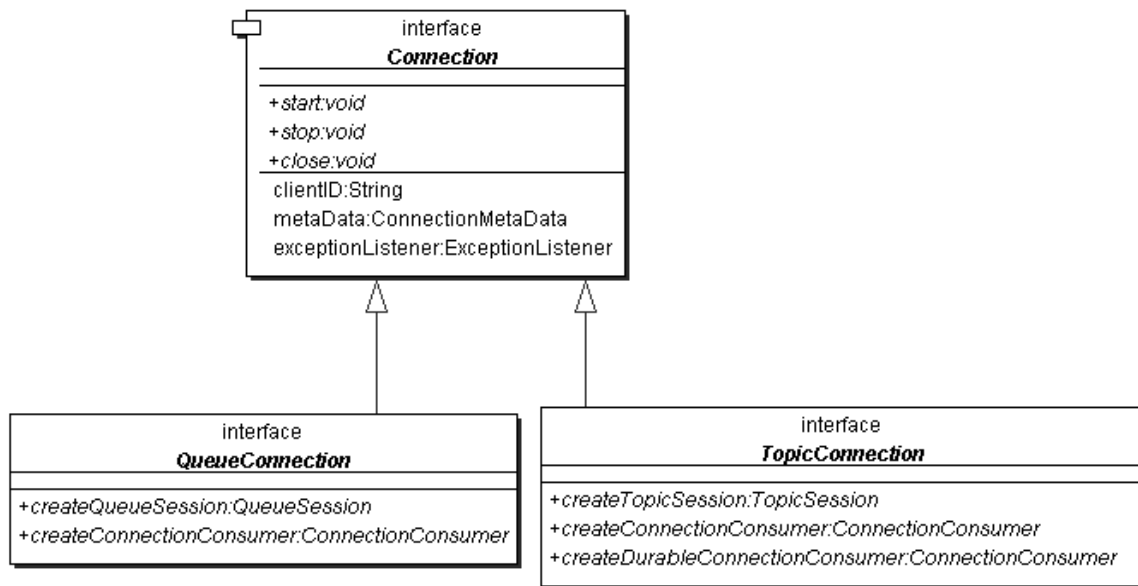


Figure 3: The Connection and related interfaces

The Client Identifier

As mentioned above, each connection object may have a client identifier associated with it. The JMS specification states the *preferred* way in which this client identifier gets set is "transparently" by the connection factory before the connection object is even returned to the client. However, some JMS providers may hold the client responsible for setting the client identifier. If this is the case, the client should use the `setClientID` method on the `Connection` interface. If the identifier has already been set then it cannot be set again and attempting to do so will raise an `IllegalStateException`. If the client identifier must be set by the client, it must be the first thing done on the connection object. If any other action has already been performed on the

connection object, an `IllegalStateException` will be thrown. The purpose of client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. The JMS specification mandates that client state identified by a client identifier can only be 'in use' by one client at a time i.e. A JMS provider must prevent concurrently executing clients from using the client state at the same time. JMS does not specify what action the JMS provider must take if concurrently executing clients attempt to access the client state simultaneously. Therefore, the action taken will be provider specific. For example, one provider may throw a `JMSEException`⁵; another may simply block each client to wait for its turn.

Connection? Tell me about any errors!

The `Connection` interface has a pair of methods that deal with an entity called the "Exception Listener", which is an object that implements the `ExceptionListener` interface shown below:

```
public interface ExceptionListener {
    void onException(JMSEException exception);
}
```

By calling the `setExceptionListener` method on the connection object and passing it an object that implements the `ExceptionListener` interface, we allow the connection to asynchronously notify us of any problems it encounters. It does this by calling the `onException` method of the registered `ExceptionListener` object. JMS specifies that the exceptions delivered to `ExceptionListener` are those which don't have any other place to be reported. For example, if an exception is thrown on a JMS call, it must not be delivered to an `ExceptionListener`. Such an exception must be received by the calling thread itself.

Connection Metadata

The connection object provides access to a connection metadata object via the `getMetaData` method on the `Connection` interface. The connection metadata object implements the `ConnectionMetaData` interface shown below:

```
public interface ConnectionMetaData {
    String getJMSVersion() throws JMSEException;
    int getJMSMajorVersion() throws JMSEException;
    int getJMSMinorVersion() throws JMSEException;
    String getJMSProviderName() throws JMSEException;
    String getProviderVersion() throws JMSEException;
    int getProviderMajorVersion() throws JMSEException;
    int getProviderMinorVersion() throws JMSEException;
    Enumeration getJMSXPropertyNames() throws JMSEException;
}
```

This object provides the following data:

- The latest version (major and minor) of JMS supported by the provider
- The provider's product name and version (major and minor).
- A list of the JMS defined property names supported by the connection. I will discuss JMS properties in detail in chapter 5.

"Starting" the Connection

When a connection object is initially created, it is in "stopped" mode. This means that no messages are being delivered to/from it. It is typical to leave the connection object in stopped mode until setup is complete. At that point the connection's `start` method [on the `Connection` interface] is called and messages begin arriving at the connection's consumers. This setup convention minimizes any client confusion that may result from asynchronous message delivery while the client is still in the process of setting itself up. A connection can immediately be started and the setup can be done afterwards. However, clients that do so must be prepared to handle

⁵ For details of the JMS exception family refer to Appendix A.

asynchronous message delivery while they are still in the process of setting up. As mentioned before, I will discuss asynchronous message delivery in detail in the next chapter.

As we saw in chapter 2, in order to actually anything useful with the JMS provider, a client must use the connection to create a new session, which is the focus of the next section.

The Session Object

As I showed in chapter 2, the connection to a JMS provider acts as a factory of an object known as the session object. This session object implements the `Session` interface shown below:

```
public interface Session extends Runnable {
    static final int AUTO_ACKNOWLEDGE = 1;
    static final int CLIENT_ACKNOWLEDGE = 2;
    static final int DUPS_OK_ACKNOWLEDGE = 3;

    BytesMessage createBytesMessage() throws JMSException;
    MapMessage createMapMessage() throws JMSException;
    Message createMessage() throws JMSException;
    ObjectMessage createObjectMessage() throws JMSException;
    ObjectMessage createObjectMessage(Serializable object)
        throws JMSException;
    StreamMessage createStreamMessage() throws JMSException;
    TextMessage createTextMessage() throws JMSException;
    TextMessage createTextMessage(String text) throws JMSException;
    boolean getTransacted() throws JMSException;
    void commit() throws JMSException;
    void rollback() throws JMSException;
    void close() throws JMSException;
    void recover() throws JMSException;
    MessageListener getMessageListener() throws JMSException;
    void setMessageListener(MessageListener listener)
        throws JMSException;

    public void run();
}
```

That's a big interface. Let's look at the details. The first three static integers – `AUTO_ACKNOWLEDGE`, `CLIENT_ACKNOWLEDGE`, and `DUPS_OK_ACKNOWLEDGE` – in the interface define the various acknowledgement modes. Each message that a JMS provider delivers to a consumer must be acknowledged. If a message is not acknowledged it may be redelivered to the consumer by the JMS provider. The session can be configured to automatically acknowledge each message as it is received/processed. For example, consider the following code fragment:

```
QueueConnection connection = // Get the connection?
QueueSession session = null;
session = connection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

Here, I create a new session object with the automatic acknowledgement mode by passing in `Session.AUTO_ACKNOWLEDGE` as the second parameter, which means that the session will automatically acknowledge the receipt of each message to JMS. This is what I did in all the examples in chapter 2.

Alternatively, a session can be configured not to acknowledge any messages and leave it up to the client consuming the messages to acknowledge them by passing in `Session.CLIENT_ACKNOWLEDGE` as the second parameter above. The client acknowledges a message by calling the `acknowledge` method on it. I will discuss JMS messages in detail in chapter 5. Note that as defined by the JMS specification, acknowledging one message actually acknowledges all messages that the session has consumed. Clients may individually acknowledge messages or they may choose to acknowledge messages in application-defined groups (which is done by

acknowledging the last received message in the group). Remember that if a message is not acknowledged, the JMS provider may redeliver it to the consumer.

The third acknowledgement mode, `Session.DUPS_OK_ACKNOWLEDGE`, instructs the session object to lazily acknowledge messages. So, it is possible that the JMS provider may redeliver a previously received message, and as indicated by the name of the constant (`DUPS_OK_ACKNOWLEDGE`), such duplicates are OK. That means message consumers are coded to deal with such duplicates. The advantage of this mode is that the session has much less overhead associated with preventing duplicates.

The interface contains several methods to create different types of messages. I will discuss each of these different types of messages in detail in chapter 5. For now, you'll recognize the `createTextMessage`, which is the method I used in chapter 2 to create the messages in the sender and publisher programs.

Sessions can be of two kinds: *transacted* and *non-transacted*. In a non-transacted session, messages are sent/published and received/consumed one at a time, while transacted sessions allow grouping of messages in bunches before sending or receiving them. The `getTransacted` method returns a boolean to indicate if the session is transacted. For example, consider the following code fragment:

```
QueueConnection connection = // Get the connection?
QueueSession session = null;
session = connection.createQueueSession(false,
                                     Session.AUTO_ACKNOWLEDGE);
if(session.getTransacted())
    System.out.println("Session is Transacted.");
else
    System.out.println("Session is not Transacted.");
```

This session is not transacted as indicated by the first parameter [`false`] to the `createQueueSession` method call. So, the `getTransacted` method will return `false`, which means that the message "*Session is not Transacted.*" will be sent to standard output.

The `commit` and `rollback` methods are used to commit and rollback the session transaction, if the session is transacted. If these methods are called on a session that is not transacted then an `IllegalStateException` will be thrown. In addition if the `commit` method is called on a transacted session and the transaction actually gets "rolled back", a `TransactionRolledBackException` is thrown. I will discuss more about transactions in the next chapter.

There are a pair of methods, `setMessageListener` and `getMessageListener`, that deal with message listeners. A message listener is an object that implements the `MessageListener` interface shown below:

```
public interface MessageListener {
    void onMessage(Message message);
}
```

A message listener is used to asynchronously receive delivered messages. I will cover asynchronous message delivery in detail in the next chapter as well.

The `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 [and their order] that the session now delivers to the consumer may be different from those which were originally delivered due to reasons such as message expiration and the arrival of higher priority messages. A session must set the redelivered

property of each message it redelivers due to a recovery. I will discuss the redelivered property in chapter 5.

The `run` method is a method derived from the `Runnable` interface and is intended for use by Application Servers. It is an optional method, which means that a JMS provider may decide not to implement it.

As we already know, a session cannot send or receive messages either. Instead it is a factory for creating message senders and receivers, which are called message producers and message consumers respectively. I will discuss both of these in much more detail in the next sections. Also, in practice, one seldom uses the `Session` interface directly. Rather depending on the messaging model, one uses either the `QueueSession` or the `TopicSession` interface, which are derived from this interface. I will discuss these interfaces in chapter 6. This relation is shown in figure 4.

Message Consumers

As discussed above (and as seen in chapter 2), the session serves as a factory of message consumers. A message consumer is an object that implements the `MessageConsumer` interface shown below:

```
public interface MessageConsumer {
    String getMessageSelector() throws JMSEException;
    MessageListener getMessageListener() throws JMSEException;
    void setMessageListener(MessageListener listener)
        throws JMSEException;

    Message receive() throws JMSEException;
    Message receive(long timeout) throws JMSEException;
    Message receiveNoWait() throws JMSEException;
    void close() throws JMSEException;
}
```

A client uses a message consumer to receive messages from a destination object. Examples of message consumers are the queue receiver and topic subscriber objects that we used in chapter 2. A message consumer can be created with or without a message selector. Specifying a message selector allows the client to restrict the messages delivered to the message consumer to those that match the selector. I will discuss the message selector syntax in detail in chapter 5.

There are two ways a client can receive messages from a message consumer:

1. A client can request the next message from a message consumer using one of its `receive` methods. There are several variations of `receive` that allow a client to poll or wait for the next message. A client can choose to call the blocking `receive` method with no parameters (used in chapter 2), or poll the consumer by calling either the `receive` method with a time out or the `receiveNoWait` method that does not wait at all.

Alternatively, a client may register a message listener object with a message consumer by calling the `setMessageListener` method and passing in the message listener as a parameter. Remember, a message listener is an object that implements the `MessageListener` interface. As messages arrive at the message consumer, the message consumer delivers them by calling the message listener's `onMessage` method. Registering a message listener allows clients to asynchronously receive messages without having to block/poll the message consumer.

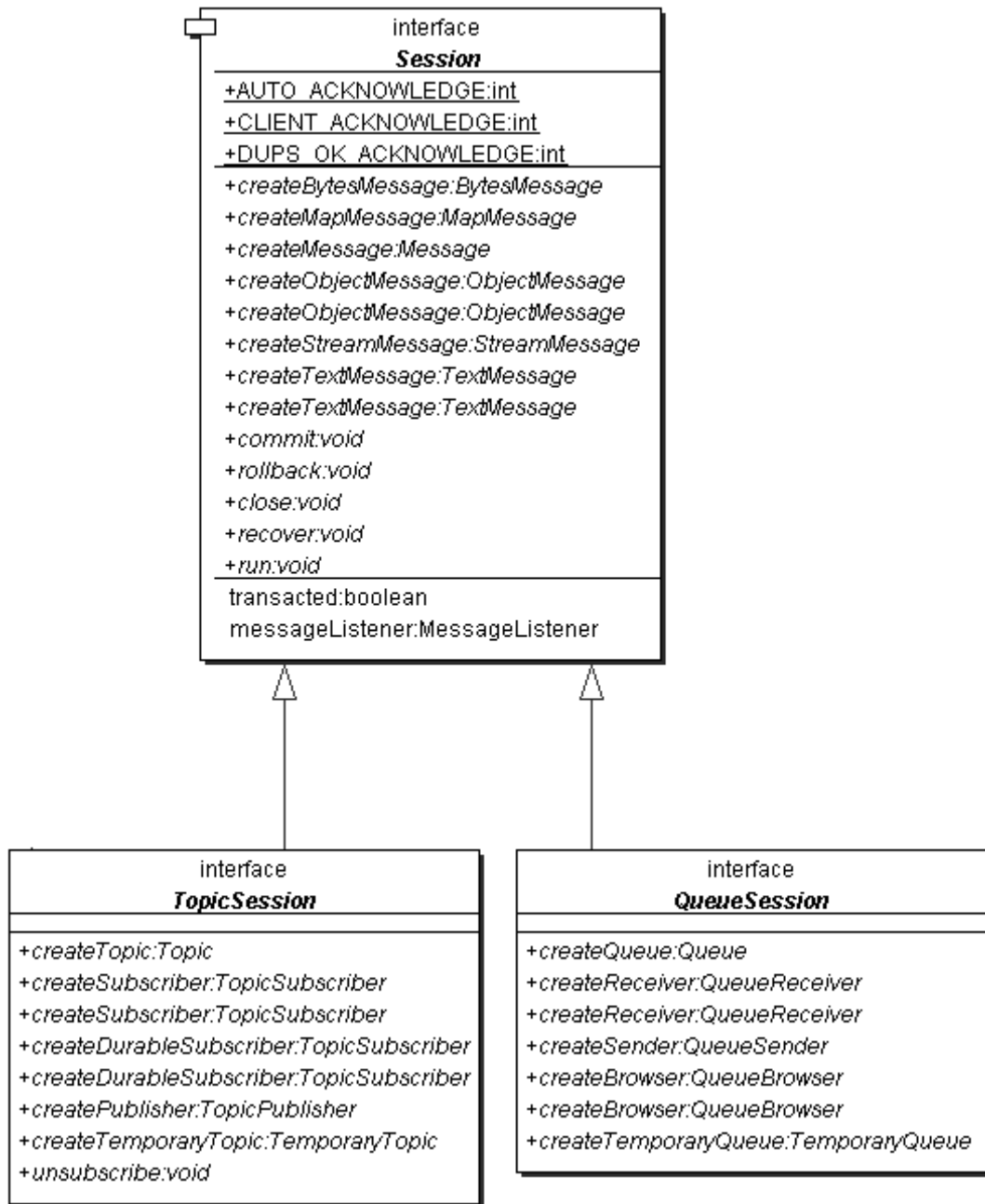


Figure 4: The Session and its related interfaces

Since a provider may allocate some resources on behalf of a message consumer, perhaps even outside the local Java VM. As a result, clients should call the `close` method on the message consumer when it is not needed anymore. Relying on garbage collection to eventually reclaim the resources may not be timely enough. The `close` method call blocks until any `receive` method call or message listener call in progress has completed. A blocked message consumer `receive` call returns `null` when the message consumer is closed, so clients must be prepared to deal with this situation.

For example, if thread 1 in a client is blocked as follows:

```
Message m = consumer.receive();
```

And thread 2 in the same client calls the `close` method on `consumer` as follows:

```
consumer.close();
```

then assuming no messages were available at the time `close` was called, `m` will be equal to null. Thread 1 must detect this and take the appropriate action.

In practice, one seldom uses the `MessageConsumer` interface directly. Rather, one uses either the `QueueReceiver` or the `TopicSubscriber` interface depending on the messaging style being used. Both `QueueReceiver` and `TopicSubscriber` are derived from the `MessageConsumer` interface as shown in figure 5. I will discuss both of these interfaces in detail in chapter 6.

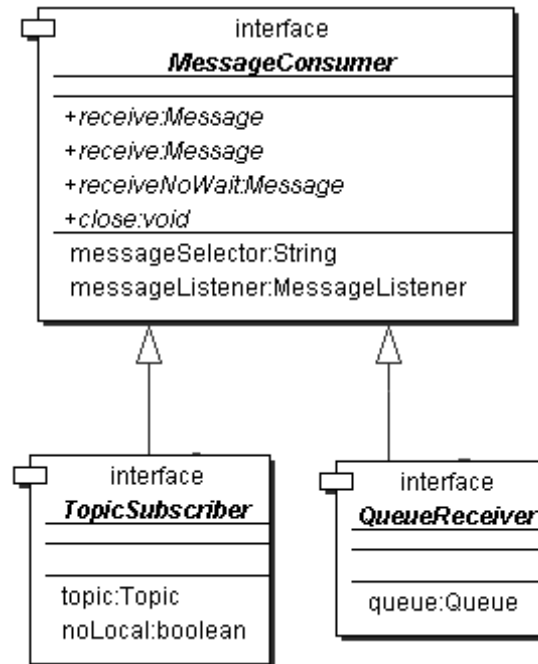


Figure 5: The `MessageConsumer` and related interfaces

Message Producers

A session also serves as a factory of message producers, which are used to send messages. A message producer is an object that implements the `MessageProducer` interface shown below:

```
public interface MessageProducer {
    void setDisableMessageID(boolean value) throws JMSEException;
    boolean getDisableMessageID() throws JMSEException;
    void setDisableMessageTimestamp(boolean value)
        throws JMSEException;
    boolean getDisableMessageTimestamp() throws JMSEException;
    void setDeliveryMode(int deliveryMode) throws JMSEException;
    int getDeliveryMode() throws JMSEException;
    long getTimeToLive() throws JMSEException;
    void setTimeToLive(long timeToLive) throws JMSEException;
    void setPriority(int defaultPriority) throws JMSEException;
    int getPriority() throws JMSEException;
    void close() throws JMSEException;
}
```

As mentioned above, a client uses a message producer to send messages to a destination. Examples of message producers are the queue sender and topic publisher objects that we used in chapter 2. A client has the option of creating a message producer without supplying a destination. If no destination is specified, a destination must be input on every `send` operation. This is very similar to how UDP (Uniform Datagram Protocol) operates. A typical use for this style of message producer is to send replies to requests using the request message's `JMSReplyTo` property. This is known as "Request/Reply" mode of operation. I will discuss two examples of this in chapter 6.

Each message can be associated with a unique identifier. This is known as the *message ID*. Since message ID's take some effort to create and increase a message's size, some JMS providers may be able to optimize message overhead if they are given a hint that a client does not use the message ID. JMS message producers can give this hint if a client calls the `setDisableMessageID` method on the producer with a `true` parameter. These messages must either have message ID set to `null` or, if the hint is ignored, the message ID must be set to its normal unique value.

Similarly each message can be marked with a *timestamp*. 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 a client does not use the timestamp. JMS message producers can give this hint if a client calls the `setDisableMessageTimestamp` method on the producer with a `true` parameter. These messages must either have timestamp set to `null` or, if the hint is ignored, the timestamp must be set to its normal value.

Message producers allow clients the option of specifying a default delivery mode, priority and time-to-live for all messages sent. Individual messages can override these defaults. I will discuss this in chapter 5.

The `MessageProducer` interface provides a pair of methods, `setDeliveryMode` and `getDeliveryMode`, to set and get the default message delivery mode.

JMS supports two modes of message delivery: persistent and non-persistent. Of these two modes, the non-persistent mode is the lower overhead delivery mode because it does not require that the message be logged to stable storage. In case of a JMS provider failure [or even without one] a non-persistent message may be lost. In fact, a provider that discards every non-persistent message would still be JMS compliant, although not very competitive in the market *. The non-persistent mode implies "best effort" semantics, which is open to interpretation by each JMS provider. However, the JMS specification mandates that a provider must deliver a non-persistent message at-most-once. *This means it may lose the message but must not deliver it more than once.*

On the other hand, the persistent mode forces the JMS compliant provider to take extra care to insure the message is not lost in transit due to a JMS provider failure. The JMS specification mandates that a provider must deliver a persistent message once-and-only-once. *This means that the provider must not allow the message to be lost and it must not deliver it more than once.*

Important Sidebar: Persistence != Reliability

Delivery mode only covers the transport of the message to its destination. Retention of a message at the destination until its receipt is acknowledged is not guaranteed by a persistent delivery mode. Clients should assume that message retention policies are set administratively. Message retention policy governs the reliability of message delivery from destination to message consumer. For example, if a client's message storage space is exhausted, some messages as defined by a site-specific message retention policy may be dropped.

By providing two mechanisms for message delivery, JMS allows clients to make tradeoffs between performance and reliability. The non-persistent mode has a performance advantage over the persistent mode, but is at a disadvantage from a reliability standpoint; the persistent delivery mode is more reliable (not 100% though. See sidebar). When a client selects the non-persistent delivery mode it is indicating that it values performance over reliability and that an occasional lost message is tolerable. On the other hand, a client marks a message as persistent if it feels that the application will have problems if the message is lost in transit. Clients use delivery mode to tell a JMS provider how to balance message transport reliability/throughput.

JMS provides a `DeliveryMode` interface with two constants corresponding to the two delivery modes. This interface is shown below:

```
public interface DeliveryMode {
    static final int NON_PERSISTENT = 1;
    static final int PERSISTENT = 2;
}
```

Delivery mode is set to persistent by default. To set the default delivery mode to non-persistent delivery, a client would do something like:

```
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

A client can also specify a default time-to-live value in milliseconds using the `setTimeToLive` method on the `MessageProducer` interface. To calculate the message's actual expiration time this value is added to the time (GMT) the message is actually sent. Even for transacted sends, this is the time the message is sent and not the time at which the transaction is committed. The JMS specification states that a JMS provider should do its best to accurately expire messages, but does not define the accuracy required. At the same time, the specification makes it clear that it is not acceptable for a JMS provider to simply ignore time-to-live. The default time-to-live is set to zero, which is a special value indicating no expiration or unlimited life. To set the time-to-live to 10 seconds, a client would do something like:

```
producer.setTimeToLive(10*1000);
```

The `MessageProducer` interface also allows the client to specify the default priority of sent messages via `setPriority` the method. JMS defines a ten level priority value with 0 as the lowest priority and 9 as the highest. 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. So once again, a provider that simply ignored priorities would be considered JMS compliant but not very competitive. Priority is set to 4, by default. To set this to another value, a client would do something like:

```
producer.setPriority(9);
```

Since a provider may allocate some resources on behalf of a message producer, perhaps even outside the local Java VM, clients should call the `close` method on the message consumer when it is not needed anymore. Relying on garbage collection to eventually reclaim the resources may not be timely enough.

In practice, one seldom uses the `MessageProducer` interface directly. Rather, one uses either the `QueueSender` or the `TopicPublisher` interface depending on the messaging style. Both these interfaces are derived from the `MessageProducer` interface as shown in figure 6. I will discuss both of these interfaces in detail in chapter 6.

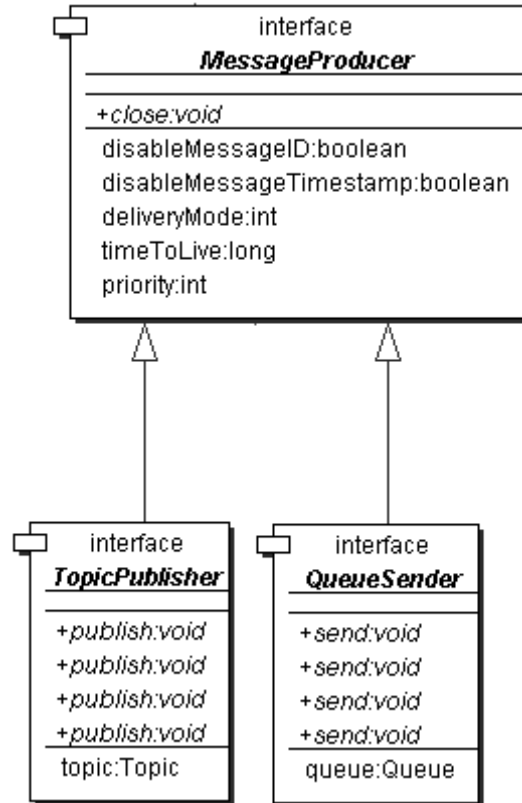


Figure 6: The MessageProducer and its related interfaces

Shutting down Cleanly

All JMS clients must always shutdown cleanly to ensure that all resources used by the JMS runtime, both client and server-side, are released properly and as soon as possible. For example, the sample programs in chapter 2 closed the session and connection before shutting down. Let's look at both of these actions and their implications in detail now.

Closing a Session

Let's focus on the `close` method on the `Session` interface. Since a provider may allocate some resources on behalf of a session outside the local Java VM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough. Following are points to remember with regards to the `close` method on a session:

- The `close` method on a session will not return until its message processing has been orderly shut down. This means that this method will wait until:
 - None of its message listeners are running and,
 - If there are any pending receive (i.e. a blocked call to one of the `receive` methods) all of them return with either a `null` or a message.
- When a session is closed there is no need to close its constituent message producers and consumers. The session close is sufficient to signal the JMS provider that all resources for the session should be released.
- The JMS specification mandates that closing a *transacted* session must rollback its transaction in progress.

- The JMS specification mandates that once a session has been closed an attempt to use it or its message consumers and producers must throw an `IllegalStateException`, with the exception of calls to the `close` method, which are 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 the Connection to the JMS Provider

To temporarily stop the connection's delivery of incoming messages a client may call the `stop` [on the `Connection` interface] method on the connection object. The message delivery can be resumed later by invoking the `start` method. Clients sending messages to a stopped connection are not affected in any way. JMS specifies that the `stop` method call on the connection object must not return until delivery of messages has paused. This means that the client thread calling the `stop` method may block for some time.

As discussed earlier, a connection object is an expensive object that most likely uses resources outside of the local Java virtual machine. Therefore, when a connection should be closed as soon as it is not needed anymore. This is done by calling the `close` method [on the `Connection` interface] on the connection object. A call to the `close` method terminates all pending message receives (i.e. blocking `receive` method calls) on all consumers for that connection. The `receive` calls may return with a message or `null` depending on whether there was a message or not available at the time of the close. If a message is returned, the client cannot call the `acknowledge` method on the message.

Summary

In this chapter, I introduced you to the basic concepts of JMS and showed you how they fit together. These basic concepts allow clients to create and build JMS-based applications that are simple, yet very powerful, scalable, and support reliable distribution across machines and platforms. Most clients will spend a lot of their time interacting directly or indirectly with a session since it has so many responsibilities. To provide a point of reference, I'll summarize the main features of the session object below.

- The session object acts a factory for creating message producers and message consumers.
- The session object also acts as a factory for creating messages.
- The session object defines a serial order for the messages it consumes and produces.
- The session object retains messages it consumes until they have been acknowledged.
- The session object serializes execution of message listeners registered with it.
- The session object supports a single series of transactions that combine work spanning that session's producers and consumers into atomic units that can either be committed or rolled back.

In the next chapter, I will take you beyond the basics and discuss some of the more involved aspects of JMS such as multithreading, transactions, asynchronous message delivery, and more.

Chapter 4

Beyond the Basics

In the previous chapter, I discussed the basic concepts in JMS. In this chapter, I will discuss some of the more involved topics related with these concepts. The topics discussed in this chapter cover a variety of different aspects of JMS.

JMS Transaction Support

JMS transactions adhere to a set of properties known as the *ACID properties*. The word ACID is an acronym that stands for atomicity, consistency, isolation, and durability. Atomicity means that either all of the messages within a transaction are sent/received or none of them are. Consistency means that all messages sent/received within a transaction are in a consistent state. Isolation means that although many transactions may be going on concurrently within the system, it appears to each individual transaction that all other transactions either complete before it or after it. In essence, this property implies that two transactions should not affect each other. Durability means that when a transaction commits, all changes made by it will survive system failures.

Assume that you go to an online bookseller to buy a book. After entering your credit card number, you hit submit. After doing some initial processing such as initial verification of your credit card number, the server sends off two messages; one to the credit card issuer and the other to their warehouse to ship the book. These two messages need to be part of a transaction. Either both of them get sent or neither does. For example, if the first message gets sent and the second one does not, your credit card will be billed, but you will not receive the book. On the other hand if the first message did not get sent, but the second one does, you will essentially get the book for free; a situation that would upset the bookseller.

JMS's support for transactions is built into the session object, which may be optionally specified as transacted, such as shown below:

```
QueueConnection connection = // Get the connection?
QueueSession session = null;
session = connection.createQueueSession(true,
                                       Session.AUTO_ACKNOWLEDGE);
```

In this case, the first parameter passed in to the `createQueueSession` method is `true`, which indicates a transacted session is required.

A transacted session groups a set of produced messages and a set of consumed messages into an atomic unit of work. When a transaction is committed by a call to the `commit` method on the session, the consumed messages are acknowledged as an atomic unit and the produced messages are sent. If on the other hand, the transaction is rolled back by calling the `rollback` method on the session, its produced messages are destroyed (i.e. never sent) and its consumed messages are automatically recovered. I discussed the recovery process in the previous chapter. To recap, the recovery process resets the session to the point after its last acknowledged message. When this happens the session redelivers the messages that are still applicable. However, the messages it now delivers may be different from that which were originally delivered due to message expiration, the arrival of higher priority messages, and of course the transaction rollback.

Note that there are no explicit methods to begin a transaction. Instead, the completion of a session's current transaction, indicated by a call to either the `commit` or `rollback` method, automatically begins the next transaction.

As an example, consider the following code fragment:

```
// assume ts is a transacted session
TopicPublisher tp = ts.createPublisher(topic);
tp.publish(msg1);
tp.publish(msg2);
tp.publish(msg3);
```

In the above code, the messages `msg1`, `msg2`, and `msg3` are not sent immediately. Instead, they are stored by the session until a `commit` or `rollback` method on the session is executed. When

the transaction is committed, `ts.commit()`, the three messages are sent as a single packaged unit. On the other hand, if a rollback is performed, `ts.rollback()`, then all three messages are discarded. Also, as motioned above a `commit` or a `rollback` method call signifies the end of the current transaction; all subsequent operations in the session automatically become part of the next transaction. Thus, a transacted session always has a current transaction within which its work is done.

XA Compliance

JMS does not require that a compliant provider implement transactions to be XA compliant. XA compliant transactions follow the two-phase commit protocol. As an example, transactions in FioranoMQ 3.0, which is a compliant JMS provider, are not XA compliant.

Distributed Transactions

JMS does not require that a provider support distributed transactions. However, if a provider does support such transactions, it should do so 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. Most of the commercial JMS providers do not support distributed transactions and so I will not discuss this topic further.

Message Acknowledgement Revisited

In the previous chapter, I discussed how the session handles message acknowledgement and the three different acknowledgement modes that it supports. It is important to remember that if a session is transacted, message acknowledgment is *always* handled automatically by commit and recovery is *always* handled automatically by rollback. Therefore the only legal acknowledgement mode for a transacted session is `Session.AUTO_ACKNOWLEDGE`.

JMS Message Delivery Styles

JMS supports synchronous, asynchronous, and concurrent delivery of messages. Let's look at each one next.

Synchronous Delivery

A client can request the next message from a message consumer using one of its `receive` methods. As discussed in chapter 3, there are several variations of `receive` that allow a client to poll or wait for the next message. For example,

```
// assume that session is a QueueSession and queue is a Queue.
QueueReceiver receiver = null;
receiver = session.createReceiver(queue);
StreamMessage stockMessage;
stockMessage = (StreamMessage)receiver.receive();
```

In the above code fragment, the `receiver` will wait indefinitely for a message. Alternatively, I could have specified a timeout in milliseconds, such as:

```
// wait for 10 seconds only.
stockMessage = (StreamMessage)receiver.receive(10*1000);
```

Or, no wait at all:

```
// Don't wait?
stockMessage = (StreamMessage)receiver.receiveNoWait();
```

Asynchronous Delivery

Instead of waiting/polling the message consumer for messages, a client can register a message listener with a message consumer. A message listener is an object that implements the

MessageListener interface. I listed the MessageListener interface in chapter 3, which is shown again for reference:

```
public interface MessageListener {
    void onMessage(Message message);
}
```

A message listener can be installed on a session or on a message consumer by calling the setMessageListener method and passing in an object that implements the MessageListener interface, such as:

```
// Assume that tc is TopicConnection?
// Use tc to create a transacted topic session
TopicSession ts = tc.createTopicSession(true,
    Session.AUTO_ACKNOWLEDGE);

// Install message listener on the TopicSession
ts.setMessageListener (new MyMessageHandler());

// create a TopicSubscriber for the topic testTopic
TopicSubscriber tsub = ts.createSubscriber(testTopic);

// Install message listener on the TopicSubscriber
tsub.setMessageListener (new MyMessageHandler());
```

In the above code fragment, MyMessageHandler implements the MessageListener interface as shown below:

```
class MyMessageHandler implements javax.jms.MessageHandler {
    public void onMessage(javax.jms.Message msg) {

        // catch all exceptions
        try {
            // Do something with the message
        }
        catch(java.lang.Exception e) {
        }
    }
}
```

As messages arrive for the consumer, the provider delivers them by calling the listener's onMessage method. Note that the onMessage method is declared as not throwing any exceptions i.e. it does not have a throws clause. It is still possible for a listener to throw a RuntimeException; however, this is considered a client programming error i.e. bad practice. Well-behaved listeners should catch all runtime 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 and is described below:

- If the session's acknowledgement mode is AUTO_ACKNOWLEDGE then the message will be immediately redelivered. The number of times a JMS provider will redeliver the same message before giving up is provider dependent.
- If the session's acknowledgement mode is CLIENT_ACKNOWLEDGE then 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 i.e. call the recover method on the session.
- If the session is *transacted* then the next message for the listener is delivered. The client can either commit or rollback the session by calling the appropriate method on the session. Note that throwing a RuntimeException does not automatically rollback the session.

The JMS specification suggests that a JMS provider should flag a client with a message listener that throws a `RuntimeException` as possibly malfunctioning.

Concurrent Delivery

Clients that desire concurrent delivery can use multiple sessions. As I'll discuss in the section "JMS and Multi-threading" later in this chapter, a single session cannot support concurrent delivery. However, a connection can have multiple sessions and each individual session's listener thread will run concurrently. This means that while a listener on one session is executing, a listener on another session may also be executing. This allows a client to handle concurrent message delivery if it so desires.

For example, consider the following code fragment:

```
// Assume that tc is TopicConnection?
// Use tc to create a transacted topic session
TopicSession ts1 = tc.createTopicSession(true,
                                         Session.AUTO_ACKNOWLEDGE);
TopicSession ts2 = tc.createTopicSession(true,
                                         Session.AUTO_ACKNOWLEDGE);

// create two TopicSubscribers for the topic testTopic
TopicSubscriber tsub1 = ts1.createSubscriber(testTopic);
TopicSubscriber tsub2 = ts2.createSubscriber(testTopic);

// Install message listeners (callbacks) on each
// TopicSubscriber
tsub1.setMessageListener (new MyMessageHandler1());
tsub2.setMessageListener (new MyMessageHandler2());
```

Given this code fragment, the listener on both sessions, `ts1` and `ts2`, could be processing messages concurrently. However, each message is processed in its own transaction and independent of each other. Depending on the situation, this may or may not be the desired behavior. As mentioned earlier, JMS does not standardize distributed transactions, so clients would have to do their own coding to make these separate transactions cooperate.

JMS Message Delivery Order

JMS clients need to understand when they can depend on message order and when they cannot. 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 cannot be controlled by the client and hence must not be critical to the operation of the client system. If order is critical to the client, then the client can always put some extra information into the message to recreate the order at the other end.

Several things *can* affect the order of messages delivered to the end consumer:

- Messages of higher priority may jump ahead of prior, lower priority messages.
- There is also the possibility that a client may not receive a non-persistent message, most likely due to a JMS provider failure.
- The JMS specification only guarantees delivery order within delivery mode. That is, if both persistent and non-persistent messages are sent to a destination, the order of the messages is only guaranteed within delivery mode. This means that a later non-persistent message may arrive ahead of an earlier persistent message; however, it must never arrive ahead of an earlier non-persistent message with the same priority, unless of course the former non-persistent message *never* arrives i.e. was lost.
- A client may use a transacted session to group its sent messages into atomic units. A transaction's order of messages to a particular destination will be maintained. The order of

these messages sent across destinations is not guaranteed. Remember, JMS does not explicitly support distributed transactions.

As a result of the above-mentioned factors, the message delivery order for the same set of messages even with identical delivery modes and priorities can be provider dependent.

JMS Message Duplication

The JMS specification mandates that a JMS provider must never deliver a second copy of an acknowledged message. When a client uses a session with the `AUTO_ACKNOWLEDGE` mode it is not in direct control of message acknowledgment. Since such clients cannot know for certain if a particular message has been acknowledged, they must be prepared for re-delivery of the last consumed message. For example, this can occur if the client completed its work just prior to a failure that prevents the session from acknowledging the message. Only a session's last consumed message is subject to this ambiguity.

The JMS specification also mandates that providers must never produce duplicate messages. This means that a client that produces a message can rely on its JMS provider to insure the consumers of the message will only receive it once. No client error can cause a provider to duplicate a message. For example, consider the case when 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. In such cases where ambiguity may exist, it is up to the client to deal with this ambiguity, such as by resending the message(s). Note that these are not considered duplicate messages even though they may be functionally the same. Also, a message that is redelivered due to session recovery is not considered a duplicate message.

JMS Multi-Threading

Multi-threading is deeply embedded within the Java platform. Java provides a simple, elegant, and extremely powerful model for creating multi-threaded programs. As with any other powerful feature, multi-threading must be used with care. JMS could have required that all its objects support concurrent use. Designing an object to support concurrent use adds complexity to the design and overhead during runtime. This in itself is not bad, but not all objects are supposed to be used by multiple threads and in those cases this complexity and overhead are an unnecessary burden.

JMS has classified objects into two categories: those that would naturally be shared by a multi-threaded client and those that are to be accessed by one logical thread of control at a time. Table 1 summarizes the categories and the JMS objects belonging to these categories:

JMS objects that support concurrent use do not need any more discussion. Basically, a client can use them from any thread without the need for any synchronization.

JMS Objects that support concurrent use	Destination, ConnectionFactory, Connection
JMS Objects that do NOT support concurrent use	Session, MessageProducer, MessageConsumer

Table 1: JMS Objects and their Concurrency support

So, having said that, let's take a detailed look at the objects that do not support concurrent use.

In the previous chapter, I discussed the session object at length. One detail that I glossed over is that *it is a single threaded context* for producing and consuming messages. This means that 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. The `close` method on a session can be legally called by another thread than the thread that owns the session.

Why does the JMS enforce this?

There are two reasons for restricting concurrent access to sessions:

1. Transaction Support.

As discussed previously, sessions are the JMS entity that supports transactions. As a general rule, it is very difficult to implement transactions that are multi-threaded.

2. The 80/20 rule.

As discussed previously, sessions support asynchronous message consumption. If the session object supported concurrent access, clients would have to code their asynchronous message handlers to be capable of handling multiple concurrent messages. In addition, if the session had been set up with multiple asynchronous consumers, the client would have to explicitly code to handle the situation where these separate consumers are executing concurrently. Thus had the session been designed to allow multi-threaded access, clients using asynchronous delivery would have had to explicitly deal with three major situations:

- a. A single message listener is processing multiple messages at the same time.
- b. Multiple message listeners are processing the same message at the same time.
- c. Multiple message listeners are processing different messages at the same time.

This would be detrimental in at least two ways:

- a. It would place a burden on most clients who do not have the need for concurrency
- b. It would raise the level of development experience required for the creator of a JMS client significantly, because dealing with bugs introduced by ill-designed multi-threaded programs is not for the "faint of the heart".

For these reasons, a session serializes all asynchronous delivery of messages. 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. As discussed earlier in this chapter, more sophisticated clients can get the concurrency they desire by using multiple sessions.

The Rules for Using Sessions, Message Consumers, and Message Producers

Here are some rules to remember when using sessions. Note that not all JMS providers enforce all the threading constraints suggested by the JMS specification. However, for maximum portability, I recommend that you follow these rules whole-heartedly.

1. Only one thread may call the `receive` method at a time in the same session. The JMS specification states that it is erroneous for a client to use a thread of control to attempt to synchronously receive a message if there is already another client thread of control waiting to receive a message in the same session. This is true even for two threads of the same client.

Consider the following code fragment:

```
// assume that session is a TopicSession
// and topic is a Topic.
MyThread t1 = new MyThread(session, topic).start();
MyThread t2 = new MyThread(session, topic).start();

class MyThread extends Thread {
    public MyThread(?) {
```

```

        // save params to member variables?
    }

    public void run() {
        TopicSubscriber subscriber = null;
        subscriber = session.createSubscriber(topic);
        Message msg = subscriber.receive();
    }
}

```

In the above code fragment, if both threads t1 and t2 execute the `receive` method concurrently (which most likely will happen), it would be a violation of the JMS specification. The correct way to do this would be to have two separate sessions such as:

```

// assume that session1 and session2 are TopicSessions
// and topic is a Topic.
MyThread t1 = new MyThread(session1, topic).start();
MyThread t2 = new MyThread(session2, topic).start();

class MyThread extends Thread {
    public MyThread(?) {
        // save params to member variables?
    }

    public void run() {
        TopicSubscriber subscriber = null;
        subscriber = session.createSubscriber(topic);
        Message msg = subscriber.receive();
    }
}

```

2. 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. The JMS specification states that it is erroneous for client code to use such a session from another thread of control. The only exception to this rule is that the `close` method of the session or connection may be called concurrently from another thread.

Since a session with a registered message listener is dedicated to the thread of control that delivers messages to it, a session with message listeners cannot also be used to synchronously receive messages. That is, 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. The JMS specification states that it is erroneous to attempt to combine both in the same session. So, the following is not allowed by JMS:

```

// assume that session is a QueueSession
// and queue is a Queue.
QueueReceiver receiver = null;
receiver = session.createReceiver(queue);

receiver.setMessageListener(new MyMessageHandler());

// This is not allowed since we already installed
// a message listener?
Message msg = receiver.receive();

```

3. If a client desires to have one thread producing messages while another thread asynchronously consumes messages *at the same time*, the client should use a separate session for its producing thread. This is especially important with *transacted* sessions.

Remember, whenever a session contains one or more asynchronous message listeners, the JMS provider implementation automatically creates a single thread that is used to deliver all messages to ALL message listeners. In this case, the client should avoid

creating producer objects within that session. So now we have one thread that's executing the message listeners and another thread i.e. the client thread, that's sending/publishing messages. This may result in two threads operating on the same session at the same time, even though the session is not thread-safe.

Consider the following code fragment:

```
// Assume that tc is TopicConnection?
// Use tc to create a transacted topic session
TopicSession ts = tc.createTopicSession(true,
                                         Session.AUTO_ACKNOWLEDGE);

// create two TopicSubscribers for the topic testTopic
TopicSubscriber tsub1 = ts.createSubscriber(testTopic);
TopicSubscriber tsub2 = ts.createSubscriber(testTopic);

// Install message listeners on each TopicSubscriber
// The JMS provider will execute all these listeners on the //
// same, special thread. One such thread per session.
tsub1.setMessageListener (new MyMessageHandler1());
tsub2.setMessageListener (new MyMessageHandler2());

// create a TopicPublisher for the topic testTopic
TopicPublisher tp = ts.createPublisher(topic);

// bad! This is executed on the client thread.
tp.publish(?);
```

Now consider the following scenario:

The client thread is publishing a message at the same time that one of the message listeners has just finished executing a commit on the session. At this point all the messages published by tp are also automatically committed (i.e. all messages produced by all publishers will be sent to the JMS Server, regardless of whether the application-code executes a commit operation or not). Thus, operations performed by the message listener threads can affect the operations in the client thread and vice-versa. For this reason, it is strongly recommended that clients use separate sessions to send and receive messages, if the message receipt is done asynchronously.

However, the following is legal:

```
// Assume that tc is TopicConnection?
// Use tc to create a transacted topic session
TopicSession ts = tc.createTopicSession(true,
                                         Session.AUTO_ACKNOWLEDGE);

// create two TopicSubscribers for the topic testTopic
TopicSubscriber tsub1 = ts.createSubscriber(testTopic);
TopicSubscriber tsub2 = ts.createSubscriber(testTopic);

// create a TopicPublisher for the topic testTopic
TopicPublisher tp = ts.createPublisher(topic);

// Install message listeners on each TopicSubscriber
tsub1.setMessageListener (new MyMessageHandler1(tp));
tsub2.setMessageListener (new MyMessageHandler2());

.
.
.

class MyMessageHandler1 implements javax.jms.MessageHandler {
```

```

public MyMessageHandler1() {
    // save tp to a member variable.
}

public void onMessage(javax.jms.Message msg) {

    // catch all exceptions
    try {
        // Do something with the message
        tp.publish(?);
    }
    catch(java.lang.Exception e) {
    }
}
}

```

So, why is this legal? I just said that you should create separate sessions for sending/publishing and asynchronous receiving. Now I'm giving an example that doesn't require this. No, I'm not nuts. In this example, the publisher tp is used in the same thread that is invoking the message listeners, so we are NOT violating the single thread access requirement for using sessions.

4. To setup a session with more than one message listener, the connection that the session belongs to must be in the stopped mode. This is because if the connection is not in the stopped mode, it may deliver a message to the listener registered with the listener. At this point the session is controlled by the thread that delivered the message to it and hence cannot be configured by the client thread anymore. In general, as discussed in chapter 3, it is always good coding practice to finish all setup before starting the connection.

JMS and Security

With the advent of distributed computing, security has become an even more important topic of discussion. A good security policy should allow the setting up of at least the following (let's call these requirements):

1. **Authentication Policies**
Identifies each user of the JMS provider.
2. **Authorization Policies**
Identifies every operation that a user can/cannot perform on the JMS provider. For example, a user "Mallory" may not be allowed to send messages over the queue called "Executive".
3. **Message Integrity**
Allows setting up policies that can be used to detect whether a message has been altered during transit.
4. **Message Privacy**
Allows setting up privacy levels that determine how the message travels from the source to the destination i.e. is the message sent as plaintext or it is encrypted? If it is encrypted, how strong is the encryption, etc?

In addition, any security policy enforcement should be separable from the actual business code that a developer must write. That is, setting up security should be a separate task from actually creating the business application (unless of course your business is security *). This allows the business/domain expert to create the application, allowing the system administrator/deployment expert to concentrate on setting up the security policies. Although not related to the above four requirements, I would add this as requirement #5. This is a feature found in all successful application servers today.

As I mentioned in chapter 1, JMS does not define a security model for secured messages. Instead it is upto each individual JMS provider to implement their own security features. This means that a JMS provider may not provide any security features at all and still be JMS compliant. It also means that most commercial (and open source) implementations will provide at least some security features as a point of competitive differentiation.

JMS does provide the hooks for implementing a simple username and password based security model as seen in the `QueueConnectionFactory` and `TopicConnectionFactory` interfaces below:

```
public interface QueueConnectionFactory extends ConnectionFactory {
    QueueConnection createQueueConnection()
        throws JMSEException;
    QueueConnection createQueueConnection(String userName,
        String password) throws JMSEException;
}
```

and,

```
public interface TopicConnectionFactory extends ConnectionFactory {
    TopicConnection createTopicConnection()
        throws JMSEException;
    TopicConnection createTopicConnection(String userName,
        String password) throws JMSEException;
}
```

Both these interfaces derive from the generic `ConnectionFactory` interface that I discussed in chapter 3. As its name implies, the `QueueConnectionFactory` is used for creating queue connections. The `createQueueConnection` method instructs the JMS provider to create a queue connection with default user identity. The exact meaning of the default user identity is provider specific. The other version of the `createQueueConnection` method takes a username and password and instructs the JMS provider to create a queue connection with these user credentials. Similarly, the `TopicConnectionFactory` is used to create topic connections. The two versions of `createTopicConnection` are very similar to the two versions of `createQueueConnection`, except that the former pair returns a topic connection instead of a queue connection. These methods may throw a `JMSSecurityException` if client authentication fails due to an invalid username or password. How authenticity is determined is provider specific.

Thus JMS provides the facilities to satisfy requirements #1 (directly) and #2 (indirectly) above. Requirements #3 and #4 are still entirely provider specific.

Let's take a step-by-step look at how a typical [hypothetical] JMS provider could enforce security policies setup by an administrator.

1. Let's assume user "Alice" wants to send a message to user "Bob" via the queue called "privateQ". The administrator has setup Alice and Bob as users and a queue called "privateQ" using the administration features of our hypothetical JMS provider.
2. Anne opens a new connection to the JMS provider as follows.

```
QueueConnection connection =
    qcf.createQueueConnection("Anne", "Anne's Password");
```

Here `qcf` is a queue connection factory (don't worry how we got it for now), "Anne" is the username and "Anne's Password" is her password. In our hypothetical JMS provider, the client side JMS runtime sends the username and a one-way hash (using the MD5

algorithm, for example) to the JMS server along with the connection request. The JMS server maintains a [secure] database of usernames and the hashed passwords, which it uses to verify Anne's identity. This is known as "Authentication." Since Anne is a "valid" user, she gets to establish the queue connection.

3. Now Anne wants to send a message over the queue called "privateQ," which she does as follows:

```
QueueSession session = connection.createQueueSession(false,  
                                                    Session.AUTO_ACKNOWLEDGE);  
Queue queue = session.createQueue("privateQ");
```

In our hypothetical JMS provider, a session object inherits the security credentials of its creator ("Anne") from the connection. So when Anne tries to create the queue "privateQ", the JMS runtime sends the username, the password hash and the request to create the queue. Let's assume that Anne has been granted the permission to create the queue and so this attempt succeeds. This is known as "Authorization".

4. Now Alice attempts to send a message to Bob via the queue as follows.

```
QueueSender sender = session.createSender(queue);  
Sender.send(message);
```

Two checks are performed at this stage. The first check occurs when Alice attempts to create the queue sender and the second check occurs when Alice uses this sender to actually send a message. Either one of these could fail. For example, Alice might have been granted permission to create a sender for the queue "privateQ" but not to send messages via the queue (not very logical, but possible).

5. The message may travel long distances before actually reaching Bob. During this time a user "Eve" may read the message. This is where privacy comes into play. During the creation of a session, the JMS client runtime and the JMS server may agree upon a session key. Every message sent from this client to the JMS server will be encrypted using this session key using an algorithm such as DES. This makes Eve do a lot more work if she wants to read our messages. While Eve simply wanted to read the message contents, a more malicious user "Mallory" may actually corrupt the message. If the message is not confidential and we simply want to detect Mallory's actions the JMS client runtime can append the message with an encrypted one-way hash of the message (i.e. digital signature). The JMS server would then also calculate the hash of the message and compare it to the decrypted hash in the message to verify the message integrity.
6. On the other side Bob goes through the same steps as Alice, except that in step 4, Bob creates a queue receiver and calls receive on it. In this case, our hypothetical JMS provider verifies that Bob is authorized to create a queue receiver for the queue "privateQ" and receive messages on it.

In the above example, our hypothetical JMS provider has built its entire security model on top of JMS's simple security support. Plus, the JMS developer does not have to do any more work to enable security. This model is only an example and I can think of many improvements. For example, using digital certificates and a scheme similar to SSL not only can the JMS server authenticate the client, but the client can authenticate the server as well. This is known as "mutual authentication." Most Commercial JMS providers do provide a much more robust security model and many more security features. The main point to remember is that JMS does not define how a JMS provider provides security or even if any security is provided at all. However, enterprise applications require security and so the security support provided by any JMS provider must be considered while selecting a provider. Hopefully, the discussion in this section should provide a good start for any such evaluation.

Summary

In this chapter, I discussed some of the more involved issues surrounding the basic concepts of JMS. I started off with a discussion about how JMS supports transactions. We then looked at the various message delivery styles and the order in which JMS messages are delivered. Finally, I discussed in detail the considerations for using JMS in a multi-threaded environment. In the next chapter, I will discuss the lifeblood of any messaging system – messages.

Chapter 5

The JMS Message Model

Introducing the JMS Messages

At the heart of any message oriented middleware (MOM) system lies, you guessed it, messages. In other words, messages are the lifeblood of such systems, for without these messages such systems would not be able to accomplish much. As with every other aspect of available enterprise level messaging products, there exists as many message formats as there exist products. JMS attempts to standardize the message model. The JMS messaging model is simple, elegant, and provides the flexibility to send any type of data across the enterprise.

As shown in figure 1, JMS defines five different types of messages that can be published by an application, all of which derive from a common base, *Message*.

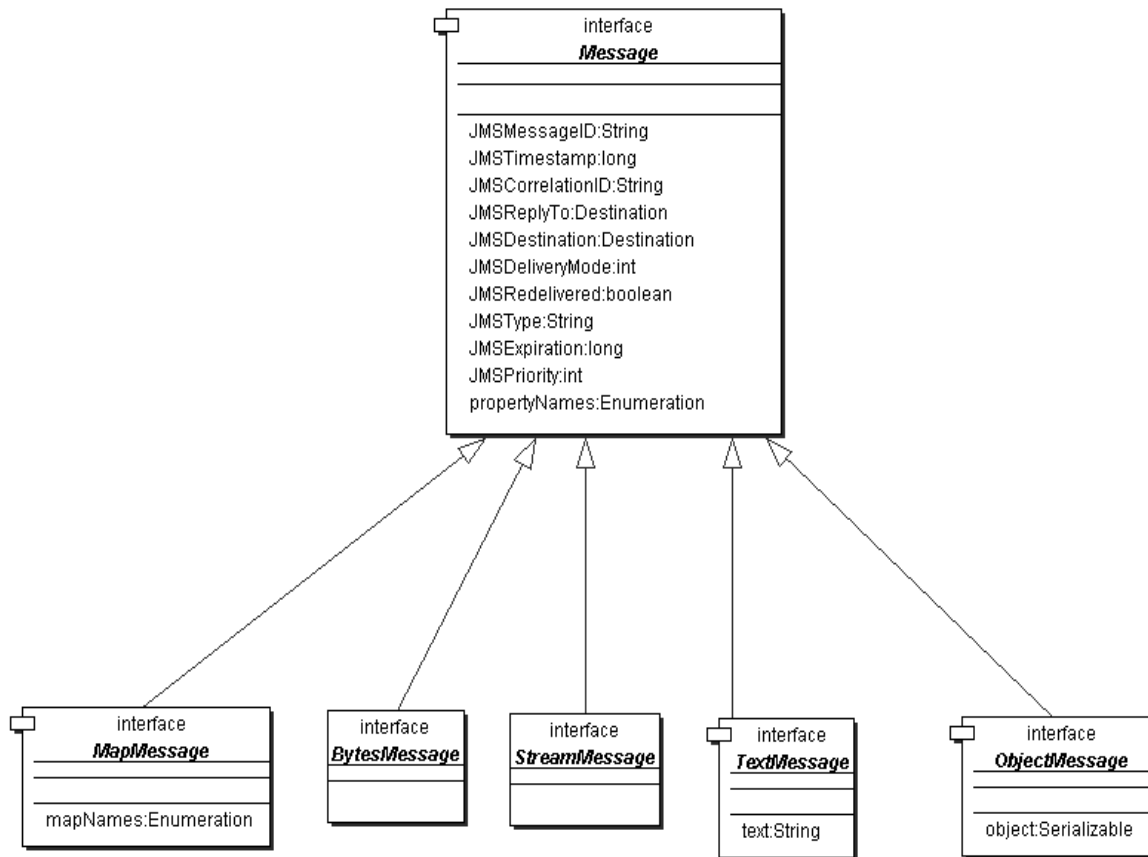


Figure 1: The JMS Message Model

As with every other aspect of JMS, JMS only provides a set of message interfaces that define the JMS message model. Each JMS provider provides the implementation of these interfaces. 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.

I mentioned in chapter 3 that the session object acts as a factory for creating messages. The relevant portion of the `Session` interface is shown below:

```
public interface Session extends Runnable {
    // Only Message Creation methods shown
    // for clarity
    .
    .
    .
    Message createMessage() throws JMSEException;
    BytesMessage createBytesMessage() throws JMSEException;
    MapMessage createMapMessage() throws JMSEException;
    ObjectMessage createObjectMessage() throws JMSEException;
    ObjectMessage createObjectMessage(Serializable object)
        throws JMSEException;
    StreamMessage createStreamMessage() throws JMSEException;
    TextMessage createTextMessage() throws JMSEException;
    TextMessage createTextMessage(String text) throws JMSEException;
    .
    .
}
```

Most JMS providers will make the constructor of each of its message implementations private or at least package protected, so that clients cannot create messages by calling `new` on the class. A valid message *should* be creatable *only* by using the appropriate factory method on the session object.

All JMS messages are composed of the following three parts:

1. *The Message 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.

2. *The Message Properties*

In addition to the standard header fields, messages provide a built-in facility for adding optional header fields to a message. These properties may also be used by message consumers for filtering out messages that they are not interested in, or rather specifying exactly which messages they are interested in. This is known as "Message Selection" in JMS parlance and will be covered in detail later in this chapter. There are three types of supported properties:

a. Application-specific properties

This provides a mechanism for adding application specific header fields to a message. Applications can define their own properties unique to their domain.

b. Standard properties

JMS defines some standard properties that are, in effect, optional header fields. JMS defines a naming convention for such properties.

c. 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 such properties.

3. *The Message Body*

JMS defines several types of message body which cover the majority of messaging styles currently in use. The message body contains the actual data that is to be transmitted via the message. Think of the message body as the actual contents of an envelope. The header and properties are analogous to the "to" and "from" addresses and any other information required to get the envelope to its destination. The JMS provider is analogous to the postal service.

The JMS message interfaces provide write/set methods for setting object values in a message body and message properties. The JMS specification states that JMS providers must implement all of these methods so that they copy their input objects into the message [as opposed to keeping a reference to the actual object itself]. The value of an input object can be `null` and will return `null` when accessed. The only exception to this is that `BytesMessage` does not support the concept of a `null` stream and attempting to write a `null` into it will throw `java.lang.NullPointerException`. The JMS message interfaces also provides read/get methods for accessing objects in a message body and message properties. Once again, the JMS specification states that all of these methods must return a copy of the accessed message objects rather than a reference to the actual message object itself.

When a client receives a message it is in read-only mode. If a client attempts to write to the message at this point, a `MessageNotWriteableException` is thrown. If `clearBody` is called on the message, the message can now be both read from and written to. Calling the `clearBody` method only clears out the message body, not the header and properties.

The Message Interface

As shown in figure 1, all JMS message types are derived from the `Message` interface, which is defined as follows:

```
public interface Message {

    static final int DEFAULT_DELIVERY_MODE = DeliveryMode.PERSISTENT;
    static final int DEFAULT_PRIORITY = 4;
    static final long DEFAULT_TIME_TO_LIVE = 0;

    ////////////////////////////////////////////////////////////////////
    // Methods to manipulate the Message Header
    ////////////////////////////////////////////////////////////////////

    // JMSMessageID
    String getJMSMessageID() throws JMSEException;
    void setJMSMessageID(String id) throws JMSEException;

    // JMSTimestamp
    long getJMSTimestamp() throws JMSEException;
    void setJMSTimestamp(long timestamp) throws JMSEException;

    // JMSCorrelationID
    byte [] getJMSCorrelationIDAsBytes() throws JMSEException;
    void setJMSCorrelationIDAsBytes(byte[] correlationID) throws
    JMSEException;
    void setJMSCorrelationID(String correlationID) throws JMSEException;
    String getJMSCorrelationID() throws JMSEException;

    // JMSReplyTo
    Destination getJMSReplyTo() throws JMSEException;
    void setJMSReplyTo(Destination replyTo) throws JMSEException;

    // JMSDestination
    Destination getJMSDestination() throws JMSEException;
    void setJMSDestination(Destination destination) throws JMSEException;
}
```



```

// JMSDeliveryMode
int getJMSDeliveryMode() throws JMSEException;
void setJMSDeliveryMode(int deliveryMode) throws JMSEException;

// JMSRedeilvered
boolean getJMSRedelivered() throws JMSEException;
void setJMSRedelivered(boolean redelivered) throws JMSEException;

// JMSType
String getJMSType() throws JMSEException;
void setJMSType(String type) throws JMSEException;

// JMSEExpiration
long getJMSEExpiration() throws JMSEException;
void setJMSEExpiration(long expiration) throws JMSEException;

// JMSPriority
int getJMSPriority() throws JMSEException;
void setJMSPriority(int priority) throws JMSEException;

////////////////////////////////////
// Methods to get/set Message Properties
////////////////////////////////////

// Get a property
boolean getBooleanProperty(String name) throws JMSEException;
byte getByteProperty(String name) throws JMSEException;
short getShortProperty(String name) throws JMSEException;
int getIntProperty(String name) throws JMSEException;
long getLongProperty(String name) throws JMSEException;
float getFloatProperty(String name) throws JMSEException;
double getDoubleProperty(String name) throws JMSEException;
String getStringProperty(String name) throws JMSEException;
Object getObjectProperty(String name) throws JMSEException;

// Set a property
void setBooleanProperty(String name, boolean value)
                                throws JMSEException;
void setByteProperty(String name, byte value) throws JMSEException;
void setShortProperty(String name, short value) throws JMSEException;
void setIntProperty(String name, int value) throws JMSEException;
void setLongProperty(String name, long value) throws JMSEException;
void setFloatProperty(String name, float value) throws JMSEException;
void setDoubleProperty(String name, double value) throws JMSEException;
void setStringProperty(String name, String value) throws JMSEException;
void setObjectProperty(String name, Object value) throws JMSEException;

void clearProperties() throws JMSEException;
Enumeration getPropertyNames() throws JMSEException;
boolean propertyExists(String name) throws JMSEException;

////////////////////////////////////
// Miscellaneous
////////////////////////////////////

void acknowledge() throws JMSEException;
void clearBody() throws JMSEException;
}

```

The bulk of the `Message` interface provides methods to access the message header and properties.

The Message Header

As mentioned earlier, there is a core set of header fields supported by all messages. Header fields contain values used by both clients and providers to identify and route messages. Let's look at each header field one by one.

JMSMessageID

The `Message` interface provides a pair of methods, `getJMSMessageID` and `setJMSMessageID`, to get and set the Message ID in a message respectively. When a message is passed to the `send/publish` method of a message producer, the `JMSMessageID` header field in the message is ignored. When the `send/publish` method returns it contains a provider-assigned value, which the client can get and keep track of. A `JMSMessageID` is a `String` value which should function as a unique key for identifying messages. The exact scope of uniqueness is provider defined. 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 (and time) to create and increase a message's size, JMS specifies a way in which clients can provide a "hint" to the JMS provider that it does not use the message ID. A client gives this hint by calling the `setDisableMessageID` method on the message producer with a `true` parameter. Note that this is called a "hint", which means that any specific JMS implementation is free to make use of the hint or to ignore it. If the hint is set to `true`, messages must either be assigned a null message ID or, if the hint is ignored, the message ID must be set to its normal unique value.

JMSTimestamp

The `Message` interface provides a pair of methods, `getJMSTimestamp` and `setJMSTimestamp`, to get and set the Message ID in a message. 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 queuing of messages. When a message is passed to the `send/publish` method of a message producer, the `JMSTimestamp` header field in the message is ignored. When the `send/publish` method returns it contains a provider-assigned value, which the client can get and keep track of.

As with message IDs, timestamps take some effort (and time) to create and increase a message's size as well. So, JMS specifies a way in which clients can provide a "hint" to the JMS provider that it does not use the timestamp. A client gives this hint by calling the `setDisableMessageTimestamp` method on the message producer with a `true` parameter. Note that this is called a "hint", which means that any specific JMS implementation is free to make use of the hint or to ignore it. If the hint is set to `true`, messages must either be assigned a null timestamp or, if the hint is ignored, the timestamp must be set to its normal value.

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 or an application-specific string
The `Message` interface has a pair of methods `setJMSCorrelationID` and `getJMSCorrelationID` to support this. For example:

```
msg.setJMSCorrelationID("2000:07:20:10:56:02:Method1");
```

- A provider-native `byte[]` value.

The `Message` interface has a pair of methods `setJMSCorrelationIDAsBytes` and `getJMSCorrelationIDAsBytes` to support this.

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. The `Message` interface has a pair of methods, `getJMSReplyTo` and `setJMSReplyTo`, to set and get the `JMSReplyTo` header field. I will discuss an example usage of this property in the next chapter in the section "Request/Reply Operation".

JMSDestination

The `JMSDestination` header field contains the destination to which the message is being sent. When the message is passed to the `send/publish` method on the message producer this value is ignored. After completion of the `send/publish` it will hold the destination object specified by the sending method. The `Message` interface has a pair of methods, `setJMSDestination` and `getJMSDestination`, to set and get this header field.

JMSDeliveryMode

The `JMSDeliveryMode` header field contains the delivery mode to be used to deliver the message is being sent. When the message is passed to the `send/publish` method on the message producer this value is ignored. After completion of the `send/publish` it will hold the delivery mode object specified by the sending method.

JMS supports two modes of message delivery: persistent and non-persistent. Of these two modes, the non-persistent mode is the lower overhead delivery mode because it does not require that the message be logged to stable storage. In case of a JMS provider failure [or even without one] a non-persistent message may be lost. In fact, a provider that discards every non-persistent message would still be JMS compliant, although not very competitive in the market *. The non-persistent mode implies "best effort" semantics, which is open to interpretation by each JMS provider. However, the JMS specification mandates that a provider must deliver a non-persistent message at-most-once. *This means it may lose the message but must not deliver it more than once.*

On the other hand, the persistent mode forces the JMS compliant provider to take extra care to insure the message is not lost in transit due to a JMS provider failure. The JMS specification mandates that a provider must deliver a persistent message once-and-only-once. *This means that the provider must not allow the message to be lost and it must not deliver it more than once.*

The `Message` interface has a pair of methods, `setJMSDeliveryMode` and `getJMSDeliveryMode`, to set and get this header field.

JMSRedelivered

If a client receives a message with the `JMSRedelivered` indicator set, it is likely, but not guaranteed, that this message was delivered to the client earlier but the client did not acknowledge its receipt at that time. This header value is set by the JMS Provider. The `Message` interface has a pair of methods, `setJMSRedelivered` and `getJMSRedelivered`, to set and get the `JMSRedelivered` header field.

JMSType

The `JMSType` header field contains a message type identifier supplied by a client when a message is sent. Some JMS providers use a message repository that contains the definition of messages sent by applications. The type header field may reference a message's definition in the provider's repository. JMS does not define a standard message definition repository nor does it define a naming policy for the definitions it contains. The `Message` interface has a pair of methods, `setJMSType` and `getJMSType`, to set and get the `JMSType` header field. For example:

```
msg.setJMSType("AccountsPayable");
```

JMSExpiration

The `JMSExpiration` header field contains the message's expected time-to-live. When a message is passed to the `send/publish` method of a message producer, the `JMSExpiration` header field in the message is ignored. After completion of the `send/publish` this field will hold the value specified by the method sending the message.

To calculate the message's actual expiration time the value contained in the `JMSExpiration` header field is added to the time (GMT) the message is actually sent. Even for transacted sends, this is the time the message is sent and not the time at which the transaction is committed. The JMS specification states that a JMS provider should do its best to accurately expire messages, but does not define the accuracy provided. At the same time, the specification makes it clear that it is not acceptable for a JMS provider to simply ignore time-to-live.

The `Message` interface has a pair of methods, `setJMSExpiration` and `getJMSExpiration`, to set and get this header field.

JMSPriority

The `JMSPriority` header field contains the message's priority. When a message is passed to the `send/publish` method of a message producer, the `JMSPriority` header field in the message is ignored. After completion of the `send/publish` this field will hold 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. 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. So, a provider that simply ignored priorities would be considered JMS compliant but not very competitive in the market.

The `Message` interface has a pair of methods, `setJMSExpiration` and `getJMSExpiration`, to set and get this header field.

JMS Note:

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

Table 1 summarizes the various header properties and who sets them

Header Field	Who sets it?
JMSDestination	The Send/Publish method
JMSDeliveryMode	The Send/Publish method
JMSExpiration	The Send/Publish method

JMSPriority	The Send/Publish method
JMSMessageID	The Send/Publish method
JMSTimestamp	The Send/Publish method
JMSCorrelationID	The Client
JMSReplyTo	The Client
JMSType	The Client
JMSRedelivered	The JMS Provider

Table 1

The Message Properties

In addition to the header fields defined above, JMS messages contain a built-in facility for supporting property values. In effect, this provides a mechanism for adding optional header fields to a message. These properties can be used to filter messages. I will discuss this later in the chapter in the section "Message Selection". Properties are specified as name/value pairs. The name of the property is a String with some restrictions, which will be discussed when I discuss message selection. The value of the property can be a `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, or `String`.

Property values are set prior to sending a message. Remember, 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.

JMS Note:

For best performance, JMS recommends that applications should only use message properties when they need to customize a message's header. This "customized" header can then be used in the JMS message selection process. Message selection is covered later in this chapter. The reason that the specification makes this recommendation is because the engineers felt that most JMS providers are likely to handle the message body much more efficiently than they handle the message properties. So per this recommendation, if a piece of data is not going to be used to filter data using the JMS message selection process then that data is better off being part of the message body. Note that even though some JMS providers may handle message properties equally or even more efficiently than they handle the message body, this is still a good recommendation to follow as it leads to a cleaner message structure by forcing the designer to think about each piece of data in the message.

Application Specific Properties

This provides a mechanism for adding application specific header fields to a message. Applications can define their own properties unique to their domain. The names of these properties must follow a set of rules that are discussed later in this chapter as part of message selection. Primitive property types can be read or written explicitly using methods for each type. They may also be read or written generically as objects. For example the following two lines of code are equally effective in setting an integer property "TrasactionNumber" to the value "6" .

```
// Assuming msg is a valid "writable" Message
msg.setIntProperty("TransactionNumber", 6);
msg.setObjectProperty("TransactionNumber", 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. Note that the `get/setObjectProperty` methods only works for the "objectified" primitive object types (`Integer`, `Double`, `Long` ...), `Strings` and `byte arrays`.

Standard Properties

JMS reserves the 'JMSX' property name prefix for JMS defined properties. The full set of these properties is provided in table 2 . New JMS defined properties may be added in later versions of JMS. `JMSXGroupID` and `JMSXGroupSeq` are standard properties that clients should use if they want to group messages. All providers must support them. Support for all other JMSX properties is optional. The `getJMSXPropertyNames` method of the `ConnectionMetaData` interface returns the names of the JMSX properties supported by a connection.

JMSX Property	Type	Who Sets it?	Typical Use
JMSXUserID	String	Provider on Send/Publish	The identity of the user sending the Message
JMSXAppID	String	Provider on Send/Publish	The identity of the application sending the message
JMSXDeliveryCount	int	Provider on Receive	The number of message delivery attempts; the first is 1, the second 2,...
JMSXGroupID	String	Client	The identity of the message group this message is part of
JMSXGroupSeq	int	Client	The sequence number of this message within the group; the first message is 1, the second 2,...
JMSXProducerTXID	String	Provider on Send/Publish	The transaction identifier of the transaction within which this message was produced
JMSXConsumerTXID	String	Provider on Receive	The transaction identifier of the transaction within which this message was consumed
JMSXRcvTimestamp	long	Provider on Receive	The time JMS delivered the message to the consumer
JMSXState	int	Provider	Provider specific data for a message

Table 2: JMS Standard Properties.

JMS Note:

Except the `JMSXGroupID` and `JMSXGroupSeq` properties, the values and semantics of all JMSX properties are undefined.

Most JMS clients will never need to use the standard properties with the exception of the `JMSXGroupID` and `JMSXGroupSeq` properties. But why use these standard properties for message grouping when JMS already allows clients to define their own application specific properties? For example, a company called Online Insight could do message grouping by defining properties called "OiGroupID" and "OiGroupSeq". One reason to use the standard properties is that, since these are standard properties, all applications which touch the message can interpret the meaning of these properties. Therefore, we can add third party applications that know nothing of the rest of our application, but who listen to a topic and do something useful such as log messages – and these third party applications will be able to know which messages are grouped together by reading these standard properties. Likewise, JMS vendors can release administrative type tools that can make use of the semantics of these standard properties.

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.

JMS Note:

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 i.e. clients should not rely upon these properties at all.

Calling `clearProperties` on the `Message` will clear all properties. A client can query the existence of a particular property by calling the `propertyExists` method on the `Message` as follows:

```
// Are we in a custom transaction?
if(msg.propertyExists("TransactionNumber")) {
    // Get the transaction number and continue?
}
// Not in a custom transaction?
else {
    // No transaction?
}
```

The client can also walk through every property, for example:

```
// Walk through all the properties?
java.util.Enumeration enum = msg.getPropertyNames();
while(enum.hasMoreElements()) {
    String name = enum.nextElement();
    // get the property value and do something useful?
}
```

Note that the order of the properties is not defined. This means that the order of property names in the enumeration returned by the `getPropertyNames` method may not be in any deterministic order, or even the order in which they were initially set on the message.

Attempting to get a property value for a name which has not been set is handled as if the the property exists with a `null` value.

JMS-mandated Type Conversions for Properties

Consider the following code fragment:

```
// Assuming msg is a valid "writable" Message
msg.setIntProperty("TransactionNumber", 6);
msg.setObjectProperty("TransactionNumber", new Integer(6));
.
.
String txNumber = msg.getStringProperty("TransactionNumber");
System.out.println("Transaction Number is " + txNumber);
```

Even though the "TransactionNumber" property was added as type `int`, this code will work without a hitch. The `int` will be converted into a type `String` by the message implementation.

Table 3 summarizes all the legal conversions which must be supported by all JMS providers.

	boolean	byte	short	char	int	long	float	double	String
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

Table 3: The legal conversions allowed between Message Properties. A value written as the row type can be read as the column type.

The "X" marked cases must be supported by a JMS provider. The unmarked cases must throw a `MessageFormatException`. The `String` to primitive conversions may throw a runtime exception if the primitive's `valueOf()` method of the corresponding primitive's class (such as `Integer` for `int`) does not accept it as a valid `String` representation of the primitive.

The Message Body

So far we've looked at the message header and properties. Now let's take a look at the actual payload of the message itself – the message body.

As mentioned before, JMS defines five types of messages. Each of these derives from the `Message` interface, so each message type provides the same support for header and property information. Based on the type of data to be sent in the message, a client can use the type of message that best suits its needs.

Let's look at each message type.

Text Message

A text message object implements the `TextMessage` interface defined by JMS as follows:

```
public interface TextMessage extends Message {
    void setText(String string) throws JMSEException;
    String getText() throws JMSEException;
}
```

A `TextMessage` is used to send a message containing a `java.lang.String`. The primary reason this message type was included was that the authors of the JMS specification anticipated that XML would become a popular mechanism for representing content; an anticipation that has come true.

To create a `TextMessage` message object, a client calls one of the two versions of `createTextMessage` on the session object. For example,

```
// Assume session is a valid Session object.
TextMessage txtMsg = session.createTextMessage();
```

To set the actual message in the text message object, a client calls the `setText` method, such as:

```
txtMsg.setText("I am a Text Message!");
```

Alternatively, these two steps can be combined as:

```
// Assume session is a valid Session object.
TextMessage txtMsg = session.createTextMessage(
    "I am a Text Message!");
```

To access the value of the data inside a `TextMessage`, a client calls the `getText` method

Object Message

An object message implements the `ObjectMessage` interface defined by JMS as follows:

```
public interface ObjectMessage extends Message {
    void setObject(Serializable object) throws JMSEException;
    Serializable getObject() throws JMSEException;
}
```

An `ObjectMessage` is used to send a message that contains a serializable Java object. To create a `ObjectMessage` message object, a client calls one of the two versions of `createObjectMessage` on the session object. An example is shown below:

```
class MyObject implements java.io.Serializable {
    String name;
    public MyObject(n) {
        name = n;
    }
    public void changeName(String newName) {
        name = newName;
    }
}

:
:
// Assume session is a valid Session object.
ObjectMessage objMessage = session.createObjectMessage();
objMessage.setObject(new MyObject("John"));

// Or use the overloaded version of createObjectMessage
// ObjectMessage objMessage = session.createObjectMessage(
//     new MyObject("John"));
:
:
```

Consider the following code fragment:

```
// Assume session is a valid Session object.
ObjectMessage objMessage = session.createObjectMessage();
MyObject obj = new MyObject("John");
objMessage.setObject(obj);
obj.changeName("Jim");
```

What name does the object in the message contain now, "John" or "Jim"? The message still has an object with the name "John" because the `ObjectMessage` implementation must make a copy of an object passed to it. Remember, the JMS specification mandates this.

To send a collection of such objects, one of the collection classes provided in JDK 1.2 may be used. For example:

```
// create a list of "MyObject" objects
java.util.ArrayList list = new java.util.ArrayList();
list.add(new MyObject("John"));
list.add(new MyObject("Bill"));
list.add(new MyObject("Phil"));
list.add(new MyObject("Bob"));

// Assume session is a valid Session object.
ObjectMessage objMessage = session.createObjectMessage();
// Send the list as the message payload.
objMessage.setObject(list);
```

Clients receiving an `ObjectMessage` should use the `getObject` method to access the object contained in the message. For example:

```

.
.
// Assume that receiver is a valid MessageConsumer object
ObjectMessage objMessage = (ObjectMessage)receiver.receive();
Serializable obj = objMessage.getObject();
if(obj instanceof MyObject) {
    // single instance?
}
else if(obj instanceof ArrayList) {
    // A list of MyObjects?
}
.
.

```

Stream Message

A stream message contains a stream of Java primitive values. A stream message implements the StreamMessage interface defined by JMS as follows:

```

public interface StreamMessage extends Message {

    // Read methods
    boolean readBoolean() throws JMSEException;
    byte readByte() throws JMSEException;
    short readShort() throws JMSEException;
    char readChar() throws JMSEException;
    int readInt() throws JMSEException;
    long readLong() throws JMSEException;
    float readFloat() throws JMSEException;
    double readDouble() throws JMSEException;
    String readString() throws JMSEException;
    int readBytes(byte[] value) throws JMSEException;
    Object readObject() throws JMSEException;

    // Write methods
    void writeBoolean(boolean value) throws JMSEException;
    void writeByte(byte value) throws JMSEException;
    void writeShort(short value) throws JMSEException;
    void writeChar(char value) throws JMSEException;
    void writeInt(int value) throws JMSEException;
    void writeLong(long value) throws JMSEException;
    void writeFloat(float value) throws JMSEException;
    void writeDouble(double value) throws JMSEException;
    void writeString(String value) throws JMSEException;
    void writeBytes(byte[] value) throws JMSEException;
    void writeBytes(byte[] value, int offset, int length)
        throws JMSEException;
    void writeObject(Object value) throws JMSEException;

    void reset() throws JMSEException;
}

```

Although bulky, this is an extremely easy interface. The interface can be summarized as

```

// pseudo-code interface
public interface StreamMessage extends Message {

    // Read data of type "XXX"
    XXX readXXX() throws JMSEException;
}

```

```

    // Write data of type "XXX"
    void writeXXX(XXX data) throws JMSEException;

    void reset() throws JMSEException;
}

```

The primitive types can be read or written explicitly using methods for each type. They may also be read or written generically as objects. For example the following two lines of code are equally effective in sending the integer value "6" in the message:

```

// Assuming streamMsg is a valid "writable" Stream Message
streamMsg.writeInt(6);
streamMsg.writeObject(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. Note that the `read/writeObject` methods only works for the "objectified" primitive object types (`Integer`, `Double`, `Long` ...), `Strings` and byte arrays.

To create a `StreamMessage`, a client uses the `createStreamMessage` method on the session as follows:

```

// Assume session is a valid Session object.
StreamMessage streamMsg = session.createStreamMessage ();

```

Using the `reset` and `clearBody` methods

When a stream message is first created the body of the message is in write-only mode. After a call to the `reset` method has been made, the message body is in read-only mode. When a client sends a stream message, the provider calls `reset` in order to read it's content, and actually send the message across. When the stream message is received on the other side, the provider reconstructs the message and calls `reset` on it so that the message body is in read-only mode for the client. Now if the client calls `clearBody` on this message, the message body is cleared and the message body is in write-only mode. This is different than the other two messages I've discussed so far in which calling `clearBody` makes the message both "readable" and "writable". If a client attempts to read a message in write-only mode, a `MessageNotReadableException` is thrown. If a client attempts to write a message in read-only mode, a `MessageNotWritableException` is thrown.

Let's go through an example.

```

class MyRectangle {
    int top;
    int bottom;
    int width;
    int height;

    public MyRectangle(int t, int b, int w, int h) {
        top = t;
        bottom = b;
        width = w;
        height = h;
    }
}
.
.
.

MyRectangle rectangle = new MyRectangle(10,20,50,30);

// Assume session is a valid Session object.
StreamMessage streamMsg = session.createStreamMessage ();

```

```

streamMsg.writeInt(rectangle.top);
streamMsg.writeInt(rectangle.bottom);
streamMsg.writeInt(rectangle.width);
streamMsg.writeInt(rectangle.height);

// This will throw a MessageNotReadableException
int x = streamMsg.readInt();

// Now it won't?
streamMsg.reset();
int y = streamMsg.readInt();

// But now I can't write anymore?
// This will throw a MessageNotWriteableException
streamMsg.writeInt(y);

```

Stream messages are filled and read sequentially. This means that if I write a `String` followed by an `int`, then I must read a `String` followed by an `int` in the same sequence. However, the `int` can be read out as an `int`, a `long`, or a `String`. This conversion is allowed by the `StreamMessage` implementation. Table 4 summarizes all the legal conversions that all JMS provider implementations of `StreamMessage` must support.

	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

Table 4: The legal conversions. A value written as the row type can be read as the column type.

The "X" marked cases must be supported by a JMS provider. The unmarked cases must throw a `JMSEException`. The `String` to primitive conversions may throw a runtime exception if the `primitives.valueOf()` method of the corresponding primitive's class (such as `Integer` for `int`) does not accept it as a valid `String` representation of the primitive.

Map Message

A map message is one whose body contains a set of name–value pairs where the names are `Strings` and the values are Java primitive types. The map message implements the `MapMessage` interface defined by JMS as follows:

```

public interface MapMessage extends Message {

    // Get values from the map.
    boolean getBoolean(String name) throws JMSEException;
    byte getByte(String name) throws JMSEException;
    short getShort(String name) throws JMSEException;
    char getChar(String name) throws JMSEException;
    int getInt(String name) throws JMSEException;
    long getLong(String name) throws JMSEException;
    float getFloat(String name) throws JMSEException;

```

```

double getDouble(String name) throws JMSEException;
String getString(String name) throws JMSEException;
byte[] getBytes(String name) throws JMSEException;
Object getObject(String name) throws JMSEException;

// Set values in the map.
void setBoolean(String name, boolean value) throws JMSEException;
void setByte(String name, byte value) throws JMSEException;
void setShort(String name, short value) throws JMSEException;
void setChar(String name, char value) throws JMSEException;
void setInt(String name, int value) throws JMSEException;
void setLong(String name, long value) throws JMSEException;
void setFloat(String name, float value) throws JMSEException;
void setDouble(String name, double value) throws JMSEException;
void setString(String name, String value) throws JMSEException;
void setBytes(String name, byte[] value) throws JMSEException;
void setBytes(String name, byte[] value, int offset,
              int length) throws JMSEException;
void setObject(String name, Object value) throws JMSEException;

// Miscellaneous.
Enumeration getMapNames() throws JMSEException;
boolean itemExists(String name) throws JMSEException;
}

```

This interface is very similar to the `StreamMessage` interface and can be summarized as

```

// pseudo-code interface
public interface MapMessage extends Message {

    // Get value of type "XXX" from the map
    XXX getXXX(String name) throws JMSEException;

    // Set data of type "XXX" in the map
    void setXXX(String name, XXX data) throws JMSEException;

    // Miscellaneous.
    Enumeration getMapNames() throws JMSEException;
    boolean itemExists(String name) throws JMSEException;
}

```

The primitive types can be get or set explicitly using methods for each type. They may also be read or written generically as objects. For example the following two lines of code are equally effective in sending the integer value "6" in the message:

```

// Assuming mapMsg is a valid "writable" Map Message
mapMsg.setInt("anInteger",6);
mapMsg.writeObject("anInteger",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. Note that the `get/setObject` methods only works for the "objectified" primitive object types (`Integer`, `Double`, `Long` ...), `Strings` and `byte` arrays.

To create a `MapMessage`, a client uses the `createMapMessage` method on the session as follows:

```

// Assume session is a valid Session object.
MapMessage mapMsg = session.createMapMessage();

```

The entries in a map message can be accessed sequentially by enumerator, which is obtained by calling the `getMapNames` method or randomly by name. For example:

```

// Random access
int x = mapMsg.getInt("anInteger");

// Walk through the map?
java.util.Enumeration enum = mapMsg.getMapNames();
while(enum.hasMoreElements()) {
    String name = enum.nextElement();
    // get the value and do something useful?
}

```

Note that the order of the entries is undefined. So, the order of the keys returned in the enumeration from the `getMapNames` method may not be the same as the order in which the values were added to the map. The `MapMessage` also supports the same conversions as supported by the `StreamMessage`, which are tabulated in table 4.

Bytes Message

A bytes message contains a stream of "uninterpreted" bytes. The bytes message implements the `BytesMessage` interface defined by JMS as follows:

```

public interface BytesMessage extends Message {

    // Read methods
    boolean readBoolean() throws JMSEException;
    byte readByte() throws JMSEException;
    int readUnsignedByte() throws JMSEException;
    short readShort() throws JMSEException;
    int readUnsignedShort() throws JMSEException;
    char readChar() throws JMSEException;
    int readInt() throws JMSEException;
    long readLong() throws JMSEException;
    float readFloat() throws JMSEException;
    double readDouble() throws JMSEException;
    String readUTF() throws JMSEException;
    int readBytes(byte[] value) throws JMSEException;
    int readBytes(byte[] value, int length) throws JMSEException;

    // Write methods
    void writeBoolean(boolean value) throws JMSEException;
    void writeByte(byte value) throws JMSEException;
    void writeShort(short value) throws JMSEException;
    void writeChar(char value) throws JMSEException;
    void writeInt(int value) throws JMSEException;
    void writeLong(long value) throws JMSEException;
    void writeFloat(float value) throws JMSEException;
    void writeDouble(double value) throws JMSEException;
    void writeUTF(String value) throws JMSEException;
    void writeBytes(byte[] value) throws JMSEException;
    void writeBytes(byte[] value, int offset, int length) throws
        JMSEException;
    void writeObject(Object value) throws JMSEException;

    void reset() throws JMSEException;
}

```

This interface is extremely similar to the `StreamMessage` interface I discussed earlier.

`BytesMessages` are typically used for literally duplicating the body of one of the other message types. Typically, applications use one of the four self-describing message types instead of using `BytesMessages`. However, `BytesMessages` are useful in situations where one needs to read in raw data, for example, from a disk file and transfer it "as is" (without any conversion at all, such as Big Endian/little Endian, etc.) to another machine and/or location.

To create a `BytesMessage`, a client uses the `createBytesMessage` method on the session as follows:

```
// Assume session is a valid Session object.  
BytesMessage bytesMsg = session.createBytesMessage ();
```

Using the `reset` and `clearBody` methods

When a bytes message is first created the body of the message is in write-only mode. After a call to the `reset` method has been made, the message body is in read-only mode. When a client sends a bytes message, the provider calls `reset` in order to read it's content, and actually send the message across. When the bytes message is received on the other side, the provider reconstructs the message and calls `reset` on it so that the message body is in read-only mode for the client. Now if the client calls `clearBody` on this message, the message body is cleared and the message body is in write-only mode. This is similar to the stream message that we saw earlier. Also remember, if a client attempts to read a message in write-only mode, a `MessageNotReadableException` is thrown. If a client attempts to write a message in read-only mode, a `MessageNotWriteableException` is thrown.

JMS Note:

Although the interfaces are very similar, the `BytesMessage` does *not* allow conversion between types as do the `StreamMessage` and `MapMessage`.

Message Selection

The number of messages going back and forth in a message based system can be overwhelming, especially if a client is not interested in all the messages it receives. In the simplest case, the client can be held responsible for browsing through the message and discarding the ones it is not interested in. There are several problems with this approach as outlined below:

1. The client has the burden of filtering out all the messages it does not need. This is extra work for each client in the system.
2. Even though the client is not interested in a message, it is still delivered to it thus wasting both processing time and network bandwidth.
3. A direct result of 1 and 2 is reduced scalability and performance.

An alternative approach is to separate the filtering criteria from the actual body of the message. This allows the provider to handle much of the filtering and routing work that would otherwise need to be done by the client. 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 can attach application-specific selection criteria to messages using message properties. Clients specify message selection criteria using JMS message selector expressions. These expressions can be based on any of the information available in the header and properties (application, standard, and provider specific) parts of the message. Only messages whose headers and properties match the specified selector are delivered to the client. A message selector matches a message if the selector expression evaluates to `true` when the message's header field and property values are substituted for their corresponding identifiers in the selector.

The Message Selector Syntax

The message selector syntax is a subset of the **SQL92** conditional expression syntax. Following are points to note about this syntax:

1. Predefined selector literals and operator names are case insensitive. So, the operator "OR" is the same as "or".
2. Rules for Identifiers (*These are the rules that must be followed when naming application specific properties*) :
 - a. An identifier does not have a limit on the number of character in its name.
 - b. An identifier name must begin with the a "start" character, which is any character for which the `Character.isJavaIdentifierStart` method returns a `true`. A character may start a Java identifier if and only if it is one of the following:
 - i.a letter
 - ii.a currency symbol (such as "\$")
 - iii.a connecting punctuation character (such as "_").
 - c. All other characters in the identifier following the first must return a `true` value when passed to the `Character.isJavaIdentifierPart` method. A character may be part of a Java identifier if and only if it is one of the following:
 - i.a letter
 - ii.a currency symbol (such as "\$")
 - iii.a connecting punctuation character (such as "_").
 - iv.a digit
 - v.a numeric letter (such as a Roman numeral character)
 - vi.a combining mark
 - vii.a non-spacing mark
 - viii.an ignorable control character
 - d. Identifiers cannot be the following reserved names: Identifiers cannot be the names: NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, and IS.
 - e. Identifiers must be either header field references or property references.
 - f. **Identifiers are case sensitive.**
 - g. Message header field references are restricted to `JMSDeliveryMode`, `JMSPriority`, `JMSMessageID`, `JMSTimestamp`, `JMSCorrelationID`, and `JMSType`.
 - h. Any name beginning with 'JMSX' is a JMS defined property name.
 - i. Any name beginning with 'JMS_' is a provider-specific property name.
 - j. Any name that does not begin with 'JMS' is an application-specific property name. If a property is referenced that does not exist in a message its value is NULL. If it does exist, its value is the corresponding property value.
3. Rules for literals
 - a. A string literal is enclosed in single quotes. For example, 'a string literal'.
 - b. 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.
 - c. An approximate numeric literal is a numeric value in scientific notation such as 7E3, -57.9E2 or a numeric value with a decimal such as 7., -95.7, +6.2; numbers in the range of Java double are supported.
 - d. The boolean literals TRUE and FALSE are supported.
4. All spaces, horizontal tabs, form feeds and line terminators are considered whitespace and follow the same rules as in Java.
5. Expression Evaluation
 - a. Standard bracketing () for ordering expression evaluation is supported.
 - b. Expressions are evaluated from left to right. Paranthesis can be used to alter the order. For example, $5 + 4 * 2$ evaluates to 18, but $5 + (4 * 2)$ evaluates to 13.
 - c. Logical operators in precedence order: NOT, AND, OR
 - d. Comparison operators: =, >, >=, <, <=, <> (not equal)

- i. 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 selector is always evaluated to false.
 - ii. String and Boolean comparison is restricted to = and <>.
 - e. Precedence of Arithmetic operators is as follows:
 - i. +, - (unary)
 - ii. multiplication and division
 - iii. addition and subtraction
- 6. Miscellaneous Operators
 - a. [NOT] BETWEEN
 - i. age BETWEEN 15 and 19 is equivalent to age >= 15 AND age <= 19
 - ii. age NOT BETWEEN 15 and 19 is equivalent to age < 15 OR age > 19
 - b. [NOT] IN
 - i. Country IN ('UK', 'US', 'France') is true for 'UK' and false for 'Peru' and is equivalent to the expression (Country = 'UK') OR (Country = 'US') OR (Country = 'France').
 - c. [NOT] LIKE
 - i. Used for pattern matching. '_' stands for any single character, '%' stands for any sequence of characters, and all other characters stand for themselves.
 - ii. phone LIKE '12%3' is true for '123' '12993' and false for '1234'
 - iii. word LIKE 'l_se' is true for 'lose' and false for 'loose'
 - d. IS NULL
 - i. Color IS NULL evaluates to true if the value of Color is NULL or if it's not present.
 - e. IS NOT NULL
 - i. Color IS NOT NULL evaluates to true if the value of Color is not NULL.

To clarify, let's look at the following code fragment:

```
String selector="(JMSType='AccountsPayable' AND DollarCredit>5000)";
selector = selector + " OR (JMSXUserID = 'johndoe')";

// Assume that session is a valid QueueSession
// and queue is a valid Queue.
receiver = session.createReceiver(queue,selector);
```

As shown above, a selector expression is specified while creating the message consumer and is passed in as the second parameter. The JMS specification mandates that JMS providers 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 `InvalidSelectorException` being thrown. If a selector is specified, a consumer will only receive messages that match the selection criteria.

Let's consider two messages:

The first message has the following relevant header properties:

- JMSType = 'AccountsPayable'
- DollarCredit = 2500

This message will not reach the consumer with the above selector since the DollarCredit property value is not greater than 5000, which makes the first half of the selector evaluate to false and JMSXUserID is not present at all, which makes the second half false as well.

The second message has the following relevant header properties:

- JMSType='AccountsPayable'
- DollarCredit = 6000
- JMSXUserID = 'johndoe'

This message would be received because the second half of the selector evaluates to true, thus making the entire selector (in this case) true.

Summary

Messages are the lifeblood of a message based system. Therefore, the importance of having a firm grasp over the JMS message model cannot be overstated. In this chapter, I've covered this model in detail. We've discussed the structure of JMS messages, its components, the different types of messages, and the message selection syntax. The JMS message model is simple yet powerful. It provides developers with a unified message API that enables the creation of messages with any type of data and supports the development of heterogeneous message based applications that span operating systems, machine architectures, and computer languages.

Chapter 6

The JMS Messaging Styles

In chapter 2, I introduced you to the various messaging styles supported by JMS. To recap, JMS supports both messaging styles that are most popularly available [and in use] in commercial enterprise-level messaging products today. These are *point-to-point* and *publish-and-subscribe*. Since many messaging systems [may] only support one of these styles, JMS provides a separate domain for each and defines compliance for each domain. *This means that messaging products can be JMS compliant even if they do not support both messaging styles.* In addition, JMS supports a third [variation] style known as *request/reply*, which applies to both the point-to-point and publish-and-subscribe styles.

In this chapter, I will discuss all three of these styles as they relate to JMS in detail along with code examples to clarify. As in chapter 2, the examples in this chapter are based on Sun Microsystem's Java Message Queue product and will be dependent on it. In chapter 8, I will show you a technique I use to get rid of this dependency.

Let's look at the point-to-point style first.

Point-To-Point Messaging

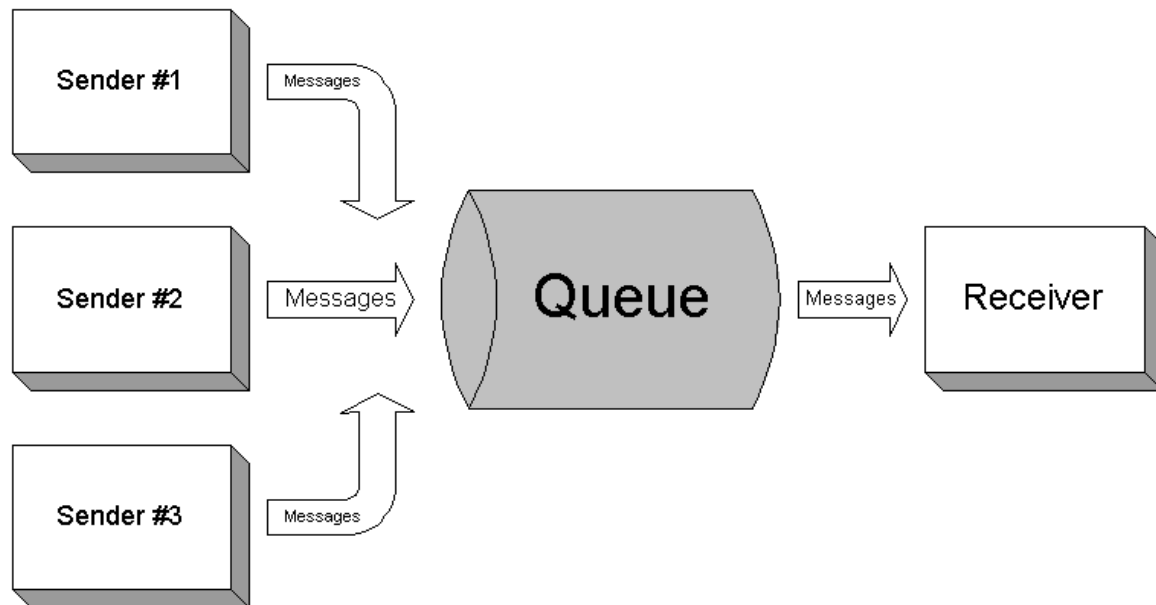


Figure 1: Point-to-point Messaging Style

Following are the basic components/pieces of JMS involved in this style:

- The Destination is known as Queue.
- The ConnectionFactory is known as a QueueConnectionFactory.
- The Connection is known as QueueConnection.
- The Session is known as a QueueSession
- The MessageProducer is known as a QueueSender.

- The MessageConsumer is known as a QueueReceiver.

Note the use of the word *Queue* in every piece of JMS associated with this style.

In this style, a messaging product is used by two applications to communicate with each other, often as an asynchronous replacement for RPC. As shown in figure 1, multiple senders can send messages to a single receiver. JMS does not specify the support for multiple receivers. Some JMS providers may allow multiple receivers for a single queue, but remember, this support will be vendor specific.

In the following sections I'll discuss each one of the basic pieces involved in the point-to-point messaging style. These pieces were identified above. Let's start with the QueueConnectionFactory.

QueueConnectionFactory

As shown in figure 2 in chapter 3, the QueueConnectionFactory interface is actually derived from the ConnectionFactory interface. The QueueConnectionFactory interface is shown below:

```
public interface QueueConnectionFactory
    extends ConnectionFactory {
    QueueConnection createQueueConnection()
        throws JMSEException;
    QueueConnection createQueueConnection(String userName,
        String password) throws JMSEException;
}
```

As its name implies, the queue connection factory is used for creating queue connections. The createQueueConnection method instructs the JMS provider to create a queue connection with default user identity. The exact meaning of the default user identity is provider specific. The other version of the createQueueConnection method takes a username and password and instructs the JMS provider to create a queue connection with these user credentials. This version of the method may throw a JMSSecurityException if client authentication fails due to a invalid username or password. Both methods throw a JMSEException if the queue connection cannot be created.

Like its parent, the queue connection factory is an administrable object as well. JMS does not define any facilities for creating, administering, or deleting queue connection factories. These facilities are provided by the JMS provider and hence are provider specific.

QueueConnection

As discussed above a queue connection is obtained through a queue connection factory. Figure 3 in chapter 3 showed that the QueueConnection interface derives from the Connection interface. The QueueConnection interface is shown below:

```
public interface QueueConnection extends Connection {
    QueueSession createQueueSession(boolean transacted,
        int acknowledgeMode) throws JMSEException;
    ConnectionConsumer createConnectionConsumer(Queue queue,
        String messageSelector, ServerSessionPool sessionPool,
        int maxMessages) throws JMSEException;
}
```

Just as the queue connection factory is a factory for creating queue connections, the queue connection is a factory for creating queue sessions through the createQueueSession method. This method takes two parameters; the first parameter indicates whether the session is transacted and the second parameter specifies the message acknowledgement mode. I've discussed both of

these in detail in chapters 2 and 3. This method will throw a `JMSEException` if the connection fails to create a session due to some internal error or a lack of support for specific transaction and/or acknowledgement mode.

This interface also has an additional method `createConnectionConsumer`. The JMS specification has flagged this method as an optional method that providers may choose not to implement and further states that this is an expert facility that is not used by regular clients. Most providers do not implement this method and so I will not discuss it in this book.

QueueSession

As shown in figure 4 in chapter 3, the `QueueSession` interface is actually derived from the `Session` interface. The `QueueSession` interface is shown below:

```
public interface QueueSession extends Session {
    Queue createQueue(String queueName) throws JMSEException;
    QueueReceiver createReceiver(Queue queue)
        throws JMSEException;
    QueueReceiver createReceiver(Queue queue,
        String messageSelector) throws JMSEException;
    QueueSender createSender(Queue queue) throws JMSEException;
    QueueBrowser createBrowser(Queue queue) throws JMSEException;
    QueueBrowser createBrowser(Queue queue,
        String messageSelector) throws JMSEException;
    TemporaryQueue createTemporaryQueue() throws JMSEException;
}
```

The `createQueue` method is somewhat misleading. The JMS specification states that this method is provided "for the rare cases where clients need to dynamically manipulate queue identity. This allows the creation of a queue identity with a provider specific name. **Clients that depend on this ability are not portable.**"

Note – createQueue caveat

Different providers have slightly different behaviors for this method. For example, Sun's Java Message Queue actually allows the creation of the queue (as the name implies). Most JMS compliant messaging products require the use of their administration facilities to actually create the queue beforehand. This method in those providers merely lets you get a reference to the queue to pass into other methods, such as `createReceiver` and `createSender`.

The `createReceiver` method is used to create a queue receiver to receive messages from a specified queue. I'll discuss both the queue and queue receiver objects in more detail in a moment. If an invalid queue is specified, `InvalidDestinationException` will be thrown. A second version of this method takes a message selector string as a parameter as well. Message selectors were discussed in detail in chapter 5. By specifying a message selector, only those messages whose header meet the criteria specified by the selector will be delivered to the receiver.

The `createSender` method is used to create a sender for the specified queue. If the sender cannot be created for any reason, a `JMSEException` will be thrown. I'll discuss queue senders later in a moment as well. The `createBrowser` method is used to create a browser to peek at the messages on a specified queue. The `createBrowser` methods are similar to the `createReceiver` methods. A queue browser implements the `QueueBrowser` interface shown below:

```
public interface QueueBrowser {
    Queue getQueue() throws JMSEException;
    String getMessageSelector() throws JMSEException;
    Enumeration getEnumeration() throws JMSEException;
    void close() throws JMSEException;
}
```

```
}
```

The `getQueue` and `getMessageSelector` methods return a reference to the queue and message selector specified in the `createBrowser` methods. The `close` method should be called whenever a browser is no longer needed to free any resources that the JMS provider may have allocated on behalf of this browser. This is just good programming practise in general. The most interesting method of this interface is the `getEnumeration` method. This method returns an enumeration of all the messages in the queue. Actually this enumeration may be an enumeration of the entire content of the queue or it may only contain the messages matching a message selector if one was specified.

JMS Note

New messages may be arriving and old ones expiring while the the client is browsing the queue using the enumeration returned by the `getEnumeration` method of the queue browser. 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.

The `createTemporaryQueue` method on the `QueueSession` interface is used to create a temporary queue, which we'll discuss in the next section.

Queue

As shown in figure 1 in chapter 3, the `Queue` interface is actually derived from the `Destination` interface. The `Queue` interface is shown below:

```
public interface Queue extends Destination {
    String getQueueName() throws JMSEException;
    String toString();
}
```

The `getQueueName` method returns the name of the queue as used by the messaging product while the `toString` method returns a "prettier" version of this name. As a general rule, clients should not depend on the name returned by these methods in their code, since these names may be provider specific and may not be the same as the name defined by the queue administrator.

Like its parent, the queue object is an administrable object as well. There are two types of queues – long lived and temporary. JMS does not define any facilities for creating, administering, or deleting long lived queues. These facilities are provided by the JMS provider and hence are provider specific. Remember, JMS does define a standard way to gain access to a long lived queue through the queue session via the `createQueue` method that we saw above. Temporary queues, on the other hand can be created programmatically through the queue session via the `createTemporaryQueue` method. Such queues are valid only with [and for the life of] the queue connection to which the queue session used to create the temporary queue belonged. Temporary queues implement the `TemporaryQueue` interface shown below:

```
public interface TemporaryQueue extends Queue {
    void delete() throws JMSEException;
}
```

The `delete` method is used to delete the temporary queue. If there are still existing senders or receivers still using it, then a `JMSEException` will be thrown. It is good programming practise to call the `delete` method on a temporary queue when it is no longer required to free up any resources consumed by the queue.

QueueReceiver

As shown in figure 5 in chapter 3, the `QueueReceiver` interface is actually derived from the `MessageConsumer` interface. The `QueueReceiver` interface is shown below:

```

public interface QueueReceiver extends MessageConsumer {
    Queue getQueue() throws JMSEException;
}

```

The `getQueue` method returns a reference to the queue object specified in the `createReceiver` method of the queue session. This may not be a reference to the exact same physical object. Consider the following code fragment:

```

// queue is a valid Queue
// queueSession is a valid QueueSession
receiver = queueSession.createReceiver(queue);
queue1 = receiver.getQueue();
if(queue == queue1)
    System.out.println("Same physical object.");
else
    System.out.println("Not the same physical object.");

```

Depending on the provider any one of the two statements may be printed. But the statement

```
queue.equals(queue1)
```

should always evaluate to `true`.

QueueSender

As shown in figure 6 in chapter 3, the `QueueSender` interface is actually derived from the `MessageProducer` interface. The `QueueSender` interface is shown below:

```

public interface QueueSender extends MessageProducer {
    Queue getQueue() throws JMSEException;
    void send(Message message) throws JMSEException;
    void send(Message message, int deliveryMode, int priority,
        long timeToLive) throws JMSEException;
    void send(Queue queue, Message message)
        throws JMSEException;
    void send(Queue queue, Message message, int deliveryMode,
        int priority, long timeToLive) throws JMSEException;
}

```

The `getQueue` method returns a reference to the queue object specified in the `createSender` method of the queue session. This may not be a reference to the exact same physical object. I discussed this above with an example code fragment. The interface has a variety of `send` methods. One of the first two `send` methods are used when a queue is specified during the creation of the sender using the `createSender` method of the queue session object. The first method uses the default values for the delivery mode, priority, and time-to-live parameters. The last two `send` methods are used when a queue is not specified during the sender creation i.e. a null value is passed in to the `createSender` method. The first parameter to these two methods is a valid queue to which to send the message. Note that all four `send` methods can never be used in the same sender. If a queue was specified during the creation of the sender, then attempting to use the last two `send` methods will throw a `UnsupportedOperationException`. On the other hand if no queue was specified, it is quite obvious that the first two `send` methods cannot be used. All `send` methods will throw a `MessageFormatException` if there is a formatting error in the message, an `InvalidDestinationException` if the queue is invalid, and a `JMSEException` if there is any other error during the send.

An Example

Let's tie all these concepts together with an example program. This is the first complete example in this book (Hoorah!). In this example, I will attempt to simulate a simple phone i.e. not 3-way conferencing, call waiting, etc. Obviously, I will do this using the point-to-point messaging style of

a JMS provider, which in this case will be Sun's Java Message Queue. There is no special reason for selecting this product over others in the market apart from the fact that I need "a" JMS provider that supports the point-to-point messaging style and is not horrendous to install and setup. You will also need JDK 1.1.6 or higher.

Note

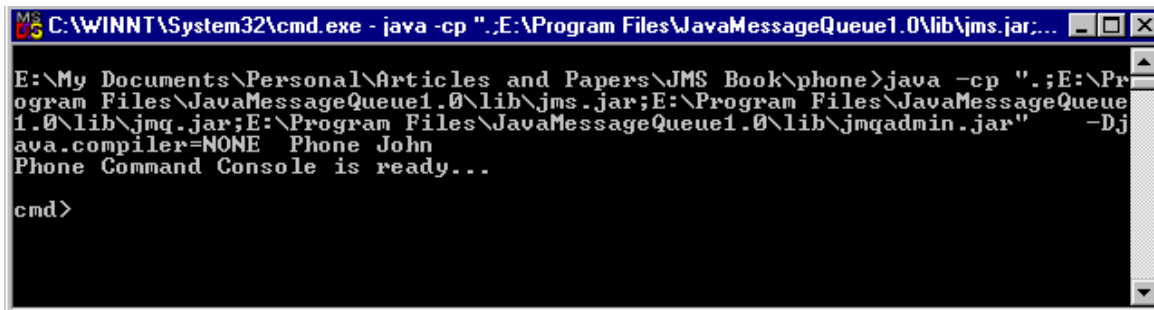
The only Sun specific part of this program is how I obtain the queue connection factory. In chapter 8, I will show you a technique I use to make my JMS clients "provider independent".

When the phone is started, it is given the name of the line/user to which the phone belongs. For example:

```
REM Set up the classpath. My installation of Sun's Java Message Queue
REM is in the directory E:\Program Files\JavaMessageQueue1.0
set CPATH=.;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmq.jar;
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar

java -cp "%CPATH%" -Djava.compiler=NONE Phone John
```

Here "John" is the name that this phone will be recognized with. This will result in the creation of a command console, which is the same window/shell from which the program was started and an "Output and Information" GUI window. These windows are shown in figures 2 and 3 respectively.



```
C:\WINNT\System32\cmd.exe - java -cp ".;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar;...
E:\My Documents\Personal\Articles and Papers\JMS Book\phone>java -cp ".;E:\Pr
ogram Files\JavaMessageQueue1.0\lib\jms.jar;E:\Program Files\JavaMessageQueue
1.0\lib\jmq.jar;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar" -Dj
ava.compiler=NONE Phone John
Phone Command Console is ready...

cmd>
```

Figure 2: The phone command console window

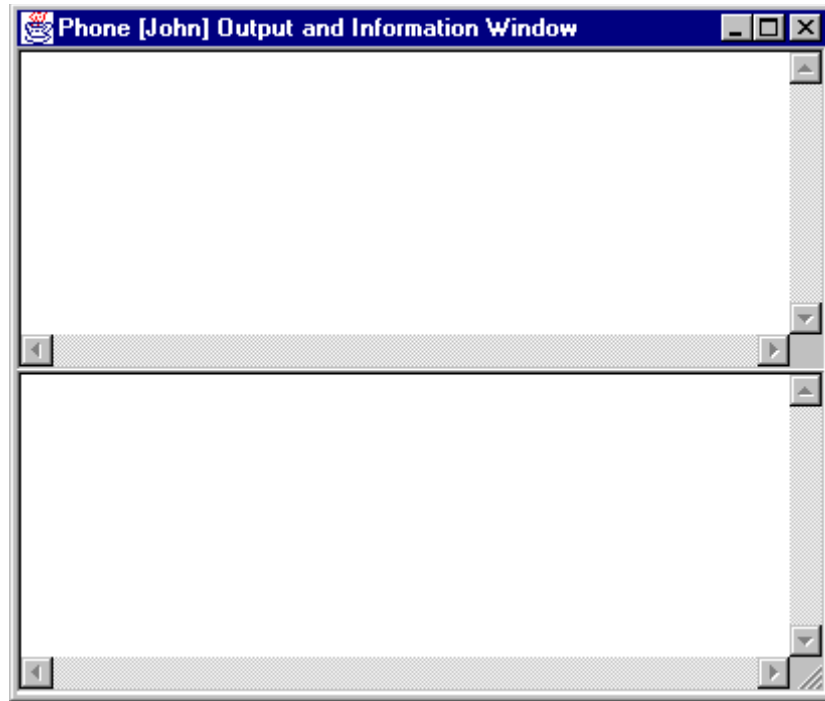


Figure 3: The phone Output and Information window.

There are two types of messages in this system. *Control messages* always begin with "ControlMessage:XXX", where XXX is the name of an action. *User messages* are messages that the user sends during normal conversation using the phone. These messages are sent using the `say` command (discussed below). The upper portion of the "Output and Information" window is the output panel and is used to display user messages. The lower portion is the information panel, which is used for displaying control messages and other useful information.

The command console is where all the action takes place. To get a list of all supported commands, type in "help" at the prompt. At this point you should see the following output:

```
Supported commands are: dial <destination>, disconnect, hangup, help, and say
<message>
```

Initially, the only valid commands are `dial` and `disconnect` (and `help` of course). Typing in `disconnect` ends the phone program. This is the only way to cleanly shutdown the phone program. To dial up to another phone type in "dial" followed by the destination i.e. the user/line name of the other phone. For example:

```
dial Bob
```

, where "Bob" has been started up in a similar way to "John". If the phone "Bob" does not exist you will see the following error message on the console at 15 seconds.

```
No Answer. Line may have been disconnected.
```

Also, look at the information panel of the GUI window for interesting messages going back and forth between "John" and "Bob" during the dial-up process. Don't worry if all the messages don't make sense right now. Trust me they will by the end of this section.

The only valid commands at this point are `say`, `disconnect`, and `hangup` (and `help` of course). Both "Bob" and "John" can send messages to each other. For example, "John" could do something like:

```
say Hi, Bob! How are you doing?
```

At any point either "Bob" or "John" could end the call by typing in `hangup`.

So, how does all this work under the covers?

The `main` Method

Let's start by looking at the `main` method of the program. This program takes one parameter – the name of the line/user to which the phone belongs. I will show you how this name is important later. This method creates a new instance of the phone passing in the name and starts an instance of the class `MainThread` as follows:

```
// Create a new phone
Phone phone = new Phone(args[0]);
// Start the main thread for the command console.
MainThread mainThread = new MainThread(phone);
mainThread.start();
```

The `MainThread` class extends the `java.lang.Thread` class. This class encapsulates the command console functionality. Let's analyze this class step-by-step:

1. Wait for the user to type in commands at the command console.
2. Process the commands and invoke the appropriate method on the `phone` instance. For example, the following code processes the `disconnect` command:

```
if( cmd.startsWith("disconnect") ) {
    try {
        // Call the disconnect method on the phone.
        phone.disconnect();
    }
    catch( Exception e ) {
        System.err.println(e.getMessage());
    }
    // End the JVM.
    // Don't worry, the disconnect method has done all the
    // cleanup.
    System.exit(-1);
}
```

The code for processing the `dial` and `say` commands is a little more involved, since both these commands take a parameter. The code finds this parameter and passes it to the `dial` and `say` methods on the phone respectively. The code for the `dial` command is shown below:

```
if( cmd.startsWith("dial") ) {
    try {
        // find the "destination" parameter
        int i = 4;
        for( ; i < cmd.length(); i++ )
            if( cmd.charAt(i) != ' ' )
                break;

        // Not found? That's an error.
        if( i == cmd.length() )
            System.out.println("You must specify a destination.");
        else
            // Pass the destination to the dial method.
```

```

        phone.dial(cmd.substring(i));
    }
    catch( Exception e ) {
        System.err.println(e.getMessage());
    }
}

```

The Phone Constructor

Now let's take a look at the constructor of the `Phone` class. The one and only constructor does the following, in order:

1. Save the name of this user/line in the `localQueueName` member variable. All member variables pertaining to this phone begin with "local".
2. Create and show the "Output and Information" GUI window by calling the `showGUI` method.
3. Get the queue connection factory as follows:

```

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

```

Remember, this is the only Sun specific code in this program.

4. Use the connection factory to get the queue connection as follows:

```

connection = connectionFactory.createQueueConnection();

```

5. Create five queue sessions using this queue connection as follows:

```

localSessionSender = connection.createQueueSession(false,1);
localSessionReceiver =connection.createQueueSession(false,1);
localSessionListener =connection.createQueueSession(false,1);
remoteSessionSender = connection.createQueueSession(false,1);
remoteSessionReceiver=connection.createQueueSession(false,1);

```

All sessions are non-transacted and support auto acknowledgement of messages. The reason we have five different sessions is to support concurrent sending and receiving of messages. Remember, the session is a single threaded object.

Note: This is probably the most important aspect of this program.

6. Use the `localSessionListener` session object to get a reference to a queue object for the listener queue, which has the name `localQueueName + "Listener"`, and create a queue receiver for this queue as follows:

```

javax.jms.Queue queue =
    localSessionListener.createQueue(localQueueName +
        "Listener");
localListener = localSessionListener.createReceiver(queue);

```

Now start a thread instance of the class `ListenerThread` passing it the `phone` instance (`this`) and the `localListener` receiver. This thread will continuously wait for "Dial" messages. The "Dial" message format is as follows:

```

ControlMessage:Dial <Name>

```

, where `Name` is the name of the user/line that sent the message.

When a "Dial" message is received this thread will call the `dialReceived` method on the `phone` instance. The name of the remote caller is passed in as the parameter to this method call.

Note

For each user of the phone, three queues will have to be setup. For example, if a username is "John", the three queues would be "JohnListener", "JohnSender", and "JohnReceiver". However, Sun's provider automatically creates these queues when the `createQueue` method is called on the queue session. So no special work is required. See the "createQueue caveat" sidebar.

7. Finally, after all the setup work is done, the connection is started as follows:

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

The ReceiverThread class

At this point let's take a look at the `ReceiverThread` class. The constructor gets a reference to a phone and a queue receiver. One started, this thread continuously waits for a message sent to this receiver. If the message is a "Hangup" message, the thread will call the `hangupReceived` method on the phone instance. The "Hangup" message format is as follows:

```
ControlMessage:Hangup
```

Any other message is considered a "user" message and is displayed in the output panel. How this is done will be explained in the section "Multi-threaded access to the GUI components".

Now let's take a look at the methods of phone that are critical to its operation.

The dial Method

This method is called by the command console thread when the user types in the `dial` command at the command console. This method takes one parameter – the name of the remote/other user/line to dial up. This method proceeds as follows:

1. Call the `isBusy` method to see if the phone is busy i.e. already in use. If the phone is busy then a `PhoneException` with the message "You must hangup the existing call first" is thrown.

```
if( isBusy() )
    throw new PhoneException(
        "You must hangup the existing call first.");
```

2. Send a "Dial" message to the remote phone's listener. The name of the queue is the remote user/line's name plus "Listener". The name of this phone's user i.e. the value of `localQueueName`, is sent as part of this message. The code for this is as follows:

```
// Send a "Dial" message to the other phone's listener.
javax.jms.Queue queue =
    remoteSessionSender.createQueue(
        remoteQueueName + "Listener");
QueueSender remoteListener =
    remoteSessionSender.createSender(queue);
sendMessage(remoteSessionSender, remoteListener,
    "ControlMessage:Dial " + localQueueName);
```

Let's go back to our earlier example in which "John" typed in the command "dial Bob" at the command console. This would result in the message "ControlMessage:Dial John" being sent to the BobListener queue.

Let's look at the `sendMessage` method. This is a helper method that takes a queue session, a queue sender, and a string message as its parameters and sends the string message using the provided session and sender objects as follows:

```
TextMessage msg = session.createTextMessage();
msg.setText(message);
sender.send(msg, DeliveryMode.NON_PERSISTENT, 9, 15000);
```

The message is sent with a timeout of 15 seconds and a non-persistent delivery mode. This is because we don't want messages staying around forever. But, why 15 seconds? As you'll see in the next step, that's how long this method is going to wait for a response to the "Dial" message. Also, reliability is not a key concern of this program, so it is safe to use the non-persistent delivery mode.

3. The method now waits for 15 seconds for a response from the remote phone on the queue whose name is this phone's user/line name plus "Receiver".

```
queue = localSessionReceiver.createQueue(
    localQueueName + "Receiver");
localReceiver = localSessionReceiver.createReceiver(queue);
TextMessage msg = (TextMessage)localReceiver.receive(15000);
```

In our example, the queue name would be "JohnReceiver".

4. At this point one of four things may happen
 - a. No message is received, which the method interprets as a sign that the requested phone line does not exist and a `PhoneException` with the message "No answer. Line may have been disconnected" is thrown.
 - b. An invalid response is received, which is an error condition and a `PhoneException` with the message "Invalid Control Response received" is thrown.
 - c. A "Busy" message is received. This is a message of the form "ControlMessage:Busy". A `PhoneException` with the message "Line is Busy" is thrown.
 - d. An "OK" message is received. This is a message of the form "ControlMessage:OK". Go to step 5 from here.
5. The method uses the `remoteSenderSession` session object to create a queue sender on the queue whose name is formed by taking the name of the remote phone and appending "Receiver" to it. Why append "Receiver" and not "Sender"? Because, this is the queue that the remote expects to receive messages on. So that's the queue we'll send messages on.

```
// Send to remote's receiver
queue = remoteSessionSender.createQueue(
    remoteQueueName + "Receiver");
remoteSender = remoteSessionSender.createSender(queue);
```

Similarly, the method uses the `remoteReceiverSession` object to create a queue receiver on the queue whose name is formed by taking the name of the remote phone and appending "Sender" to it. This is the queue that the remote will send messages on. The method then creates a new instance of the `ReceiverThread` class and gives this queue receiver to it.

```
// Receive from remote's Sender.
queue = remoteSessionReceiver.createQueue(
    remoteQueueName + "Sender");
```

```

remoteReceiver = remoteSessionReceiver.createReceiver(queue);
remoteReceiverThread =
    new ReceiverThread(this,remoteReceiver);
remoteReceiverThread.start();

```

In our example, these queues would be "BobReceiver" and "BobSender" respectively.

6. Finally, the method saves the remote name to the `remoteQueueName` member variable and marks the phone as busy.

The dialReceived Method

This method is called by the listener thread when a "Dial" message is received. This method takes one parameter – the name of the remote phone user/line that sent the "Dial" message. So, in our example, this method would be called on "Bob" in response to a "Dial" message sent by "John".

If the phone is busy, the method sends a "Busy" message back to the remote phone on the queue whose name is formed by taking the remote phone's user/line name and appending "Receiver" to it. This is the queue that the remote phone expects a response to the "Dial" message. If the phone is not busy then the method does the following:

1. Send back an "OK" message.
2. Create a queue receiver on the queue whose name is formed by taking this phone's name and appending "Receiver" to it. This is the queue that the remote phone will send messages to. The method then creates a new instance of the `ReceiverThread` class and gives this queue receiver to it.

```

// remote phone will send on this queue i.e. our receiver
queue = localSessionReceiver.createQueue(localQueueName+
    "Receiver");
localReceiver = localSessionReceiver.createReceiver(queue);
localReceiverThread = new ReceiverThread(this,localReceiver);
localReceiverThread.start();

```

In our example, this queue would be called "BobReceiver".

3. Create a queue sender on the queue whose name is formed by taking this phone's name and appending "Sender" to it. This is the queue that the remote phone will expect to receive messages.

```

// remote will receive on this queue i.e. our sender
queue = localSessionSender.createQueue(localQueueName +
    "Sender");
localSender = localSessionSender.createSender(queue);

```

In our example, this queue would be called "BobSender".

The hangup Method

This method is called by the command console thread when the user types in the `hangup` command at the command console. If the phone is not busy, the method throws a `PhoneException` with the message "Not dialed anywhere" otherwise a "Hangup" message is sent. Depending on which queues are being used for this connection, the receiver and sender are closed, and the receiving thread is stopped.

The hangupReceived Method

This method is called by the receiver thread when it receives a "Hangup" message. This method is exactly similar to the `hangup` method with the exception that it does not send a "Hangup" message. If it did we would have an infinite loop of "Hangup" messages going back and forth.

The disconnect Method

This method is called by the command console thread when the user types in the `disconnect` command at the command console. If the phone is "dialed up" then this method hangs up the call using the `hangup` method discussed above. It then calls the private `shutdown` method. Together, the `hangup` and `shutdown` methods ensure a clean shutdown of the phone.

Multi-threaded access to the GUI components

Java AWT components are single threaded and hence must be accessed by only one thread at a time. The safest way to access these components is through the thread that the JVM creates for handling these components (surprise!). So, what I've done is create text buffers (Strings) for holding the content to be displayed in the output and information panels. These buffers are accessed through thread safe i.e. synchronized methods only. Any time I need to change the text in any of the two panels, I change the appropriate text buffer and schedule a "screen update" by calling the `repaint` method. For example,

```
debugText += ("Dialing out to " + remoteQueueName + "\n");
repaint();
```

As a result of this call, the JVM will call the `paint` method, which I override, with the correct thread.

```
public void paint(Graphics g) {
    super.paint(g);
    synchronized(this) {
        messageAreaDebug.setText(debugText);
        messageAreaIn.setText(inText);
    }
}
```

Note how I synchronize access to the text buffers. The contents of these buffers are now shown in the panels.

The phone application shown in its entirety is as follows:

```
import java.awt.*;
import java.awt.event.*;
import javax.jms.*;

public class Phone extends Frame {

    // JMS Queue Related member variables
    // Note that each sender and receiver are created using a
    // separate session. This allows concurrent sending and
    // receiving without violating the single-threadedness
    // of the session. (Refer to chapter 3 for a refresher).
    private QueueConnectionFactory connectionFactory = null;
    private QueueConnection connection = null;

    private String localQueueName = null;
    private QueueSession localSessionSender = null;
    private QueueSession localSessionReceiver = null;
    private QueueSession localSessionListener = null;
    private QueueSender localSender = null;
    private QueueReceiver localReceiver = null;
    private QueueReceiver localListener = null;

    private String remoteQueueName = null;
    private QueueSession remoteSessionSender = null;
    private QueueSession remoteSessionReceiver = null;
    private QueueSender remoteSender = null;
    private QueueReceiver remoteReceiver = null;
```

```

// This member variable is set to true if the phone
// is "busy".
private boolean busy = false;

// The three threads...
// one thread to listen for incoming calls
// one thread to receive messages if the call is using the local
// lines and one thread to receive messages if the call is using
// remote lines.
private ReceiverThread localReceiverThread = null;
private ListenerThread localListenerThread = null;
private ReceiverThread remoteReceiverThread = null;

// member variables for GUI.
private TextArea messageAreaIn = null;
private TextArea messageAreaDebug = null;
String debugText = "";
String inText = "";

public static void main(String args[]) {
    if( args.length != 1 ) {
        System.err.println("You must pass in the name for this line.");
        System.exit(-1);
    }

    try {
        // Create a new phone
        Phone phone = new Phone(args[0]);
        // Start the main thread for the command console.
        MainThread mainThread = new MainThread(phone);
        mainThread.start();
    }
    catch( Exception e ) {
        System.out.println(e.getMessage());
        System.exit(-1);
    }
}

public Phone(String localQueueName) throws JMSEException
{
    this.localQueueName = localQueueName;
    showGUI();

    // Get the queue connection factory.
    // This is the only "Sun Java Message Queue" specific code.
    connectionFactory =
    new com.sun.messaging.QueueConnectionFactory();

    // Use the factory to create the queue connection.
    connection = connectionFactory.createQueueConnection();

    // setup this side's phone line...
    // All sessions are non-transacted and auto ack.
    localSessionSender = connection.createQueueSession(false,1);
    localSessionReceiver = connection.createQueueSession(false,1);
    localSessionListener = connection.createQueueSession(false,1);
    javax.jms.Queue queue =
    localSessionListener.createQueue(localQueueName + "Listener");
    localListener = localSessionListener.createReceiver(queue);

    // listen for incoming calls
    localListenerThread = new ListenerThread(this,localListener);
    localListenerThread.start();

    // setup remote line receiver and sender
    remoteSessionSender = connection.createQueueSession(false,1);
    remoteSessionReceiver = connection.createQueueSession(false,1);

```



```

    // All setup work is done so start the connection
    connection.start();
}

synchronized boolean isBusy() {
    return(busy);
}

// This method is used to dial out to another phone.
synchronized void dial(String remoteQueueName)
    throws JMSEException, PhoneException
{
    // Only one call at a time...
    if( isBusy() )
        throw new PhoneException(
            "You must hangup the existing call first.");

    debugText += ("Dialing out to " + remoteQueueName + "\n");
    repaint();

    // Send a "Dial" message to the other phone's listener.
    javax.jms.Queue queue =
        remoteSessionSender.createQueue(remoteQueueName + "Listener");
    QueueSender remoteListener =
        remoteSessionSender.createSender(queue);
    sendMessage(remoteSessionSender, remoteListener,
        "ControlMessage:Dial " + localQueueName);
    remoteListener.close();

    // Now wait for a response from the other phone on our receiver
    // Wait only for 15 seconds.
    queue =
        localSessionReceiver.createQueue(localQueueName + "Receiver");
    localReceiver = localSessionReceiver.createReceiver(queue);
    TextMessage msg = (TextMessage)localReceiver.receive(15000);
    localReceiver.close();

    // What did we get back
    if( msg == null )
        throw new PhoneException(
            "No answer. Line may have been disconnected.");
    else if( msg.getText().equals("ControlMessage:Busy") )
        throw new PhoneException("Line is Busy.");
    else if( !msg.getText().equals("ControlMessage:OK") )
        throw new PhoneException("Invalid Control Response received");

    // If we got here then the connection has been established.

    // Receive from remote's Sender.
    queue =
        remoteSessionReceiver.createQueue(remoteQueueName + "Sender");
    remoteReceiver = remoteSessionReceiver.createReceiver(queue);
    remoteReceiverThread = new ReceiverThread(this,remoteReceiver);
    remoteReceiverThread.start();

    // Send to remote's receiver
    queue =
        remoteSessionSender.createQueue(remoteQueueName + "Receiver");
    remoteSender = remoteSessionSender.createSender(queue);

    // Remember who we're talking to.
    this.remoteQueueName = remoteQueueName;
    // Phone is busy now.
    busy = true;
}

// This method is called by our listener thread

```

```

// when another phone tries to call us.
synchronized void dialReceived(String fromQueue)
    throws JMSEException, PhoneException
{
    debugText += ("Dial request received from " + fromQueue + "\n");
    repaint();

    // Depending on our status...
    // Send a "Busy" or "OK" message to the reomote queue's receiver
    javax.jms.Queue queue =
        localSessionSender.createQueue(fromQueue + "Receiver");
    QueueSender sender = localSessionSender.createSender(queue);
    if( isBusy() )
        sendMessage(localSessionSender, sender, "ControlMessage:Busy");
    else {
        sendMessage(localSessionSender, sender, "ControlMessage:OK");

        // remote will send on this queue i.e. our receiver
        queue =
            localSessionReceiver.createQueue(localQueueName+"Receiver");
        localReceiver = localSessionReceiver.createReceiver(queue);
        localReceiverThread = new ReceiverThread(this, localReceiver);
        localReceiverThread.start();

        // remote will receive on this queue i.e. our sender
        queue =
            localSessionSender.createQueue(localQueueName + "Sender");
        localSender = localSessionSender.createSender(queue);

        // We're busy now...
        busy = true;
    }
}

// This method is called to hangup i.e. terminate
// the current call from our side.
synchronized void hangup() throws JMSEException, PhoneException
{
    // Must be busy...
    if( !isBusy() )
        throw new PhoneException("Not dialed anywhere.");

    debugText += "Hanging up call.\n" ;

    // If we initiated the call then send a "Hangup"
    // message to the other phone on the other phone's
    // queuesender.
    if( remoteQueueName != null ) {
        sendMessage(remoteSessionSender, remoteSender,
            "ControlMessage:Hangup");
        remoteQueueName = null;
        remoteReceiverThread.stop();
        remoteReceiverThread = null;
        remoteSender.close();
        remoteSender = null;
        remoteReceiver.close();
        remoteReceiver = null;
    }
    // otherwise send a hangup message on our queuesender
    else {
        sendMessage(localSessionSender, localSender,
            "ControlMessage:Hangup");
        localReceiverThread.stop();
        localReceiverThread = null;
        localSender.close();
        localSender = null;
        localReceiver.close();
        localReceiver = null;
    }
}

```

```

    }

    // No longer busy.
    busy = false;
}

// The other side wants to terminate this call
synchronized void hangupReceived()
    throws JMSEException, PhoneException
{
    // Must be busy...
    if( !isBusy() )
        throw new PhoneException("Not dialed anywhere.");

    busy = false;

    debugText += "Hang up request received.\n";
    repaint();

    // Cleanup based on which side's lines are being used.
    if( remoteQueueName != null ) {
        remoteQueueName = null;
        remoteReceiverThread.stop();
        remoteReceiverThread = null;
        remoteSender.close();
        remoteSender = null;
        remoteReceiver.close();
        remoteReceiver = null;
    }
    else {
        localReceiverThread.stop();
        localReceiverThread = null;
        localSender.close();
        localSender = null;
        localReceiver.close();
        localReceiver = null;
    }
}

// Disconnect this phone line.
// This line will no longer be available.
synchronized void disconnect() {
    try {
        // If the phone is busy, hangup the call first
        if( isBusy() )
            hangup();
        // Now cleanup
        shutdown();
    }
    catch( Exception e ) {
        debugText += (e.getMessage() + "\n");
        repaint();
    }
}

// Send a "user" message to the other phone.
// This method calls the sendMessage method with
// the appropriate parameters.
synchronized void say(String message)
    throws JMSEException, PhoneException
{
    if( !isBusy() )
        throw new PhoneException("You must dial out first");
    if( remoteQueueName != null )
        sendMessage(remoteSessionSender, remoteSender, message);
    else
        sendMessage(localSessionSender, localSender, message);
}

```

```

// Send a message using the sender
void sendMessage(QueueSession session, QueueSender sender,
                String message) throws JMSEException
{
    debugText += ("Sending message " + message + " to queue " +
                 sender.getQueue().getQueueName() + "\n");
    repaint();

    // Create a text message, set its text to the message
    // The message is set to expire in 15 seconds.
    TextMessage msg = session.createTextMessage();
    msg.setText(message);
    sender.send(msg, DeliveryMode.NON_PERSISTENT, 9, 15000);
}

// Cleanup time...
private void shutdown() {
    try {
        localListenerThread.stop();
        localSessionSender.close();
        localSessionReceiver.close();
        localSessionListener.close();
        remoteSessionSender.close();
        remoteSessionReceiver.close();
        connection.close();
    }
    catch( Exception e ) {
        debugText += (e.getMessage() + "\n");
        repaint();
    }
}

// Show the output and information window.
private void showGUI() {
    setTitle("Phone [" + localQueueName +
            "] Output and Information Window");

    Panel messagePanel = new Panel();
    messagePanel.setLayout(new GridLayout(2, 1));
    messageAreaIn = new TextArea();
    messageAreaIn.setEditable(false);
    messagePanel.add(messageAreaIn);
    messageAreaDebug = new TextArea();
    messageAreaDebug.setEditable(false);
    messagePanel.add(messageAreaDebug);
    add(messagePanel, "Center");

    Dimension screenSize =
        Toolkit.getDefaultToolkit().getScreenSize();

    int HEIGHT = 350, WIDTH = 410;
    setLocation(screenSize.width/2 - WIDTH/2,
                screenSize.height/2 - HEIGHT/2);
    setSize(WIDTH, HEIGHT);
    setVisible(true);
}

public void paint(Graphics g) {
    super.paint(g);
    synchronized(this) {
        messageAreaDebug.setText(debugText);
        messageAreaIn.setText(inText);
    }
}
}

// This thread corresponds to the command console thread.

```

```

// It will run for the life of the application and allow
// the user to enter commands. The thread will then call
// the appropriate methods on the phone in response to
// those commands.
class MainThread extends Thread {
    private Phone phone = null;

    public MainThread(Phone p) {
        phone = p;
    }

    public void run() {
        System.out.println("Phone Command Console is ready...\n");

        byte[] bytes = new byte[1000];
        while( true ) {
            System.out.print("cmd>");

            int n = 0;
            try {
                n = System.in.read(bytes);
            }
            catch( Exception e ) {
            }

            if( n <= 2 )
                continue;

            String cmd = new String(bytes,0,n-2);
            cmd = cmd.trim();

            if( cmd.startsWith("dial") ) {
                try {
                    int i = 4;
                    for( ;i<cmd.length(); i++ )
                        if( cmd.charAt(i) != ' ' )
                            break;

                    if( i == cmd.length() )
                        System.out.println("You must specify a destination.");
                    else
                        phone.dial(cmd.substring(i));
                }
                catch( Exception e ) {
                    System.err.println(e.getMessage());
                }
            }
            else if( cmd.startsWith("disconnect") ) {
                try {
                    phone.disconnect();
                }
                catch( Exception e ) {
                    System.err.println(e.getMessage());
                }
                System.exit(-1);
            }
            else if( cmd.startsWith("hangup") ) {
                try {
                    phone.hangup();
                }
                catch( Exception e ) {
                    System.err.println(e.getMessage());
                }
            }
            else if( cmd.startsWith("help") || cmd.startsWith("?") ) {
                System.out.println("Supported commands are: dial " +
                    "<destination>, disconnect, hangup, help," +
                    " and say <message>");
            }
        }
    }
}

```

```

    }
    else if( cmd.startsWith("say") ) {
        try {
            int i = 3;
            for( ;i<cmd.length(); i++ )
                if( cmd.charAt(i) != ' ' )
                    break;

            if( i == cmd.length() )
                System.out.println(
                    "You must specifiy a message.");
            else
                phone.say(cmd.substring(i));
        }
        catch( Exception e ) {
            System.err.println(e.getMessage());
        }
    }
    else {
        System.out.println("Unknown command.\n" +
            "Type help for a list of valid commands.");
    }
}
}
}

// This thread listens for incoming calls.
class ListenerThread extends Thread {
    private Phone phone = null;
    private QueueReceiver receiver = null;

    // The constructor gets the receiver that is the "listener"
    public ListenerThread(Phone p, QueueReceiver r) {
        phone = p;
        receiver = r;
    }

    public void run() {
        while( true ) {
            try {
                // Block till a message arrives
                TextMessage msg = (TextMessage)receiver.receive();

                // Is this a "Dial" message?
                if( msg.getText().startsWith("ControlMessage:Dial") ) {

                    // Find the phone that dialed us...
                    String cmd = msg.getText().trim();
                    int i = new String("ControlMessage:Dial").length();
                    for( ;i<cmd.length(); i++ )
                        if( cmd.charAt(i) != ' ' )
                            break;

                    if( i == cmd.length() ) {
                        phone.debugText +=
                            "Received Dial command without a destination.\n";
                        phone.repaint();
                    }
                    else
                        // and call the dialReceived method with
                        // this phone name.
                        phone.dialReceived(cmd.substring(i));
                }
                else {
                    phone.debugText +=
                        ("[ListenerThread] Invalid Control Message \"" +
                            msg.getText() + "\" received.\n");
                    phone.repaint();
                }
            }
            catch (Exception e) {
                // ignore
            }
        }
    }
}

```

```

        }
    }
    catch( Exception e ) {
        synchronized(phone) {
            phone.debugText += (e.getMessage() + "\n");
            phone.repaint();
        }
    }
}

// This thread is used to receive messages
// from a receiver.
class ReceiverThread extends Thread {
    private Phone phone = null;
    private QueueReceiver receiver = null;

    // The constructor gets the receiver to receive from.
    public ReceiverThread(Phone p, QueueReceiver r) {
        phone = p;
        receiver = r;
    }

    public void run() {
        while( true ) {
            try {
                // Block till a message arrives. We know it will be a
                // text message.
                TextMessage msg = (TextMessage)receiver.receive();

                // Is this a "Hangup" message?
                if( msg.getText().equals("ControlMessage:Hangup") )
                    phone.hangupReceived();
                // If not then it is a "user" message. Show it.
                else {
                    if( phone.isBusy() ) {
                        synchronized(phone) {
                            phone.inText += (msg.getText() + "\n");
                            phone.repaint();
                        }
                    }
                }
            }
            catch( Exception e ) {
                synchronized(phone) {
                    phone.debugText += (e.getMessage() + "\n");
                    phone.repaint();
                }
            }
        }
    }
}

// Nothing too special about this exception.
class PhoneException extends Exception {
    public PhoneException(String message) {
        super(message);
    }
}

```

Publish-And-Subscribe Messaging

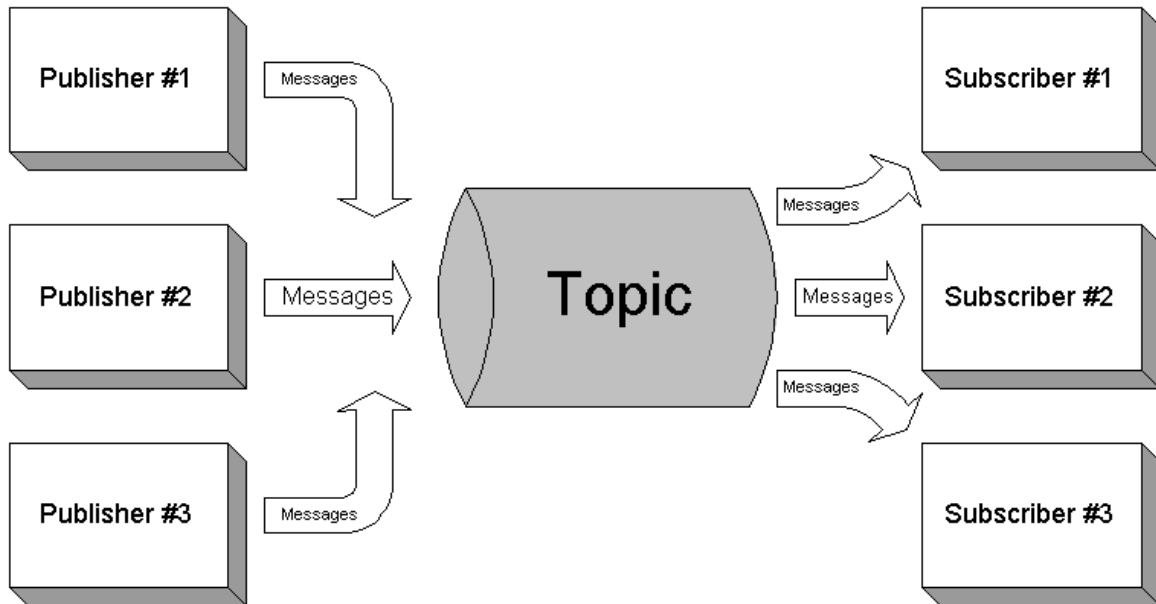


Figure 4: Publish-and-Subscribe Messaging Style

As shown in figure 4, in this messaging style multiple applications connect to the messaging product as either publishers i.e. producers or subscribers i.e. consumers of messages.

Following are the basic components/pieces of JMS involved in this style:

- The Destination is known as Topic.
- The ConnectionFactory is known as a TopicConnectionFactory.
- The Connection is known as TopicConnection.
- The Session is known as a TopicSession
- The MessageProducer is known as a TopicPublisher.
- The MessageConsumer is known as a TopicSubscriber.

Note the use of the word *Topic* in every piece of JMS associated with this style.

In the following sections I'll discuss each one of the basic pieces involved in the publish-and-subscribe messaging style. Let's start with the TopicConnectionFactory.

TopicConnectionFactory

As shown in figure 2 in chapter 3, the TopicConnectionFactory interface is actually derived from the ConnectionFactory interface. The TopicConnectionFactory interface is shown below:

```
public interface TopicConnectionFactory extends ConnectionFactory {
    TopicConnection createTopicConnection() throws JMSEException;
    TopicConnection createTopicConnection(String userName,
        String password) throws JMSEException;
}
```


As its name implies, the topic connection factory is used for creating topic connections. The `createTopicConnection` method instructs the JMS provider to create a topic connection with default user identity. The exact meaning of the default user identity is provider specific. The other version of the `createTopicConnection` method takes a username and password and instructs the JMS provider to create a topic connection with these user credentials. This version of the method may throw a `JMSSecurityException` if client authentication fails due to an invalid username or password. Both methods throw a `JMSEException` if the topic connection cannot be created.

Like its parent, the topic connection factory is an administrable object as well. JMS does not define any facilities for creating, administering, or deleting topic connection factories. These facilities are provided by the JMS provider and hence are provider specific.

TopicConnection

As discussed above a topic connection is obtained through a topic connection factory. Figure 3 in chapter 3 showed that the `TopicConnection` interface derives from the `Connection` interface. The `TopicConnection` interface is shown below:

```
public interface TopicConnection extends Connection {
    TopicSession createTopicSession(boolean transacted,
        int acknowledgeMode) throws JMSEException;

    ConnectionConsumer createConnectionConsumer(Topic topic,
        String messageSelector, ServerSessionPool sessionPool,
        int maxMessages) throws JMSEException;

    ConnectionConsumer createDurableConnectionConsumer(
        Topic topic, String subscriptionName,
        String messageSelector, ServerSessionPool sessionPool,
        int maxMessages) throws JMSEException;
}
```

Just as the topic connection factory is a factory for creating topic connections, the topic connection is a factory for creating topic sessions through the `createTopicSession` method. This method takes two parameters; the first parameter indicates whether the session is transacted and the second parameter specifies the message acknowledgement mode. I've discussed both of these in detail in chapters 2 and 3. This method will throw a `JMSEException` if the connection fails to create a session due to some internal error or a lack of support for specific transaction and/or acknowledgement mode.

This interface also has two additional methods `createConnectionConsumer` and `createDurableConnectionConsumer`. The JMS specification has flagged these methods as optional methods that providers may choose not to implement and further states that these are an expert facility that is not used by regular clients. Most providers do not implement these methods and so I will not discuss these methods in this book.

TopicSession

As shown in figure 4 in chapter 3, the `TopicSession` interface is actually derived from the `Session` interface. The `TopicSession` interface is shown below:

```
public interface TopicSession extends Session {
    Topic createTopic(String topicName) throws JMSEException;
    TopicSubscriber createSubscriber(Topic topic)
        throws JMSEException;
    TopicSubscriber createSubscriber(Topic topic,
        String messageSelector, boolean noLocal) throws JMSEException;

    TopicSubscriber createDurableSubscriber(Topic topic,
```

```

        String name) throws JMSEException;
    TopicSubscriber createDurableSubscriber(Topic topic,
        String name, String messageSelector, boolean noLocal)
        throws JMSEException;

    TopicPublisher createPublisher(Topic topic) throws JMSEException;

    TemporaryTopic createTemporaryTopic() throws JMSEException;
    void unsubscribe(String name) throws JMSEException;
}

```

The `createTopic` method is somewhat misleading. The JMS specification states that this method is provided "for the rare cases where clients need to dynamically manipulate topic identity. This allows the creation of a topic identity with a provider specific name. **Clients that depend on this ability are not portable.**"

Note – createTopic caveat

Different providers have slightly different behaviors for this method. For example, Sun's Java Message Queue actually allows the creation of the topic (as the name implies). Most JMS compliant messaging products require the use of their administration facilities to actually create the topic beforehand. This method in those providers merely lets you get a reference to the topic to pass into other methods, such as `createSubscriber` and `createPublisher`.

The `createPublisher` method is used to create a publisher for the specified topic. If the publisher cannot be created for any reason, a `JMSEException` will be thrown. I'll come back to topic publishers later in this chapter.

The JMS publish-and-subscribe style supports two types of subscribers: Durable and Non-durable. Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client will only see the messages published on a topic while its subscriber is active. If the subscriber is not active, the client will miss the published messages. Durable subscriptions, on the other hand, last beyond the lifetime of the subscriber. These subscriptions are tied not to the object but to the object identity. A later [different] subscriber with the same identity may resume subscription in the state left off by the previous subscriber with that identity. A JMS provider is required to retain a durable subscription's messages until they are either received or they expire. It therefore follows that durable subscriptions are more "expensive" than non-durable subscriptions.

How durable is Durable?

If a message sent to a durable subscription using non-persistent delivery mode, the message may be lost if the subscriber is not active at the time the message is published. This is because JMS does only provides "at most once" guarantee with non-persistent delivery, which means the message may not be delivered at all. So, if you're using Durable subscriptions, consider sending the messages in persistent delivery mode.

The `TopicSession` interface provides methods to create both durable and non-durable subscribers. Durable subscriptions are created using the `createDurableSubscriber` pair of methods. Similarly, non-durable subscriptions are created using the `createSubscriber` pair of methods. The first version of both these methods takes a reference to a topic object. The second version of both these methods take two additional parameters – a message selector string, `messageSelector`, and a boolean, `noLocal`. Message selectors were discussed in detail in chapter 5. By specifying a message selector, only those messages whose header meet the criteria specified by the selector will be delivered to the subscriber. The boolean `noLocal` parameter is more interesting at this time. If the value of this parameter is true, then messages published by a publisher will not be delivered to subscribers of the same connection. Confused? Let's look at the following code fragment to clarify things.

```

// Define a message selector
String someSelector = ?;

// topicConn1 is a valid TopicConnection
// create a session using topicConn1.
TopicSession topicSession1 =
    topicConn1.createTopicSession(false,1);
// create a topic using topicSession1.
Topic topic = topicSession1.createTopic("MyTopic");
// create a publisher using topicSession1.
TopicPublisher topicPub = topicSession1.createPublisher(topic);
// create a subscriber using topicSession1.
TopicSubscriber topicSub1 =
    topicSession1.createSubscriber(topic,someSelector,true);

// topicConn2 is a valid TopicConnection
// create a session using topicConn2.
TopicSession topicSession2 =
    topicConn2.createTopicSession(false,1);
// create a topic using topicSession2.
topic = topicSession2.createTopic("MyTopic");
// create a subscriber using topicSession2.
TopicSubscriber topicSub2 =
    topicSession2.createSubscriber(topic,someSelector,true);

// create and publish a message that satisfies someSelector
Message msg = ?;
topicPub.publish(msg);

```

In the above code fragment the value of the `noLocal` boolean parameter in the `createSubscriber` methods used for creating `topicSub1` and `topicSub2` is specified as `true`. As a result, if both subscribers `topicSub1` and `topicSub2` were listening for messages at the point `topicPub` published the message, `topicSub1` would not receive the message since both `topicSub1` and `topicPub` are created through the same topic connection.

The `createTemporaryTopic` method on the `TopicSession` interface is used to create a temporary topic, which we'll discuss in the next section.

Finally, the `unsubscribe` method is used to remove a durable subscription that has been created by a client. Invoking this method will instruct the JMS provider to delete all the state being maintained on behalf of the durable subscriber by its provider. It is erroneous for a client to delete a durable subscription while it has an active subscriber for it, or while a message received by it is part of a transaction or has not been acknowledged in the session. This method will throw an `InvalidDestinationException` if a non-existent/invalid subscription name is specified.

Topic

As shown in figure 1 in chapter 3, the `Topic` interface is actually derived from the `Destination` interface. The `Topic` interface is shown below:

```

public interface Topic extends Destination {
    String getTopicName() throws JMSEException;
    String toString();
}

```

The `getTopicName` method returns the name of the topic as used by the messaging product while the `toString` method returns a "prettier" version of this name. As a general rule, clients should not depend on the name returned by these methods in their code, since these names may be provider specific and may not be the same as the name defined by the topic administrator.

Like its parent, the topic object is an administrable object as well. There are two types of topics: long lived and temporary. JMS does not define any facilities for creating, administering, or deleting long lived topics. These facilities are provided by the JMS provider and hence are provider specific. Remember, JMS does define a standard way to gain access to a long lived topic through the topic session via the `createTopic` method that we saw above. Temporary topics, on the other hand can be created programmatically through the topic session via the `createTemporaryTopic` method. Such topics are valid only with [and for the life of] the topic connection to which the topic session used to create the temporary queue belonged. Temporary topics implement the `TemporaryTopic` interface shown below:

```
public interface TemporaryTopic extends Topic {
    void delete() throws JMSEException;
}
```

The `delete` method is used to delete the temporary topic. If there are existing publishers or subscribers still using it, then a `JMSEException` will be thrown. It is good programming practise to call the `delete` method on a temporary topic when it is no longer required to free up any resources consumed by the topic.

TopicSubscriber

Per the JMS specification, a topic session allows the creation of multiple topic subscribers per topic and will deliver each message for that topic to each topic subscriber eligible to receive it. Each copy of the message is treated as a completely separate message so work done on one copy has no affect on the other; acknowledging one does not acknowledge the other; one message may be delivered immediately while another waits for its consumer to process messages ahead of it and so on.

As shown in figure 5 in chapter 3, the `TopicSubscriber` interface is actually derived from the `MessageConsumer` interface. The `TopicSubscriber` interface is shown below:

```
public interface TopicSubscriber extends MessageConsumer {
    Topic getTopic() throws JMSEException;
    boolean getNoLocal() throws JMSEException;
}
```

The `getTopic` method returns a reference to the topic object specified in the `create[Durable]Subscriber` method of the topic session. This may not be a reference to the exact same physical object. Consider the following code fragment:

```
// topic is a valid Topic
// topicSession is a valid TopicSession
subscriber = topicSession.createSubscriber(topic);
topic1 = subscriber.getQueue();
if(topic == topic1)
    System.out.println("Same physical object.");
else
    System.out.println("Not the same physical object.");
```

Depending on the provider any one of the two statements may be printed. But the statement

```
topic.equals(topic1)
```

should always evaluate to `true`.

The `getNoLocal` method returns a `true` value if local messages are being inhibited. By default this value is `false`. Remember, this value can be set while creating the subscriber with the second version of the `createSubscriber` and `createDurableSubscriber` methods descibed in the `TopicSession` interface above.

TopicPublisher

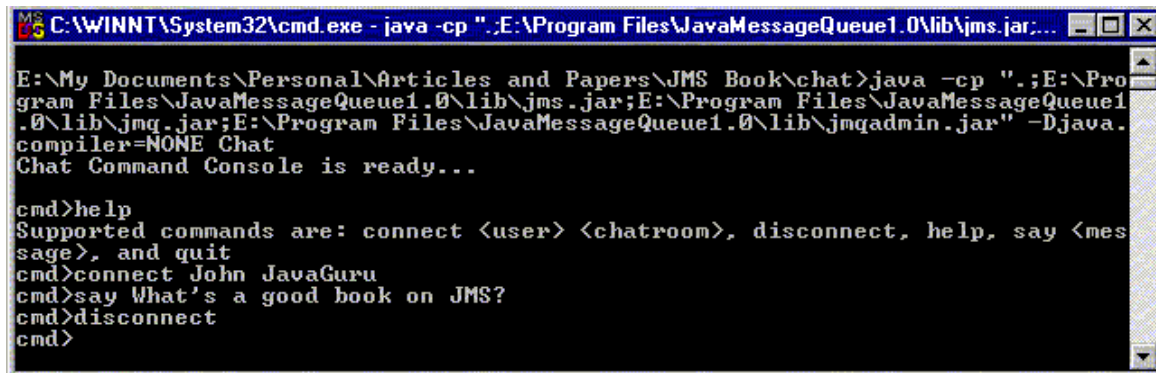
As shown in figure 6 in chapter 3, the `TopicPublisher` interface is actually derived from the `MessageProducer` interface. The `TopicPublisher` interface is shown below:

```
public interface TopicPublisher extends MessageProducer {
    Topic getTopic() throws JMSEException;
    void publish(Message message) throws JMSEException;
    void publish(Message message, int deliveryMode, int priority,
        long timeToLive) throws JMSEException;
    void publish(Topic topic, Message message) throws JMSEException;
    void publish(Topic topic, Message message, int deliveryMode,
        int priority, long timeToLive) throws JMSEException;
}
```

The `getTopic` method returns a reference to the topic object specified in the `createPublisher` method of the topic session. This may not be a reference to the exact same physical object. I discussed this above with an example code fragment. The interface has a variety of `publish` methods. One of the first two `publish` methods are used when a topic is specified during the creation of the publisher using the `createPublisher` method of the topic session object. The first method uses the default values for the delivery mode, priority, and time-to-live parameters and the second version allows you to specify values for these. The last two `publish` methods are used when a topic is not specified during the publisher creation i.e. a `null` value is passed in to the `createPublisher` method. The first parameter to these two methods is a valid topic to which to send the message. Note that *all four publish methods can never be used in the same sender*. If a topic was specified during the creation of the publisher, then attempting to use the last two `publish` methods will throw a `UnsupportedOperationException`. On the other hand if no topic was specified, it is quite obvious that the first two `publish` methods cannot be used. All `publish` methods will throw a `MessageFormatException` if there is a formatting error in the message, an `InvalidDestinationException` if the topic is invalid, and a `JMSEException` if there is any other error during the publish operation.

An Example

Let's tie all these concepts together with an example program. This will be a Chat program and we will continue to use Sun's Java Message Queue.



```
C:\WINNT\System32\cmd.exe - java -cp ".;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar;...
E:\My Documents\Personal\Articles and Papers\JMS Book\chat>java -cp ".;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar;E:\Program Files\JavaMessageQueue1.0\lib\jmq.jar;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar" -Djava.compiler=NONE Chat
Chat Command Console is ready...

cmd>help
Supported commands are: connect <user> <chatroom>, disconnect, help, say <message>, and quit
cmd>connect John JavaGuru
cmd>say What's a good book on JMS?
cmd>disconnect
cmd>
```

Figure 5: The Chat Command Console window (after executing a few commands)

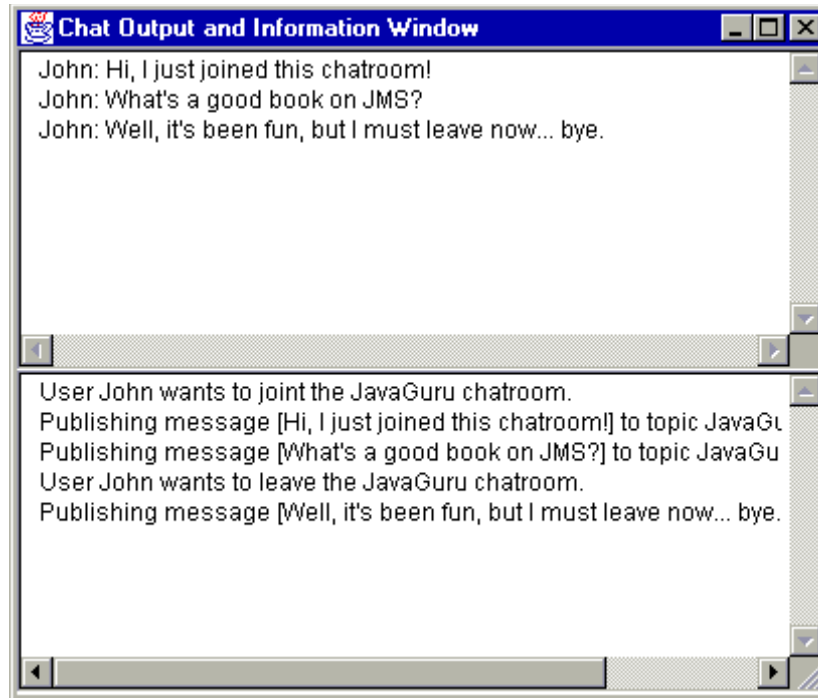


Figure 6: The Chat Output and Information window
(after executing a few commands at the command console window)

First let me show you how to start the chat program.

```
REM Set up the classpath. My installation of Sun's Java Message Queue
REM is in the directory E:\Program Files\JavaMessageQueue1.0
set CPATH=.;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmq.jar;
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar

java -cp "%CPATH%" -Djava.compiler=NONE Chat
```

This will result in the creation of a command console, which the same window/shell from which the program was started and an "Output and Information" GUI window. These windows are shown in figures 5 and 6 respectively. These windows have a similar purpose to those in the phone program discussed earlier. Once again, the command console is where all the action takes place. To get a list of all supported commands, type in `help` at the prompt. At this point you should see the following output:

```
connect <user> <chatroom> , disconnect, help, say <message>, and quit
```

Initially, the only valid commands are `connect` and `quit` (and `help` of course). Typing in `quit` ends the chat program. This is the only way to cleanly shutdown the chat program. To join a "chatroom" type in `connect` followed by a username for that chatroom and the name of the chatroom. For example:

```
connect John JavaGuru
```

, where "JavaGuru" is an existing chatroom. I'll discuss what "existing" means later.

The only valid commands at this point are `say`, `disconnect`, and `quit` (and `help` of course). To send a message to the chatroom you use the `say` command, as follows:

```
say What's a good book on JMS?
```

To leave the chatroom at any point use the `disconnect` command.

So, how does all this work under the covers?

The main Method

Let's start by looking at the `main` method of the program. This method creates a new instance of `Chat` and starts an instance of the class `MainThread` as follows:

```
// Create the chat object
Chat chat = new Chat();
// Start the command console.
MainThread mainThread = new MainThread(chat);
mainThread.start();
```

The `MainThread` class extends the `java.lang.Thread` class. This class encapsulates the command console functionality and is similar in functionality to the `MainThread` class discussed in the phone example except that it processes a different command set.

The Chat Constructor

The constructor starts off by calling the `showGUI` method to create and display the output and information window. It then gets a reference to a topic connection factory as follows:

```
// Get the topic connection factory.
connectionFactory = new com.sun.messaging.TopicConnectionFactory();
```

This is the only Sun specific code in this program. As I've mentioned several times before, in chapter 8 I will discuss a way to not have any JMS provider specific code in your JMS applications/clients.

The connection factory is then used to create a connection to the provider, which is then used to create two separate sessions: one for the publisher and one for the subscriber. As we discussed in chapter 4, JMS session objects are single-threaded. To allow concurrent publication and subscription of messages, we must create separate sessions. Finally, after all the setup work is complete, the connection is started.

```
// Use the factory to create a topic connection
connection = connectionFactory.createTopicConnection();
// Create two separate sessions -- one for the subscriber
// and one for the publisher.
sessionForPublisher =
connection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
sessionForSubscriber =
connection.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);
// setup complete, start the connection.
connection.start();
```

Note

In the phone example, I achieved concurrent receives and sends by creating multiple threads i.e. I used a separate thread to receive messages from a queue and a separate thread to send messages to the queue. In this example, I will show you another way to accomplish this – via message listeners. Instead of creating a separate thread for message subscriptions, I will install a message listener. Internally, the JMS provider does create another thread to service the listener, but at least the client code is cleaner.

The connect method

Next, let's take a look at one of the key methods of the Chat class – the `connect` method. This method is called by the command console when the user types in the `connect` command. This method takes two parameters: a username and the name of a chatroom. The username is only used to associate the user with the messages he sends and is not used for any security/authentication purposes. Earlier I had stated that the name of chatroom must be an "existing" chatroom. This is because the name of the chatroom is used as the name of the topic in the `createTopic` method call on the topic session as shown below:

```
// topicname is the chatroom
topic = sessionForPublisher.createTopic(topicname);
```

Note

A topic corresponding to each chatroom name to be supported must be setup using the admin features of the JMS provider. For example, to support a chatroom "JavaGuru", a topic called "JavaGuru" must be setup. However, Sun's provider automatically creates such a topic when the `createTopic` method is called on the topic session. So no special work is required. See the "createTopic caveat" sidebar.

The `connect` method makes sure that no other "chats" are in progress by calling the private `isConnected` method. It then creates a topic as shown above and creates a publisher for this topic as follows:

To create a subscriber for this topic, we must recreate the topic using the session object for the subscriber. This is shown below:

```
// Recreate the topic and then create a subscriber
topic = sessionForSubscriber.createTopic(topicname);
subscriber = sessionForSubscriber.createSubscriber(topic);
```

Note that I do not create a durable subscription since I do not want/need messages to be stored when the chat client is not active. Next, I set the message listener on the subscriber as follows:

```
// Set the message listener on the subscriber.
subscriber.setMessageListener(this);
```

The message listener is the chat object itself. The Chat class implements the `MessageListener` interface to allow this. Remember, this interface has one method: `onMessage`. The crux of this method's implementation is as follows:

```
// Cast the message as a text message.
// Show the contents in the output panel.
TextMessage textMessage = (TextMessage) aMessage;
inText += (textMessage.getText() + "\n");
.
.
.
repaint();
```


Also, note that I explicitly catch all runtime exceptions as all good message listeners should. I discussed this in chapter 4.

The sendMessage method

This method is called by the command console in response to the `say` command. This method is also called by the `connect` and `disconnect` methods. It takes the message to publish as its only parameter. This gist of this method is as follows:

```
// Create a TextMessage, set its text content and publish it.
TextMessage msg = sessionForPublisher.createTextMessage();
msg.setText(userName + ": " + message);
publisher.publish(msg);
```

The disconnect method

This method is called by the command console in response to the `disconnect` command. Note that this is not the same as the `disconnect` method in the phone example in which the phone example was terminated i.e. the phone was disconnected. That corresponds to the `shutdown` method (`quit` command) in this case. The `disconnect` method here simply undoes what the `connect` method did earlier. In that regard it is similar to the `hangup` method in the phone example.

The shutdown method

This method is called to end the chat example when a user enters the `quit` command at the command console. If necessary, this method will call the `disconnect` method to stop an ongoing chat. It then closes both topic sessions and the connection as follows:

```
if( isConnected() )
    disconnect();
// cleanup.
sessionForPublisher.close();
sessionForSubscriber.close();
connection.close();
```

Here's the complete listing of the Chat program

```
import java.awt.*;
import java.awt.event.*;
import javax.jms.*;

public class Chat extends Frame implements MessageListener {

    // Topic related stuff.
    private String topicName = null;
    private String userName = null;
    private TopicConnectionFactory connectionFactory = null;
    private TopicConnection connection;
    private Topic topic = null;
    private TopicSession sessionForPublisher = null;
    private TopicSession sessionForSubscriber = null;
    private TopicPublisher publisher = null;
    private TopicSubscriber subscriber = null;

    // member variables for GUI.
    private TextArea messageAreaIn;
    private TextArea messageAreaDebug;
    String debugText = "";
    String inText = "";

    public static void main(String args[]) {
        // Create the chat object
        Chat chat = new Chat();
        // Start the command console.
```

```

    MainThread mainThread = new MainThread(chat);
    mainThread.start();
}

public Chat() {
    // Show the GUI "Output and Information" window
    showGUI();
    try {
        // Get the topic connection factory.
        connectionFactory =
            new com.sun.messaging.TopicConnectionFactory();
        // Use the factory to create a topic connection
        connection = connectionFactory.createTopicConnection();
        // Create two separate sessions -- one for the subscriber
        // and one for the publisher.
        sessionForPublisher =
            connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
        sessionForSubscriber =
            connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

        // setup complete, start the connection.
        connection.start();
    }
    catch( JMSEException e ) {
        debugText += (e.getMessage() + "\n");
        repaint();
    }
}

// MessageListener interface implementation
public void onMessage(Message aMessage) {
    try {
        // Cast the message as a text message.
        // Show the contents in the output panel.
        TextMessage textMessage = (TextMessage) aMessage;
        inText += (textMessage.getText() + "\n");
    }
    catch( JMSEException e ) {
        debugText += (e.getMessage() + "\n");
    }
    // catch all runtime exceptions like a good message listener.
    catch( java.lang.RuntimeException e ) {
        debugText += (e.getMessage() + "\n");
    }
    finally {
        repaint();
    }
}

// shut down...
synchronized void shutdown() {
    try {
        // if we're connected - disconnect
        if( isConnected() )
            disconnect();
        // cleanup.
        sessionForPublisher.close();
        sessionForSubscriber.close();
        connection.close();
    }
    catch( Exception e ) {
        debugText += (e.getMessage() + "\n");
        repaint();
    }
}

synchronized void connect(String username, String topicname)
    throws JMSEException, ChatException

```

```

{
    // only one chat room at a time.
    if( isConnected() )
        throw new ChatException("Already connected as " + userName +
            " to chatroom " + topicName);

    debugText += ("User " + username + " wants to joint the " +
        topicname + " chatroom.\n");
    repaint();

    // Create a publisher and subscriber using the
    // appropriate sessions.
    topic = sessionForPublisher.createTopic(topicname);
    publisher = sessionForPublisher.createPublisher(topic);
    topic = sessionForSubscriber.createTopic(topicname);
    subscriber = sessionForSubscriber.createSubscriber(topic);

    // Set the message listener on the subscriber.
    subscriber.setMessageListener(this);

    userName = username;
    topicName = topicname;

    // first message
    sendMessage("Hi, I just joined this chatroom!");
}

// The disconnect method. Called to disconnect from the
// chat room
synchronized void disconnect() throws JMSEException, ChatException
{
    // must be connected to disconnect.
    if( !isConnected() )
        throw new ChatException("Not connected to any chatroom.");

    debugText += ("User " + userName + " wants to leave the " +
        topicName + " chatroom.\n");
    repaint();

    // last message
    sendMessage("Well, it's been fun, but I must leave now... bye.");

    // cleanup.
    subscriber.close();
    publisher.close();
    userName = null;
    topicName = null;
}

// Send a message to the chatroom
synchronized void sendMessage(String message)
    throws JMSEException, ChatException
{
    if( !isConnected() )
        throw new ChatException("Not connected to any chatroom.");

    debugText += ("Publishing message [" + message + "] to topic " +
        publisher.getTopic().getTopicName() + "\n");
    repaint();

    // Create a TextMessage, set its text content and publish it.
    TextMessage msg = sessionForPublisher.createTextMessage();
    msg.setText(userName + ": " + message);
    publisher.publish(msg);
}

// Are we connected to a chatroom?
private boolean isConnected() {

```

```

    return(topicName != null && userName != null);
}

// Show the output and information window.
private void showGUI() {
    setTitle("Chat Output and Information Window");

    Panel messagePanel = new Panel();
    messagePanel.setLayout(new GridLayout(2, 1));
    messageAreaIn = new TextArea();
    messageAreaIn.setEditable(false);
    messagePanel.add(messageAreaIn);
    messageAreaDebug = new TextArea();
    messageAreaDebug.setEditable(false);
    messagePanel.add(messageAreaDebug);
    add(messagePanel, "Center");

    Dimension screenSize =
        Toolkit.getDefaultToolkit().getScreenSize();

    int HEIGHT = 350, WIDTH = 410;
    setLocation(screenSize.width/2 -
                WIDTH/2, screenSize.height/2 - HEIGHT/2);
    setSize(WIDTH, HEIGHT);
    setVisible(true);
}

public void paint(Graphics g) {
    super.paint(g);
    synchronized(this) {
        messageAreaDebug.setText(debugText);
        messageAreaIn.setText(inText);
    }
}
}

// This thread corresponds to the command console thread.
// It will run for the life of the application and allow
// the user to enter commands. The thread will then call
// the appropriate methods on the phone in response to
// those commands.
class MainThread extends Thread {
    private Chat chat = null;

    public MainThread(Chat p) {
        chat = p;
    }

    public void run() {
        System.out.println("Chat Command Console is ready...\n");

        byte[] bytes = new byte[1000];
        while( true ) {
            System.out.print("cmd>");

            int n = 0;
            try {
                n = System.in.read(bytes);
            }
            catch( Exception e ) {
            }

            if( n <= 2 )
                continue;

            // Got a command...
            String cmd = new String(bytes,0,n-2);
            cmd = cmd.trim();

```

```

// Which command?
// connect.
if( cmd.startsWith("connect") ) {
    try {

        // Get the username
        // Where does it start?
        int i = 7;
        for( ;i<cmd.length(); i++ )
            if( cmd.charAt(i) != ' ' )
                break;
        if( i == cmd.length() ) {
            System.out.println("You must specifiy a user" +
                " and chatroom.");
            continue;
        }

        // Where does it end?
        int j = i;
        for( ;j<cmd.length(); j++ )
            if( cmd.charAt(j) == ' ' )
                break;
        if( j == cmd.length() ) {
            System.out.println("You must specifiy a user" +
                " and chatroom.");
            continue;
        }

        // user starts at i and ends at j.
        String user = cmd.substring(i,j);

        // Now get the chat room name
        // Where does it start?
        for( i=j;i<cmd.length(); i++ )
            if( cmd.charAt(i) != ' ' )
                break;
        if( i == cmd.length() ) {
            System.out.println("You must specifiy a user" +
                " and chatroom.");
            continue;
        }

        // Where does it end?
        j = i;
        for( ;j<cmd.length(); j++ )
            if( cmd.charAt(j) == ' ' )
                break;
        if( j == i ) {
            System.out.println("You must specifiy a user" +
                " and chatroom.");
            continue;
        }

        // chatroom starts at i and ends at j.
        String chatroom = cmd.substring(i,j);

        // connect...
        chat.connect(user, chatroom);
    }
    catch( Exception e ) {
        System.err.println(e.getMessage());
    }
}
// Process the disconnect command.
else if( cmd.startsWith("disconnect") ) {
    try {
        chat.disconnect();
    }
}

```

```

    }
    catch( Exception e ) {
        System.err.println(e.getMessage());
    }
}
// Process the help command
else if( cmd.startsWith("help") || cmd.startsWith("?") ) {
    System.out.println("Supported commands are: connect <user>" +
        "<chatroom>, disconnect, help, say <message>, and quit");
}
// Process the say command
else if( cmd.startsWith("say") ) {
    try {
        int i = 3;
        for( ;i<cmd.length(); i++ )
            if( cmd.charAt(i) != ' ' )
                break;

        if( i == cmd.length() )
            System.out.println("You must specify a message.");
        else
            chat.sendMessage(cmd.substring(i));
    }
    catch( Exception e ) {
        System.err.println(e.getMessage());
    }
}
// process the quit command
else if( cmd.startsWith("quit") ) {
    chat.shutdown();
    System.exit(0);
}
else {
    System.out.println("Unknown command.\n" +
        "Type help for a list of valid commands.");
}
}
}
}

// Nothing too special about this.
class ChatException extends Exception {
    public ChatException(String message) {
        super(message);
    }
}
}

```

Request/Reply Messaging

Let's go back to the phone example discussed earlier in this chapter. More specifically, let's return to the details of the "Dial" message and protocol. The "Dial" message includes the name of the user/line to send the response back to. This is essentially what JMS Request/Reply feature allows you to do. So, in this section I will show you how to use the JMS Request/Reply feature to accomplish the same results. Instead of going through the entire phone example again, I will explain the changes made to use the Request/Reply feature.

Let's start with the changes made to the dial method.

Changes to the dial method

In chapter 5, we saw that JMS provides the `JMSReplyTo` message header field for specifying the destination where a reply to a message should be sent. The dial method will change to reflect the use of this message header field.

Now, instead of:

```
debugText += ("Dialing out to " + remoteQueueName + "\n");
repaint();
// Send a "Dial" message to the other phone's listener.
javax.jms.Queue queue =
    remoteSessionSender.createQueue(remoteQueueName + "Listener");
QueueSender remoteListener =
    remoteSessionSender.createSender(queue);
sendMessage(remoteSessionSender, remoteListener,
    "ControlMessage:Dial " + localQueueName);
remoteListener.close();
```

the dial method has,

```
sendDialMessage(remoteQueueName);
```

Let's take a look at the sendDialMessage method, which is defined as follows:

```
private void sendDialMessage(String remoteQueueName)
    throws JMSEException {
    debugText += ("Dialing out to " + remoteQueueName + "\n");
    repaint();
    javax.jms.Queue queue =
        remoteSessionSender.createQueue(remoteQueueName +
            "Listener");
    QueueSender remoteListener =
        remoteSessionSender.createSender(queue);
    queue = null;

    debugText += ("Sending message ControlMessage:Dial to queue "
        + remoteListener.getQueue().getQueueName() + "\n");
    repaint();

    TextMessage msg = remoteSessionSender.createTextMessage();
    msg.setJMSReplyTo(remoteSessionSender.createQueue(
        localQueueName + "Receiver"));
    msg.setText("ControlMessage:Dial");
    remoteListener.send(msg, DeliveryMode.NON_PERSISTENT, 9, 15000);
    remoteListener.close();
}
```

The major lines of code are shown in bold. After creating the message, I set the JMSReplyTo header field by calling the setJMSReplyTo method on the message. Going back to our earlier example, if "John" was dialing out to "Bob" then the JMSReplyTo header field value would be set to the queue with the name "JohnReceiver". Remember, this is the queue where John expects the response from "Bob". The "Dial" message sent to "Bob" now becomes "ControlMessage:Dial" instead of "ControlMessage:Dial John".

Changes to the ListenerThread

So, what does "Bob" do in the the Request/Reply scenario? Let's take a look at the listener thread for part of the answer.

Instead of

```
// Find the phone that dialed us...
String cmd = msg.getText().trim();
int i = new String("ControlMessage:Dial").length();
for( ;i<cmd.length(); i++ )
    if( cmd.charAt(i) != ' ' )
        break;

if( i == cmd.length() ) {
```

```

        phone.debugText +=
            "Received Dial command without a destination.\n";
    }
    else
        // and call the dialReceived method with this phone name.
        phone.dialReceived(cmd.substring(i));

```

the listener thread now has,

```

    javax.jms.Queue queue = (javax.jms.Queue)msg.getJMSReplyTo();
    if(queue == null)
        phone.debugText +=
            "Received Dial command without a destination.\n";
    else
        phone.dialReceived(queue);

```

Earlier, the listener thread had to parse through the "Dial" message to find the name of the user/line dialing in. Now, that's no longer required, since it can get a reference to the queue to send the response to by calling the `getJMSReplyTo` method on the message. Instead of passing the name of the user/line to the `dialReceived` method, the listener thread now passes the queue reference to it. This requires a change to the `dialReceived` method.

Changes to the dialReceived method

This method now receives a reference to a queue to send the response. I added the following code to the beginning of the method to extract the user/line name from this queue. The rest of the method remains the same.

```

    String qName = replyQueue.getQueueName();
    // The queue name must always end with "Receiver".
    int i = qName.indexOf("Receiver");
    if(i <= 0)
        throw new PhoneException("Invalid Reply Destination " + qName);

    // Extract the user/line name
    // Call this variable "fromQueue" on purpose
    // so that the rest of the method doesn't have to change.
    String fromQueue = qName.substring(0,i);

```

That's it! With these changes in place, the phone example now uses the JMS Request/Reply feature.

Simulating Synchronous calls with Request/Reply

The most common use of Request/Reply is to simulate synchronous calls with asynchronous messaging. We saw the use of the `JMSReplyTo` header field above. There is another important header field related to Request/Reply – the `JMSCorrelationID` field. I discussed this field in chapter 5. It's main purpose is to tie a response message back to the request that resulted in the response.

The steps required by a client to use the Request/Reply style to simulate a synchronous request in the point-to-point messaging style are:

1. Create a temporary queue by calling the `createTemporaryQueue` method on the queue session.
2. Set the value of the `JMSReplyTo` header property equal to this temporary queue.
3. Send the message. I would recommend enabling message ID generation by calling the `setDisableMessageID` method on the queue sender with a `true` parameter.
4. Execute a blocking receive on the temporary queue i.e. call the `receive` method on a queue receiver for this temporary queue.

The steps taken by a client using the publish-and-subscribe messaging style would be identical except that it would use the JMS objects of that domain, such as topics instead of queues, etc. To reduce the amount of work that clients have to do, JMS provides a couple of helper classes, `QueueRequestor` and `TopicRequestor` that handle all the setup work required in steps 1–4. Let's take a look at the `QueueRequestor` class, which is listed below for reference (The `TopicRequestor` class is similar and will not be shown or discussed):

```
public class QueueRequestor {
    // The queue session the queue belongs to.
    QueueSession session;
    // The queue to perform the request/reply on.
    Queue queue;
    TemporaryQueue tempQueue;
    QueueSender sender;
    QueueReceiver receiver;

    public QueueRequestor(QueueSession session, Queue queue)
        throws JMSEException {
        this.session = session;
        this.queue = queue;
        // create a temporary queue which will
        // serve as the response destination
        tempQueue = session.createTemporaryQueue();
        // The sender sends on the queue specified
        // by the client.
        sender = session.createSender(queue);
        // Create a receiver to receive messages on the
        // temporary queue
        receiver = session.createReceiver(tempQueue);
    }

    public Message request(Message message) throws JMSEException {
        // set the reply destination
        message.setJMSReplyTo(tempQueue);
        // send the message
        sender.send(message);
        // wait for a response.
        return (receiver.receive());
    }

    public void close() throws JMSEException {
        // publisher and consumer created by
        // constructor are implicitly closed.
        session.close();
        tempQueue.delete();
    }
}
```

A client creates an instance of the `QueueRequestor` class passing in a queue and the session used to create this queue. This session must be non-transacted and support either the `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE` message acknowledgement mode. When a client wants to send a request, it simply invokes the `request` method on this instance passing it the message. This message handles steps 1–4 described above. The client blocks till the server sends a message back on the temporary queue specified in the `JMSReplyTo` message header field. The `QueueRequestor` class assumes that the server will always send a response and only one such response will be sent per request. If a server sends multiple responses then the next request sent by a client will [erroneously] get the second response sent by the server to the first request.

Now let's look at what the server i.e. the application receiving the message from the client, must do to participate in the Request/Reply dance.

1. Get the reply destination from the JMSReplyTo message header field.
2. Process the message and create a response message.
3. Set the JMSCorrelationID header field in the response message equal to the value of the JMSMessageID header field in the request message.
4. Send the response message to the destination obtained in step 1.

For example, a server using point-to-point messaging would do along these lines

```
// Get the reply destination.
Queue replyQueue = (Queue) msg.getJMSReplyTo();

// Process the request message "msg"?
.
.
.

// Create a response message and set its contents?
Message reply = ?

// Set the
// JMSCorrelationID of the response == JMSMessageID of the request
reply.setJMSCorrelationID(msg.getJMSMessageID());

// Send the response back
QueueSender replier = session.createQueueSender(null);
replier.send(replyQueue, reply);
```

Note how the `replier` queue sender object is created with no queue specified. This is important to be able to use the `send` method that takes a queue as its parameter. I discussed this in the "QueueSender" section earlier in this chapter.

Finally, let's look at a complete client/server application that uses the Request/Reply feature of JMS. The server application is capable of doing three simple types of computations – addition, subtraction, and multiplication. A client can send the server a request message in the following format:

Add|Subtract|Multiply [A space delimited list of numbers]

An example of a request is "Add 10 30 40.2 -15"

To start the server execute the following at a dos prompt:

```
REM Set up the classpath. My installation of Sun's Java Message Queue
REM is in the directory E:\Program Files\JavaMessageQueue1.0
set CPATH=.;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmq.jar;
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar

java -cp "%CPATH%" -Djava.compiler=NONE Server
```

Now let's start a client that will send the following three requests to the server

- Add 10 20 -15 2.5
- Subtract 10 20
- Multiply 10 20 6

To do so execute the following commands at a dos prompt:

```

REM Set up the classpath. My installation of Sun's Java Message Queue
REM is in the directory E:\Program Files\JavaMessageQueue1.0
set CPATH=.;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmq.jar;
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar

java -cp "%CPATH%" -Djava.compiler=NONE Client "Add 10 20 -15 2.5"
"Subtract 10 20" "Multiply 10 20 6"

```

```

C:\WINNT\System32\cmd.exe - java -cp ".;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar;...
E:\My Documents\Personal\Articles and Papers\JMS Book\RequestReply>java -cp "
.;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar;E:\Program Files\JavaMessa
geQueue1.0\lib\jmq.jar;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar"
-Djava.compiler=NONE Server
Server is ready and will run for 10 minutes.
[Command] Add 10 20 -15 2.5
Performing Add
Adding 10.0
Adding 20.0
Adding -15.0
Adding 2.5
Result of this command is 17.5
[Command] Subtract 10 20
Performing Subtract
Subtracting 10.0
Subtracting 20.0
Result of this command is -10.0
[Command] Multiply 10 20 6
Performing Multiply
Multiplying 10.0
Multiplying 20.0
Multiplying 6.0
Result of this command is 1200.0

```

Figure 7: The server window after the client has executed three commands

```

C:\WINNT\System32\cmd.exe
E:\My Documents\Personal\Articles and Papers\JMS Book\RequestReply>java -cp "
.;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar;E:\Program Files\JavaMessa
geQueue1.0\lib\jmq.jar;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar"
-Djava.compiler=NONE Client "Add 10 20 -15 2.5" "Subtract 10 20" "Multiply 1
0 20 6"
Result for request Add 10 20 -15 2.5 is 17.5
Result for request Subtract 10 20 is -10.0
Result for request Multiply 10 20 6 is 1200.0

E:\My Documents\Personal\Articles and Papers\JMS Book\RequestReply>

```

Figure 8: The client window after execution

Figure 7 shows the server window after it has finished processing the three requests. Figure 8 shows the client window after the same.

The Client

Ok, time to look at how it's done. Let's look at the client program first. By looking at the `main` method we'll be able to cover every aspect of the client program. So let's walk through it step-by-step.

1. First, the `main` method checks to see if there is at least one command line parameter and if there's not, it prints out an error message.

```
// Must have one paramter.
if( args.length < 1 ) {
    System.out.println("You must pass in at least" +
        " one request e.g. \"Add 40 5 31 65\"");
    System.exit(-1);
}
```

2. Next, it creates a new `Client` instance. The `Client` constructor is pretty straightforward. It gets a reference to a queue connection factory, uses this to create a connection and uses the connection to create a new session. The session is used to create a queue with the name "ServerQueue", which is previously agreed upon name between the client and server. Think of this as a host address/port number combination to contact a server in a traditional RPC system. The constructor also creates an instance of a `QueueRequestor` class passing it the session and the queue. This requestor class will be used to do all the grunge work required to use `Request/Reply` later on. The constructor is shown below:

```
public Client() throws JMSEException {
    // Create a connection factory,
    // get the connection and session.
    QueueConnectionFactory connectionFactory =
        new com.sun.messaging.QueueConnectionFactory();
    connection = connectionFactory.createQueueConnection();
    session = connection.createQueueSession(
        false, Session.AUTO_ACKNOWLEDGE);
    // Agreed upon queue: "ServerQueue"
    Queue queue = session.createQueue("ServerQueue");
    // And the requestor to help with the request/reply.
    requestor = new QueueRequestor(session, queue);
    // setup complete.start the connection.
    connection.start();
}
```

3. The method passes each command line parameter to the client's `execute` method as follows:

```
for(int i=0; i<args.length; i++)
    System.out.println("Result for request " + args[i] +
        " is " + client.execute(args[i]));
```

This method does three things:

- a. Create a new text message and set its content equal to the parameter passed in.

```
// Create the request message
TextMessage msg = session.createTextMessage();
msg.setText(serverCommand);
```

- b. Use the requestor to make the request to the server using its `request` method.

```
// Instead of sending the message directly,
// we will use the QueueRequestor.
Message response = requestor.request(msg);
```

- c. Take the message returned by the `request` method on the requestor and return its text content back to the caller (in this case the `main` method).

```
// The message from the Server should be a TextMessage.
```

```
        // Return the result to the caller.
        return ((TextMessage)response).getText();
```

4. Finally, after command line parameters have been processed, the main method calls destroy on the client as follows:

```
        // cleanup
        client.destroy();
```

This method will perform the cleanup work (i.e. call the close method on the session and connection objects) required prior to shutting down the client.

Following is the the client program in its entirety:

```
import javax.jms.*;
public class Client {

    // private member variables for the
    // queue connection and session
    private QueueConnection connection = null;
    private QueueSession session = null;
    // Leverage the QueueRequestor class to make our life easy.
    private QueueRequestor requestor = null;

    public static void main(String[] args) {

        // Must have one paramter.
        if( args.length < 1 ) {
            System.out.println("You must pass in at least " +
                " one request e.g. \"Add 40 5 31 65\\");
            System.exit(-1);
        }

        try {
            // Create the client
            Client client = new Client();
            // execute each request
            for(int i=0; i<args.length; i++)
                System.out.println("Result for request " + args[i] +
                    " is " + client.execute(args[i]));
            // cleanup
            client.destroy();
        }
        catch( Exception e ) {
            System.err.println(e.getMessage());
        }
    }

    public Client() throws JMSEException {
        // Create a connection factory,
        // get the connection and session.
        QueueConnectionFactory connectionFactory =
            new com.sun.messaging.QueueConnectionFactory();
        connection = connectionFactory.createQueueConnection();
        session = connection.createQueueSession(
            false, Session.AUTO_ACKNOWLEDGE);

        // Agreed upon queue: "ServerQueue"
        Queue queue = session.createQueue("ServerQueue");

        // And the requestor to help with the request/reply.
        requestor = new QueueRequestor(session, queue);

        // setup complete.start the connection.
        connection.start();
    }
}
```

```

    }

    // Clean up funtion.
    public void destroy() throws JMSEException {
        session.close();
        connection.close();
        connection = null;
    }

    // execute a server command.
    public String execute(String serverCommand) throws JMSEException {
        // Create the request message
        TextMessage msg = session.createTextMessage();
        msg.setText(serverCommand);

        // Instead of sending directly, we will
        // use the QueueRequestor.
        Message response = requestor.request(msg);

        // The message should be a TextMessage.
        // Return the result to the caller.
        return ((TextMessage)response).getText();
    }
}

```

The Server

Now let's take a look at the server program. The main method simply creates a new instance of the `Server` and waits for 10 minutes before exiting.

```

// Start the server.
Server server = null;
try {
    // create a new server instance
    server = new Server();
    System.out.println(
        "Server is ready and will run for 10 minutes.");
    // wait for 10 minutes as stated
    Thread.sleep(10*60*1000);
}
.
.
.

```

I used the "10 minute sleep" algorithm (*) because I was feeling too lazy to create an elaborate shutdown mechanism. The `Server` constructor is similar to that of the `Client`. It gets a reference to a queue connection factory, uses this to create a connection and uses the connection to create a new session. The session is used to create a queue with the name "ServerQueue", which as we know by now is a previously agreed upon name between the client and server. This is where the similarity ends. The `Server` constructor proceeds by creating a queue sender without associating the sender with any specific queue as shown below.

```

// Create a sender associated with NO queue
replier = session.createSender(null);

```

This means that the server program must specify the queue to send the message to as a parameter in the `send` method on the sender. As we'll see later, the server program knows which queue to send the message to from the `JMSReplyTo` header property in the message received from the client.

The constructor also creates a receiver to receive messages from the queue and installs itself (i.e. the `Server` class instance) as a message listener.

```
// create a receiver and install a message listener.
QueueReceiver receiver = session.createReceiver(queue);
receiver.setMessageListener(this);
```

This is possible because `Server` implements the `MessageListener` interface. As we know this interface has one method `onMessage`. In our case this is the heart and brains of the server. So, let's take a detailed look at this method:

1. Cast the message to a `TextMessage`, since this is the message type we are expecting from the client.

```
// The request is a text message
TextMessage txtMsg = (TextMessage)msg;
```

2. Extract the command from the text message.

```
// Get the command
String command = txtMsg.getText().trim();
```

The `trim` method will remove all leading and trailing white space from the command.

3. Check for the reply destination. If no destination is found it is an error condition and we stop processing.

```
// Check for a ReplyTo topic
Queue replyQueue = (Queue) msg.getJMSReplyTo();
if( replyQueue == null ) {
    System.err.println("Error: No Reply Queue Specified.");
    return;
}
```

4. Now, parse the command. Remember the command is in the form "Add 5 10 35" i.e. the operation name followed by any number of space separated parameters. So, first we find the operation as follows:

```
// Get the operation.
String operation;
int i = nextSpace(0,command);
if( i == -1 ) {
    System.err.println("No operation specified.");
    return;
}
operation = command.substring(0,i);
```

We assume that there no leading whitespaces in the command. This is a valid assumption since we called `trim` in step 2. Note the use of the private helper method `nextSpace`. This method and its corollary `nextNonSpace` will be used throughout the parsing. The `nextSpace` method takes a string parameter and a starting index and returns the index of the next space or `-1` if no more spaces are found. For example, if this method was passed the string "Hello John Smith" and a starting index of 6, it would return 10. The `nextNonSpace` method is very similar except that it does the same for nonspaces. So if this method was passed the same string "Hello John Smith" and a starting index of 5, it would return 6. Note that I do not use the `StringTokenizer` class since there is no guarantee that the operation name and the parameters will be separated by one and only space each. For example, this is not considered invalid "Add 5 10". Using `StringTokenizer` in this case would be cumbersome.

Now that we know the operation to be performed, we simply get each parameter and perform the required operation. To get the next paramter, we use the following logic:

```

// where does the next paramter start?
i = nextNonSpace(i,command);
// if i = -1, then there are no more parameters
if( i== -1 )
    break;
// where does this paramter end i.e. how many digits
// does this paramter have?
j = nextSpace(i,command);
// if j = -1 then this is the last paramter
if( j == -1 )
    j = command.length();
// Ok, so the paramter starts at "i" and ends at "j"
double val = new Double(command.substring(i,j)).doubleValue();

```

I told you we will rely heavily on the `nextSpace` and `nextNonSpace` methods. Now that we have the next (or first) parameter, we perform the operation as follows:

```

// for the first number, the result == the number.
if( firstNumber ) {
    firstNumber = false;
    result = val;
}
// for all the other numbers carry out the operation.
else if( operation.equals("Add") )
    result += val;
else if( operation.equals("Subtract") )
    result -= val;
else if( operation.equals("Multiply") )
    result *= val;

```

Note the special case for the first parameter for which the result is simply set to the parameter itself. For all parameters the operation is actually performed.

5. Send the result back to the client as follows:

```

// Send the reply back to the client.
TextMessage reply = session.createTextMessage();
reply.setText(new Double(result).toString());
// set the correlation ID == the request message ID.
reply.setJMSCorrelationID(msg.getJMSMessageID());
// send the response message to the specified temp queue.
replier.send(replyQueue,reply);

```

As shown above, we simply create a new text message, set the result in the message, set the correlation ID equal to the message ID, and send the message to the specified queue i.e. `replyQueue`. Remember, we obtained this queue in step 3.

The last method that we need to look at is `shutdown`. This method is called by the `main` method after it wakes up in 10 minutes. This is shown below.

```

.
.
.
finally {
    // cleanup before exit.
    if( server != null )
        server.shutdown();
}
// done...
System.exit(0);

```

This method will ensure that the server cleans up before shutting down as shown below.

```

// method to cleanup

```



```

public void shutdown() {
    try {
        // don't need to explicitly close the
        // receiver. Closing the session is enough.
        session.close();
        connection.close();
        connection = null;
    }
    catch( Exception e ) {
    }
}

```

Following is the server program in its entirety:

```

import javax.jms.*;

public class Server implements javax.jms.MessageListener {

    // private member variables for the
    // queue connection, session, and sender
    private javax.jms.QueueConnection connection = null;
    private javax.jms.QueueSession session = null;
    private javax.jms.QueueSender replier = null;

    public static void main(String argv[]) {
        // Start the server.
        Server server = null;
        try {
            // create a new server instance
            server = new Server();
            System.out.println(
                "Server is ready and will run for 10 minutes.");
            // wait for 10 minutes as stated
            Thread.sleep(10*60*1000);
        }
        catch( Exception e ) {
            System.err.println(e.getMessage());
        }
        finally {
            // cleanup before exit.
            if( server != null )
                server.shutdown();
        }
        // done...
        System.exit(0);
    }

    public Server() throws JMSEException {
        // Create a connection factory,
        // get the connection and session.
        QueueConnectionFactory connectionFactory =
            new com.sun.messaging.QueueConnectionFactory();
        connection = connectionFactory.createQueueConnection();
        session = connection.createQueueSession(
            false, javax.jms.Session.AUTO_ACKNOWLEDGE);

        // Create the previously agreed upon queue "ServerQueue"
        Queue queue = session.createQueue("ServerQueue");

        // create a receiver and install a message listener.
        QueueReceiver receiver = session.createReceiver(queue);
        receiver.setMessageListener(this);

        // Create a sender associated with NO queue
        // The queue will be specified in the send method
        replier = session.createSender(null);
    }
}

```

```

    // setup complete. start the connection.
    connection.start();
}

// method to cleanup
public void shutdown() {
    try {
        session.close();
        connection.close();
        connection = null;
    }
    catch( Exception e ) {
    }
}

public void onMessage(Message msg) {
    try {
        // The request is a text message
        TextMessage txtMsg = (TextMessage)msg;

        // Get the command and show it
        String command = txtMsg.getText().trim();
        System.out.println( "[Command] " + command );

        // Check for a ReplyTo topic
        Queue replyQueue = (Queue) msg.getJMSReplyTo();
        if( replyQueue == null ) {
            System.err.println("Error: No Reply Queue Specified.");
            return;
        }

        // Parse the command

        // Get the operation.
        String operation;
        int i = nextSpace(0,command);
        if( i == -1 ) {
            System.err.println("No operation specified.");
            return;
        }
        operation = command.substring(0,i);
        System.out.println("Performing " + operation);

        // Get each number and perform the requested operation.

        // the result.
        double result = 0.0;
        // firstNumber is true for the first number.
        boolean firstNumber = true;
        int j;
        while( i != -1 ) {
            i = nextNonSpace(i,command);
            if( i== -1 )
                break;
            j = nextSpace(i,command);
            if( j == -1 )
                j = command.length();
            double val =
                new Double(command.substring(i,j)).doubleValue();
            System.out.println(operation + "ing " + val);

            // for the first number, the result == the number.
            if( firstNumber ) {
                firstNumber = false;
                result = val;
            }
            // for all the other numbers carry out the operation.
            else if( operation.equals("Add") )

```

```

        result += val;
    else if( operation.equals("Subtract") )
        result -= val;
    else if( operation.equals("Multiply") )
        result *= val;
    i=j;
}

System.out.println("Result of this command is " + result);

// Send the modified message back.
TextMessage reply = session.createTextMessage();
reply.setText(new Double(result).toString());
// set the correlation ID == the request message ID.
reply.setJMSCorrelationID(msg.getJMSMessageID());
// send the response message to the specified temp queue.
replier.send(replyQueue,reply);
}
catch( Exception e ) {
    System.err.println(e.getMessage());
}
}

// Find and return the index of the next space in string "s"
// starting at index "start"
private int nextSpace(int start, String s) {
    int i = start;
    for( ;i<s.length(); i++ )
        if( s.charAt(i) == ' ' )
            break;
    if( i == s.length() )
        return(-1);
    return(i);
}

// Find and return the index of the next non-space
// in string "s" starting at index "start"
private int nextNonSpace(int start, String s) {
    int i = start;
    for( ;i<s.length(); i++ )
        if( s.charAt(i) != ' ' )
            break;
    if( i == s.length() )
        return(-1);
    return(i);
}
}
}

```

To summarize JMS provides the following to support Request/Reply

- The `JMSReplyTo` and `JMSCorrelationID` message header fields.
- The ability to create temporary queues and topics.
- A set of helper classes for both the PTP and Pub/Sub domains that implement a basic form of request/reply.

A Limitation of Request/Reply

From the above discussion it should be obvious that the support for Request/Reply does not come free. Both clients and servers taking part in the Request/Reply dance are fully aware of this and do extra work to enable this. The support provided by JMS for this is very rudimentary and fragile.

What is the protocol that a server must follow? Does the existence of a destination in the `JMSReplyTo` header field mean that the server must not send a response to the "regular" destination but to the one specified? Request/Reply is not something that is transparently handled by a JMS provider; not because JMS providers cannot or will not, but because the JMS

specification does not specify a standard way of doing this. In my opinion, this is one place where JMS could have specified a little bit more without sacrificing its goal of portability.

Summary

In this [long] chapter, I discussed the various messaging styles supported by JMS. JMS does not mandate the support of both point-to-point and publish-and-subscribe styles, so not all providers may support these. However, whenever they do support a style all material discussed in this chapter for that style will apply. In the next chapter I will talk about using XML with JMS, more specifically sending and receiving XML messages.

Chapter 7

Using XML with JMS

What can be better than using one "hot" standard? The answer is, using two "hot" standards! The standards I am referring to are JMS and XML. So far, this whole book has been devoted to JMS, so let's talk a little bit about XML.

An XML Refresher

Probably no other three-letter acronym (TLA) has become as popular as XML. XML stands for Extensible Markup Language. The World Wide Web Consortium (W3C, another TLA) introduced the world to XML in February 1998 with the release of the XML 1.0 specification. The W3C defines XML as "the universal format for structured documents and data on the Web." XML is a subset of the Standard Generalized Markup Language (SGML)⁶. An important point to remember is that XML is not a single technology; but rather a family of related and complementing technologies, such as XSL (and XSLT), XPath, XLink, XPointer, X Fragments, DOM, XML namespaces, XML Schemas, and many more.

A major advantage of XML is that it is (as the name indicates) extensible. This means that you are not restricted to only those features, or tags, that the W3C thought about, as is the case with HTML. HTML is a W3C standard as well, but with HTML, the tags available to you are well defined and cannot be changed. With XML, you can add your own tags to suit your own data.

Let's go through an example. Among other characteristics, a book has a title, an author, a publication date, a publisher, an ISBN, and a price. To represent this as HTML, you could do the following:

```
<html>
  <head><title>A Book</title></head>
  <body>
    <h1>Understanding JMS</h1>
    <h2>by Tarak Modi</h2>
    <h3>Published by ABC, Inc. in February 2001</h3>
    <h4>ISBN 0023456912</h4>
    <h4>Price $35.99 (U.S.)</h4>
  </body>
.</html>
```

There are two major noteworthy points:

1. The book information has been "force fitted" in the available tags. Merely looking at the HTML without the context of the actual content does not tell you what it is describing.
2. The HTML has the presentation logic tightly coupled with the data it is describing. This presentation logic is used by a browser such as Netscape Navigator or Internet Explorer (IE) to display the data to the user. As an example figure 1 shows the above html in IE5.0.

XML is different in both these ways. First, it allows you to define your own tags that suit your domain, in essence creating your own markup language. Examples of such specialized markup languages include the Math Markup Language (MathML) and the Commerce XML (cXML).

⁶ And so is HTML. In that sense, HTML and XML are siblings.

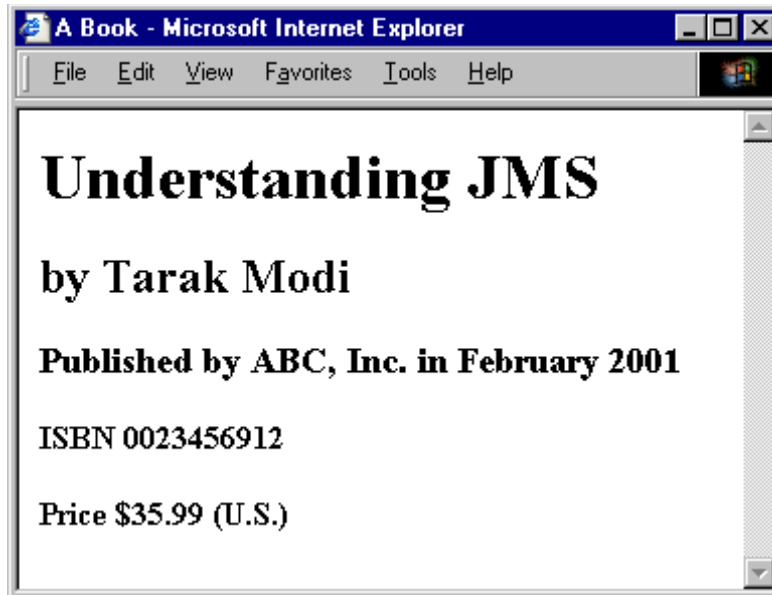


Figure 1: The book in HTML as seen in IE5.0

Second, the presentation logic is completely decoupled from the actual content. For example, XSL may be used to transform the XML data into presentable HTML.

Let's define our own version of XML for the Book. We'll call this the "BookML". There are no established set of rules that one follows while creating a new markup language, except that it must be the XML must *well-formed* and *valid*. I'll explain both these terms later. In BookML each characteristic of the book will have its own tag. So we'll have a tag for title called `Title`, a tag for the author called `Author`, and so on. Using BookML, we can describe the book as follows:

```
<Book>
  <Title>Understanding JMS</Title>
  <Author>Tarak Modi</Author>
  <PublicationDate>February 2001</PublicationDate>
  <Publisher>ABC, Inc.</Publisher>
  <ISBN>0023456912</ISBN>
  <Price>35.99</Price>
</Book>
```

Much better. This now looks like a book description. The above XML is said to be well formed, since it obeys the rules of writing XML, such as each *begin* tag has a corresponding *end* tag. Also note that unlike HTML, XML is case-sensitive. That means `Title`, `title`, and `TITLE` are all different tags. We can make further refinements to the above XML. For example, the `Author` tag can contain `First` and `Last` tags corresponding to the first and last name of the author; the `PublicationDate` tag can contain a `Month` and `Year` tag corresponding to the month and year of publication; and the `Price` tag can have a `currency` attribute that determines what currency the price is stated in. The refined XML looks like this:

```
<Book>
  <Title>Understanding JMS</Title>
  <Author>
    <First>Tarak</First>
    <Last>Modi</Last>
  </Author>
  <PublicationDate>
    <Month>February</Month>
    <Year>2001</Year>
  </PublicationDate>
```

```

    <Publisher>ABC, Inc.</Publisher>
    <ISBN>0023456912</ISBN>
    <Price currency="USD">35.99</Price>
</Book>

```

Although the above XML is well formed, it is not valid. To be considered valid, it must have a Document Type Definition (DTD)⁷ associated with it. A DTD is used to check the validity of an XML document at runtime. In a sense it defines the grammar, of a language such as BookML. Let's look at the DTD for the BookML language

```

<!DOCTYPE Book [
  <!ELEMENT Book (Title,Author,PublicationDate,Publisher,ISBN,Price)>
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Author (First,Last)>
  <!ELEMENT First (#PCDATA)>
  <!ELEMENT Last (#PCDATA)>
  <!ELEMENT PublicationDate (Month,Year)>
  <!ELEMENT Month (#PCDATA)>
  <!ELEMENT Year (#PCDATA)>
  <!ELEMENT Publisher (#PCDATA)>
  <!ELEMENT ISBN (#PCDATA)>
  <!ELEMENT Price (#PCDATA)>
  <!ATTLIST Price currency NMTOKEN #REQUIRED>
]>

```

The above DTD defines that `Book` is the root node (i.e. tag) for any valid BookML document. It also states that a `Book` may have one `Title`, `Author`, `PublicationDate`, `Publisher`, `ISBN`, and `Price`, in that order. All other elements are similar, except `Price`, which has a mandatory attribute called `currency`. `#PCDATA` is a reserved word that means Parsed Character Data that allows a tag to contain any text data that is not considered markup. So the text "John" is allowed but not `<John>`.

Manipulating XML Programmatically

The primary APIs for accessing XML documents include the Document Object Model (DOM) and the Simple API for XML (SAX). DOM is a W3C approved standard that provides an object model for accessing pieces of the XML document. SAX is an event-driven API created jointly by the members of the XML-DEV mailing list. Both APIs have their pros and cons, which I will not discuss here. The Java DOM (JDOM) is a recent introduction that is targeted to make the DOM more accessible to Java programmers. The Java API for XML Parsing (JAXP) is yet another API introduced through the Java Community Process (JCP) and is also available to Java programmers. This API allows programmers to use different SAX and DOM compliant parsers requiring knowledge of the specific implementation of the parser. *In that regard, JAXP is similar to JMS.*

Finally?

An increasing popular view of XML is that XML is just another serialization format that will never be manipulated or viewed directly by humans. This puts XML in line with protocols such as the General Inter-ORB Operability Protocol (GIOP) and RPC. A major contention at this point is that XML is simply not efficient enough when it comes to transferring data across the wire. Since XML is a text format, and it uses tags to delimit the data, XML files are nearly always larger than comparable binary formats. This was a conscious decision by the XML developers to leverage the benefits of a text format, such as easier debugging of applications using XML (ever try debugging a wire protocol such as IIOP?) and a vast base of tool support to create, edit, and maintain XML. The disadvantages of a "bloated" serailization can usually be compensated at a different level. For example, programs such as zip and gzip can compress files very well and very fast. Those programs are available for nearly all platforms (and are usually free). In addition, communication protocols such as HTTP/1/1 (the core protocol of the Web) can compress data on the fly, thus

⁷ It is said that XML Schemas will replace DTDs in the future, but that hasn't happened yet.

saving bandwidth as effectively as a binary format. Examples of the more popular serialization protocols that use XML as the serialized format are XML-RPC and the Simple Object Access Protocol (SOAP).

Back to JMS

If you want to use XML with JMS, you're not alone. For example, one could easily perceive a distributed system that uses a JMS provider as the distribution middleware as shown in figure 2.

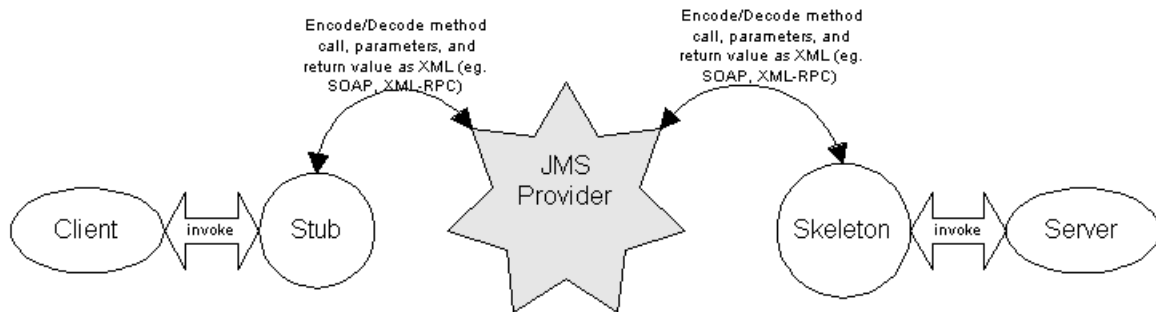


Figure 2: A distribution system that uses JMS

Just as in any other distributed system such as CORBA or RMI, the client has access to a stub that it thinks is the actual remote object. When the client invokes a method on the stub, the stub creates an XML packet that represents the method call and sends this packet to the skeleton. The skeleton parses the XML packet and invokes the method on the actual object. It then creates an XML packet that contains the result of the invocation back and sends this packet back to the stub. The stub parses the returned packet and returns the result to the client. The uniqueness of this system is that the stub and skeleton use a JMS provider for sending the XML packets back and forth.

The creators of JMS anticipated the extent to which XML would permeate the enterprise applications of the future and created the `TextMessage` message type to support XML. Since XML is a text representation it can safely be transported in JMS text messages.

To further help you use XML with JMS; I have created a class `XmlJMSBridge` that does all the grunge work of creating a JMS message from an XML `Document` object and vice versa. I have used JAXP in order to increase portability of the code by supporting a variety of parsers.

How JAXP provides portability

The JAXP API, contained in the `jaxp.jar` file, is comprised of the `javax.xml.parsers` package. That package contains two vendor-neutral factory classes: `SAXParserFactory` and `DocumentBuilderFactory` that give you a SAX parser and a `DocumentBuilder`, respectively. The `DocumentBuilder`, in turn, creates a DOM-compliant `Document` object. The key to portability here is that *the factory APIs give you the ability to plug in an XML implementation offered by another vendor without changing your source code*. The implementation you get depends on the setting of the `javax.xml.parsers.SAXParserFactory` and `javax.xml.parsers.DocumentBuilderFactory` system properties. The default values (unless overridden at runtime) point to Sun's reference implementations in the `com.sun.xml` package.

So how does the `XmlJMSBridge` class work? Let's start by examining the constructors. There are two versions of the constructor. The first version is the default constructor that simply delegates to

the second version, which takes two `boolean` parameters. The first `boolean` parameter indicates whether validation is turned on. To turn validation on this parameter must be `true`. A DTD must be specified with the XML document to use validation. The second `boolean` parameter is used to toggle the namespace support. Namespaces are an advanced XML concept beyond the scope of this book. The constructor uses the JAXP API to create a builder object, which is an object used to parse an XML document and return a `DOM Document` object. The constructor is shown below:

```
public XmlJMSBridge(boolean supportsValidation,
    boolean supportsNamespace) throws JMSEException
{
    try {
        // create the factory
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        factory.setValidating(supportsValidation);
        factory.setNamespaceAware(supportsNamespace);
        // use it to get a builder.
        builder = factory.newDocumentBuilder();
    }
    catch( Exception e ) {
        JMSEException e1 = new JMSEException(e.getMessage());
        e1.setLinkedException(e);
        throw e1;
    }
}
```

When a client is manipulating XML, it will most likely do so in the form of a `DOM Document` object. Later when the client wants to send a message containing this XML it should call the `createJMSMessage` method passing in the document object. This method also takes a reference to a session object that it uses to create a new `JMS TextMessage` instance. It then calls the `getXmlAsString` method passing it the XML document object. The string returned from this call is then placed into the text message before returning it to the caller i.e. the client. The client can then send this message like any other JMS message.

The `getXmlAsString` method deserves special attention. DOM does not define a standard way of getting the string (text) representation of a `DOM Document` object. So this method relies on a proprietary way of doing this. Project X is Sun's reference implementation of JAXP, which also includes a class `XmlDocument` in the `com.sun.xml.tree` package. This class provides a method – `write` – that allows us to access the XML in text form. My implementation of the `getXmlAsString` method uses this method on the `XmlDocument` class as shown below:

```
// Cast to XmlDocument for write() operation
// The only non-standard class used in the bridge.
com.sun.xml.tree.XmlDocument xmlDocument
    = (com.sun.xml.tree.XmlDocument)document;
// The write method on the XmlDocument class
// takes a reference to an output stream.
StringWriter writer = new StringWriter();
xmlDocument.write(writer);
// Return the string to the caller.
return(writer.toString());
```

When a client receives a message that is known to contain an XML document, it can call the `setJMSMessage` method on the `XmlJMSBridge` instance passing in the received message. This method returns a `DOM Document` object to the caller. To do so, this method calls the `getXmlAsDOMDocument` method, which uses the builder object created earlier to parse the XML (in text form) contained in the message. This is shown below:

```
org.w3c.dom.Document doc =
    builder.parse(new org.xml.sax.InputSource(
        new StringReader(xmlString)));
```

The XmlJMSBridge class is shown in its entirety below:

```
import java.io.StringReader;
import java.io.StringWriter;
import javax.jms.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;

// The only non-standard class used in the bridge.
import com.sun.xml.tree.XmlDocument;

public class XmlJMSBridge {
    private DocumentBuilder builder = null;

    public XmlJMSBridge() throws JMSEException
    {
        // delegate it...
        this(false,false);
    }

    public XmlJMSBridge(boolean supportsValidation,
        boolean supportsNamespace) throws JMSEException
    {
        try {
            // create the factory
            DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
            factory.setValidating(supportsValidation);
            factory.setNamespaceAware(supportsNamespace);
            // use it to get a builder.
            builder = factory.newDocumentBuilder();
        }
        catch( Exception e ) {
            JMSEException e1 = new JMSEException(e.getMessage());
            e1.setLinkedException(e);
            throw e1;
        }
    }

    // A client calls this method to get a JMS
    // message that contains the document.
    public TextMessage createJMSMessage(Session session, Document doc)
        throws JMSEException
    {
        // session must be valid
        if( session == null )
            throw new JMSEException("Session cannot be null.");
        String xmlString = getXmlAsString(doc);
        TextMessage txtMsg = session.createTextMessage();
        txtMsg.setText(xmlString);
        return(txtMsg);
    }

    // A client calls this method to get the document
    // contained in this message.
    public Document setJMSMessage(Message msg)
        throws JMSEException
    {
        if( !(msg instanceof TextMessage) )
            throw new JMSEException("Invalid message.");
        TextMessage txtMsg = (TextMessage)msg;
        return getXmlAsDOMDocument(txtMsg.getText());
    }

    // Helper methods.
```

```

// Not part of the core use of the XmlJMSBridge

// Convert the given string into a document.
public Document getXmlAsDOMDocument(String xmlString)
    throws JMSEException
{
    try {
        return(builder.parse(
            new InputSource(new StringReader(xmlString))));
    }
    catch( Exception e ) {
        JMSEException e1 = new JMSEException(e.getMessage());
        e1.setLinkedException(e);
        throw e1;
    }
}

// Convert the given document into a string.
public String getXmlAsString(Document document)
    throws JMSEException
{
    try {
        // Cast to XmlDocument for write() operation
        // (Not defined until DOM Level 3.)
        XmlDocument xmlDoc = (XmlDocument)document;
        StringWriter writer = new StringWriter();
        xmlDoc.write(writer);
        return(writer.toString());
    }
    catch( Exception e ) {
        JMSEException e1 = new JMSEException(e.getMessage());
        e1.setLinkedException(e);
        throw e1;
    }
}
}

```

Note that the XmlJMSBridge class may be used with both point-to-point and publish-and-subscribe messaging styles, since there is no messaging style dependent code in this class.

Let's take a look at an example of using the XmlJMSBridge class. Two programs, a sender (XmlSender) and a receiver (XmlReceiver), have been created. The sender program reads in an XML file passed in as a command line parameter. It creates a DOM Document object from this XML using the getXmlAsDOMDocument method. Reading XML in from a file and then manipulating it as a DOM Document object is typical of an application using XML. Finally, the application is ready to send this XML as a JMS message to another application i.e. the receiver. It does this by calling the createJMSMessage method on the XmlJMSBridge class. To do so it passes in a session object and the document object. This method returns a JMS message to the caller, which can then be sent like any other JMS message. Both the sender and the receiver programs take an optional parameter to turn validation on. This value is passed into the XmlJMSBridge constructor as the first parameter. Remember, to use validation the XML must have a DTD associated with it.

Both the sender and receiver programs are listed below for reference.

```

////////////////////////////////////
//////////////////////////////////// XmlSender ///////////////////////////////////
////////////////////////////////////

import javax.jms.*;
import org.w3c.dom.Document;

public class XmlSender {

```

```

public static void main (String[] args) {

    // Must specify a file containing xml.
    if( args.length < 1 ) {
        System.err.println(
            "You must specify an XML file to parse.");
        System.exit(-1);
    }
    try {
        // Read the file and get the xml.
        String xml = "";
        java.io.FileInputStream in =
            new java.io.FileInputStream(args[0]);
        byte[] buf = new byte[5];
        while( true ) {
            int len = in.read(buf);
            if( len == -1 )
                break;
            xml += new String(buf,0,len);
        }

        // Is validation on?
        boolean validate = false;
        if( args.length == 2 && args[1].equals("validate") )
            validate = true;

        // Get the queue connection factory.
        // This is the only "Sun Java Message Queue"
        // specific code.
        // Note this is not part of the bridge.
        QueueConnectionFactory connectionFactory =
            new com.sun.messaging.QueueConnectionFactory();
        QueueConnection connection =
            connectionFactory.createQueueConnection();
        QueueSession session =
            connection.createQueueSession(false,1);
        // The queue name is "XmlQueue"
        Queue queue = session.createQueue("XmlQueue");
        QueueSender sender = session.createSender(queue);
        connection.start();

        // Create the bridge.
        XmlJMSBridge bridge =
            new XmlJMSBridge(validate,false);

        // A client gets access to some Xml and wants to
        // manipulate it.
        Document document = bridge.getXmlAsDOMDocument(xml);

        // This is the producer of the message.
        // Ask the bridge to create a JMS message
        // that contains the document.
        Message m =
            bridge.createJMSMessage(session,document);
        // send the message
        sender.send(m,DeliveryMode.PERSISTENT,9,60000);

        System.out.println("Sent message.");

        // A good client cleans up.
        session.close();
        connection.close();
    }
    catch( Exception e ) {
        e.printStackTrace();
        System.err.println(e.getMessage());
    }
    // Done...
}

```

```

        System.exit (0);
    }
}

////////////////////////////////////
////////////////////////////////////  XmlReceiver  //////////////////////////////////////
////////////////////////////////////

import javax.jms.*;
import org.w3c.dom.Document;

public class XmlReceiver {

    public static void main (String[] args) {

        try {
            // Is validation on?
            boolean validate = false;
            if( args.length == 1 && args[0].equals("validate") )
                validate = true;

            // Now simulate what a client would do...

            // Get the queue connection factory.
            // This is the only "Sun Java Message Queue"
            // specific code.
            // Note this is not part of the bridge.
            QueueConnectionFactory connectionFactory =
                new com.sun.messaging.QueueConnectionFactory();
            QueueConnection connection =
                connectionFactory.createQueueConnection();
            QueueSession session =
                connection.createQueueSession(false,1);
            // The queue name is "XmlQueue"
            Queue queue = session.createQueue("XmlQueue");
            QueueReceiver receiver =
                session.createReceiver(queue);
            connection.start();

            // Create the bridge.
            XmlJMSBridge bridge =
                new XmlJMSBridge(validate,false);

            // This is the consumer of the message.
            Message m = receiver.receive();
            // Ask the bridge to extract the document
            // contained in the JMS message.
            Document document = bridge.setJMSMessage(m);
            System.out.println(
                "\n***** Got the following Xml *****\n\n" +
                bridge.getXmlAsString(document));

            // A good client cleans up.
            session.close();
            connection.close();
        }
        catch( Exception e ) {
            e.printStackTrace();
            System.err.println(e.getMessage());
        }
        // Done...
        System.exit (0);
    }
}

```

Sample usage

```

REM set the classpath for JAXP and JMS
set CPATH=.;E:\dev\pc\lib\thirdparty\jaxp.jar;
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jms.jar
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmq.jar;
set CPATH=%CPATH%;E:\Program Files\JavaMessageQueue1.0\lib\jmqadmin.jar

```

```

REM Run the XmlSender with validation?
java -Djava.compiler=NONE -cp "%CPATH%" XmlSender Book.xml true

```

```

REM Run the XmlReceiver with validation
java -Djava.compiler=NONE -cp "%CPATH%" XmlReceiver true

```

And finally, Book.xml is shown below:

```

<?xml version='1.0' encoding='us-ascii'?>
<!DOCTYPE Book [
  <!ELEMENT Book (Title,Author,PublicationDate,Publisher,ISBN,Price)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Author (First,Last)>
    <!ELEMENT First (#PCDATA)>
    <!ELEMENT Last (#PCDATA)>
    <!ELEMENT PublicationDate (Month,Year)>
    <!ELEMENT Month (#PCDATA)>
    <!ELEMENT Year (#PCDATA)>
    <!ELEMENT Publisher (#PCDATA)>
    <!ELEMENT ISBN (#PCDATA)>
    <!ELEMENT Price (#PCDATA)>
    <!ATTLIST Price currency NMTOKEN #REQUIRED>
  ]>
<Book>
  <Title>Understanding JMS</Title>
  <Author>
    <First>Tarak</First>
    <Last>Modi</Last>
  </Author>
  <PublicationDate>
    <Month>February</Month>
    <Year>2001</Year>
  </PublicationDate>
  <Publisher>ABC, Inc.</Publisher>
  <ISBN>0023456912</ISBN>
  <Price currency="USD">35.99</Price>
</Book>

```

Summary

XML has become the lingua franca of the e-commerce world. Luckily for us, the architects of JMS anticipated this and have not precluded the use of XML with JMS. In this chapter, I've taken their efforts one step further and have provided a helper class to further ease the use of XML with JMS.

In the next chapter, I will discuss a practical application of using a JMS provider. I will introduce you to space-based programming and show you how to implement your own space by leveraging a JMS provider. This application is based on a real-world project that I have done for an eCRM vendor, Online Insight.

Chapter 8

Space-based Programming with JMS

Until now we've focused almost all of our attention to the theoretical aspects of working with JMS. Even the examples discussed in the previous chapter were concentrated mainly on illustrating the technical aspects of the point-to-point, publish/subscribe, and request/reply messaging styles. In other words those examples were somewhat academic in nature. In this chapter, I will discuss a practical (and real world) application of a JMS compliant messaging product that will serve to tie together many of the concepts discussed throughout this book. This chapter requires some background of working with distributed systems.

In chapter 1, I claimed that programming distributed systems is hard and stated some reasons why this is so such as dealing with disparate machine architectures, operating systems, and network issues including failures and latency. The introduction of standard middleware solutions such as JMS has solved many of these problems. But anyone who's worked with distributed systems long enough knows very well that there are other issues involved in the design of such systems as well. These issues are not directly solved by existing middleware such as JMS. For example, one of the problems of designing highly distributed systems is figuring out how these systems discover each other. After all, the whole point of having distributed systems is to allow flexible and perhaps even dynamic configurations to maximize system performance and availability. So how do these distributed components of one system or multiple systems discover each other? Furthermore, once these systems have discovered one another, how do we allow for capabilities that yield fail-safe operation, such as rediscovery? Space based programming may provide us with a very good answer to these questions and more.

I begin this chapter by introducing the space-based programming concept and how it may be used towards mitigating some of the issues mentioned above. I will then discuss a technique of converting a JMS compliant message queue into a space. A list of resources is provided at the end of the chapter for readers interested in learning more about space-based programming and applications.

What is a Space?

As discussed in chapter 1, conventional distributed tools rely on passing messages between processes (asynchronous communication) or invoking methods on remote objects (synchronous communication). A space is an extension of the asynchronous communication model. In the space-based programming approach the two processes are not passing messages to each other. In fact the processes are totally unaware of each other. Rather, these processes pass their messages to an intermediary, the space.

Let's look at figure 1 for a moment. Process 1 places a message into the space. Process 2, which has been waiting for this type of message, takes the message out the space. Process 2 processes the message and based on the results places another message into the space. Process 3, which has been waiting for this [second] type of message, takes this message out of the space. Based on this we can make the following observations about this architecture:

1. The space may contain different types of messages. In fact I use the term "message" for clarity. These messages are actually just "things", i.e. the message may be an object, an XML document, or anything else that the space allows to be put in it. In figure 1 the different shapes in the space illustrate the different types of messages.
2. The three processes involved have no knowledge of one another. All they know is that they put a message in a space and get a message out of the space.
3. As in the message passing scenario, we are not limited to two processes; rather any number of processes may communicate via a common space. This allows the creation of extremely loosely coupled systems that can be highly distributed and extremely flexible.
4. Because a process only takes a message out of the space when it has the processing capacity available, this architecture results in natural load-balancing.

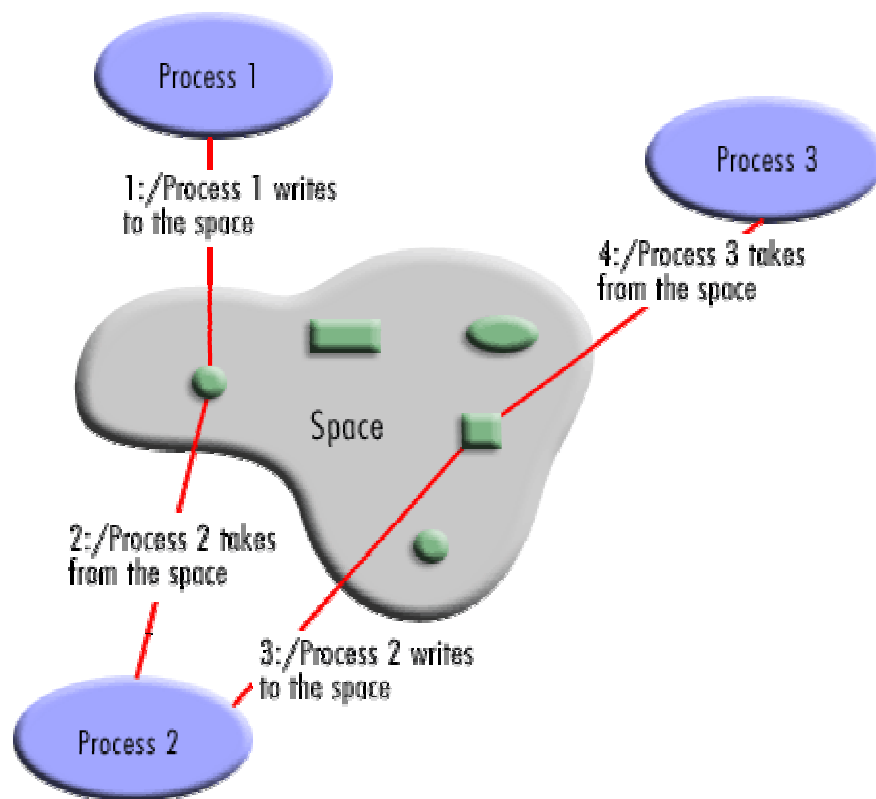


Figure 1

To further clarify the space concept, let me present a more detailed example. A common encryption method is the use of "one-way" functions, which take an input and like any other function generate an output. The distinguishing feature of such functions is that it is extremely difficult to compute the input that was given to the function to get the output, i.e. to compute the inverse of the function. Hence, the term "one-way" function. So, instead of trying to figure out the inverse of the function to get the input required for the given output, an easier way may be to take all possible inputs and compute the output for each input. When we get an output that matches the one we have, we have found the right "input". But this can be extremely time consuming given the vast number of possible inputs. Let's assume that passwords cannot be more than four characters in length and only alphanumeric ASCII characters⁸ are used. This gives us 14776336 possible passwords⁹. Furthermore, let's use the "brute" force technique to break the password. Assume that the main program breaks the input set into 16 pieces and puts each piece along with the encrypted password in the space. The password-breaking programs are watching the space for such pieces and each available program immediately grabs a piece and starts working. The programs continue till there are no more such pieces available or until the password has been broken. If the password is broken, the breaking program puts the solution in the space, which is picked up by the main program. The main program then proceeds to pick up all the remaining pieces, since it has already found the solution it needs. *The main program never knew how many*

⁸ Hence the character set is [0?9, a?z, A?Z]

⁹ Number of passwords = $(62)^4 = 14776336$

password-breaking programs were available nor did it know where these were located. The password-breaking programs had no knowledge about one another or about the main program. If there were 16 password-breaking programs available and each one was on a separate machine, we would have had 16 machines working on breaking the password simultaneously! Also, to add new password-breaking programs, no change to any configuration of the system is required. This is why spaces are so good for fault-tolerance, load balancing, and scalability.

As seen above, spaces provide an extremely powerful concept/mechanism to decouple cooperating or dependent systems. The concept of a space is not new. "Tuple spaces" were first described in 1982, in context of a programming language called "Linda". Linda consisted of "tuples", which were collections of data grouped together, and the "tuple space", which was the "shared blackboard" from which applications could place and retrieve tuples. However, the concept never gained much popularity outside of academia. Today, spaces may be an elegant solution to many of the traditional distributed computing dilemmas. In fact in recognition of this fact, JavaSoft has created its own implementation of the space concept called "JavaSpaces" and IBM has created "T Spaces", which is much more lightweight, functional, and complex than JavaSoft's JavaSpaces

We are now in a position to describe some of the key characteristics of a space:

1. *Spaces provide shared access.*

A space provides a network-accessible "shared memory" that can be accessed by many shared remote/local processes concurrently. The space handles all issues regarding concurrent access, allowing the processes to focus on their task at hand. At the very least spaces provide processes with the ability to place and retrieve "things". Some spaces also provide the ability to read/peek "things", i.e. to get the "thing" without actually removing it from the space, thus allowing other processes to access it as well.

2. *Spaces are persistent.*

A space provides reliable storage for processes to place "things". These "things" may outlive the processes that created them. This allows the dependent/cooperating processes to work together even when they have non-overlapping lifecycles. This boosts the fault-tolerance and high availability capability of distributed systems.

3. *Spaces are associative.*

Associative lookup provides processes a way to "find" the "things" that they are interested in. Since many processes may be using/sharing the same space, there may be many different types of "things" in the space. It is important that processes are able to get the things that they require without needing to filter out all the "noise" themselves. Spaces allow this by allowing processes to define filters/templates that instruct/direct the space to "find" the right "things" for that process.

These are just a few key characteristics of spaces. Many commercial space implementations, such as the ones from JavaSoft and IBM, have additional characteristics such as the ability to perform "transacted" operations on the space.

Creating your own space implementation

As discussed above, there are several commercial implementations of spaces available in the market. However, I can think of some reasons for creating your own, such as:

- Budget constraints *may* be a big reason. Commercial space implementations tend to be fairly expensive primarily because of the limited number of space implementations available.
- The functionality offered by a commercial implementation may just be too much for the job at hand. Not only may this result in a larger learning curve, it may even slow your application down due to the sheer size of the memory footprint.

- Finally, it's always fun creating your own implementation^{10*}.

In this section, I will show you how to implement your own space, which I call "QSpace". This discussion will be based on an actual implementation that I did at Online Insight¹¹, where I am a Lead Systems Architect. A version of this implementation is a core part of the architecture in some of Online Insight's products, which are successfully deployed in the real-world. At Online Insight, we decided to create our own implementation of a space. The primary reason for our decision was our limited set of requirements. These requirements are summarized as follows:

- The space must support shared access.
- The space must be fault-tolerant. For example, it may persist "messages put in it until they are taken out.
- The space must provide the ability to specify a filtering template.
- The space must allow one message in the space to be accessed by only one process/application at a time i.e. we do not support the "read/peak" operation.
- The space must perform and scale well under load¹².
- The space must not impose a limitation to what you can put in it¹³.
- The space must not impose size limitations on what you put in it¹⁴.

Note that the first three requirements are also key characteristics of a space.

At the same time while we were considering creating a space-based communication infrastructure, we were also evaluating message queue type software, more specifically, Java Message Service (JMS) implementations and we realized that we could actually build our space facility on top of one of these queues. By using message queues that expose a JMS interface, we allow ourselves the flexibility to switch vendors of message queues in case we discover that they do not meet our scalability requirements, or if a better performing message queue comes along later. This separation of implementation from interface is an important design pattern¹⁵.

Figure 2 shows a high-level class diagram of the complete system. The class `QSpaceImpl` is the space implementation. This class implements the `QSpace` interface that defines the behavior semantics of the space. The diagram also shows a very important interface – `QSpaceFactory` – and three classes that implement it. These three classes are used to obtain the initial queue connection factory for three different JMS providers. Let's take a look at the `QSpaceFactory` interface first.

¹⁰ Take this with a grain of salt. This comment is not meant to stir up the whole "Buy Vs Build" debate.

¹¹ Online Insight is an eCRM (electronic Customer Relationship Management) technology company with integrated online guided selling and customer insight solutions that help sellers to sell in a way that motivates buyers to buy.

¹² For requirements fanatics, this may be a little bit too vague.

¹³ Unlike JavaSpaces for example, which only allows Java objects that implement the `jini.net.Entry` interface to be put in the space.

¹⁴ Note however that the underlying hardware, e.g. Disk space, available memory, etc. may impose a limitation.

¹⁵ See the Bridge design pattern in Design Patterns, Elements of Reusable Object-Oriented Software, Gamma et al.

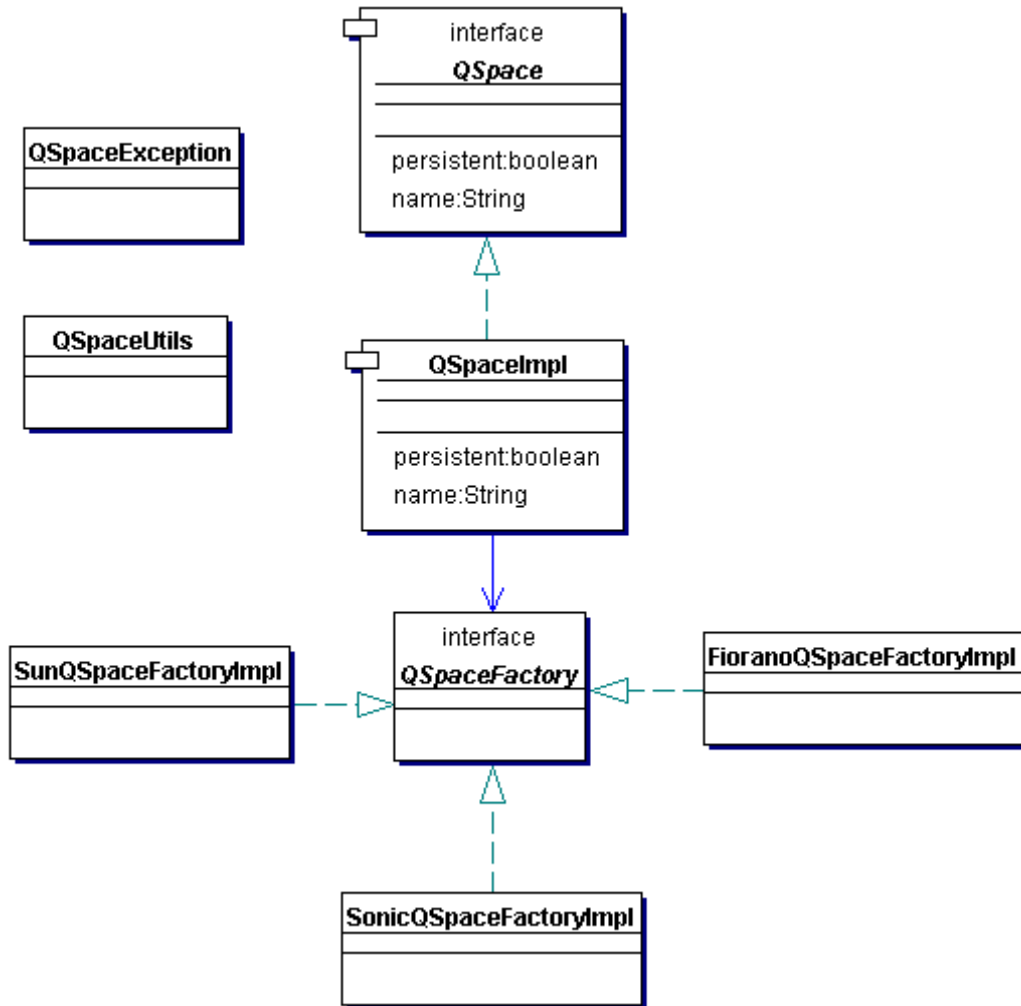


Figure 2: A class diagram for the QSpace implementation

Getting the Initial Queue Connection Factory

Remember that the JMS specification does not define a standard way of getting the initial queue connection factory, which is an instance of a class that implements the `javax.jms.QueueConnectionFactory` interface. As a result, each vendor that provides a JMS compliant messaging product must define their own way of allowing clients to get this initial queue connection factory.

In all the examples discussed in the previous chapter, I was tied to Sun's Java Message Queue as far as getting the initial queue/topic connection factory. However, as I've emphasized throughout the book, this is not a desirable situation to be for a commercial product, or piece of a commercial product¹⁶, such as for QSpace. **QSpace uses a JMS compliant provider's point-to-point messaging style.** In order to be able to switch out message queues, I have defined an interface, `QSpaceFactory`, as follows:

```

public interface QSpaceFactory {
    public javax.jms.QueueConnectionFactory
    getQueueConnectionFactory(java.util.Properties props)
}
  
```

¹⁶ Not that Sun is a bad company to be tied to *

```
        throws javax.jms.JMSEException;
    }
}
```

This interface has one method, `getQueueConnectionFactory`, that returns the initial queue connection factory. This method gets an instance of a `java.util.Properties` object as its parameter, which contains all the vendor specific messaging product information that the class implementing this interface will require to get the queue connection factory. One such class that implements the `QSpaceFactory` interface is created for every vendor's messaging product that is to be supported. For simplicity I will refer to such a class as the *factory class*. In this chapter, I will show you three such classes, one for Sun Microsystem's Java Message Queue, one for Progress Software's SonicMQ, and one for Fiorano's FioranoMQ.

QSpaceFactory Implementation for Sun Microsystem's Java Message Queue

Let's look at the implementation of the class, `SunQSpaceFactoryImpl`, for Sun Microsystem's Java Message Queue product first.

```
public class SunQSpaceFactoryImpl implements QSpaceFactory {
    public javax.jms.QueueConnectionFactory
        getQueueConnectionFactory (java.util.Properties
            props) throws javax.jms.JMSEException {
    javax.jms.QueueConnectionFactory factory =
        new com.sun.messaging.QueueConnectionFactory();
        return(factory);
    }
}
```

In this case, the queue connection factory returned is an instance of the class `com.sun.messaging.QueueConnectionFactory`, which is provided by Sun as part of their messaging product.

Note that the properties object passed in as a parameter is not used in this case. This implementation assumes that the "message router" is running on the same machine as the space. The class `com.sun.messaging.QueueConnectionFactory` has an alternate version of the constructor if the "message router" is actually running on a different machine. This constructor takes a `java.lang.Object` as its parameter as shown below:

```
public javax.jms.QueueConnectionFactory
    QueueConnectionFactory(java.lang.Object config)
```

The `config` parameter can take two forms: a string array and a `java.util.Properties` instance. The main configuration parameters are:

- "-s": defines on what host the router is running (default is localhost).
- "-n": defines the subnet on which the application and router communicate. If the number that follows `-n` is positive then it is a subnet identifier and added as an offset to a base communication port number of 9312. The default subnet is 0. For example if `"-n 2"` is specified then the subnet is 2 and the port is 9314. If the number that follows `-n` is negative then it is read as a positive value explicitly identifying the communication port number rather than a subnet. For example if `"-n-4096"` is specified then the port is 4096.

To pass these parameters as a `java.util.Properties` instance, you would do something along these lines:

```
Properties configProps = new Properties();
configProps.put("-s", "Newton");
```

```
configProps.put("-n", "-4096");
```

To pass these parameters as a string, you would do something along these lines:

```
String configString = "-- -sNewton -n-4096";
```

The "--" is important and must be the String element preceding the parameters.

In both the above cases, I am informing Sun's JMS provider that the "message router" is running on the host "Newton" and is available at port 4096.

The information necessary for creating the `config` parameter can be made available in the properties object passed in as the parameter to the `getQueueConnectionFactory` method. I will discuss how this can be done later in this chapter in the section "The QSpace Properties File".

QSpaceFactory Implementation for Progress Software's SonicMQ

Now let's look at the implementation of a similar class, `SonicQSpaceFactoryImpl`, for Progress Software's SonicMQ messaging product.

```
public class SonicQSpaceFactoryImpl implements QSpaceFactory {
    public javax.jms.QueueConnectionFactory
        getConnectionFactory(java.util.Properties props)
        throws javax.jms.JMSEException {
        javax.jms.QueueConnectionFactory factory =
            new progress.message.jclient.QueueConnectionFactory(
                props.getProperty("SpaceBroker"));
        return(factory);
    }
}
```

Progress also provides a class, `progress.message.jclient.QueueConnectionFactory`, which encapsulates the queue connection factory. However, unlike the case with Sun, this class always needs a parameter that tells it which "message broker" to connect to, even if the broker is on the same host. For example, a parameter value of "localhost:2506" implies that the broker is available on the local machine at the port 2506. Our implementation of `QSpaceFactory` gets this information from the properties object passed in as a parameter. I will discuss how this property actually got into the properties object, later.

QSpaceFactory Implementation for Fiorano's FioranoMQ

Finally, let's look at the implementation of the same class, `FioranoQSpaceFactoryImpl`, for Fiorano's messaging product.

```
import fiorano.jms.rtl.*;
public class FioranoQSpaceFactoryImpl implements QSpaceFactory {
    public javax.jms.QueueConnectionFactory
        getConnectionFactory(java.util.Properties props)
        throws javax.jms.JMSEException {
        fiorano.jms.rtl.FioranoInitialContext ic = null;
        ic = new fiorano.jms.rtl.FioranoInitialContext();
        ic.bind ();
        javax.jms.QueueConnectionFactory factory =
            (javax.jms.QueueConnectionFactory)
            ic.lookup(props.getProperty("QCFactoryName").trim());
        ic.dispose();
        return(factory);
    }
}
```

A major difference here is that Fiorano doesn't provide any [documented] way of directly creating a class that implements the `javax.jms.QueueConnectionFactory` interface, as in the other

two cases discussed above. Instead, we start by creating an instance of a `FioranoInitialContext` object. This object is used to *bind* i.e. connect to the server to create a special connection that can be used to lookup the queue connection factory (and other administered objects). The call `ic.bind()` connects to the FioranoMQ server on the local machine by default. If you wish to connect to a FioranoMQ server on a remote machine, you must use the call `ic.bind(InetAddress serverName, int port)`, which connects to the server at the supplied IP address and port number. Now, we can lookup the queue connection factory object using the `lookup` method on the context. This factory is the object that will be returned to the caller. The `lookup` method takes a string as parameter and looks up a database of administered objects on the Fiorano JMS Server for an object whose name matches the string. The name of the queue connection factory passed to the `lookup` method is obtained from the properties object. Once again, how this property got into the properties object is discussed later. Before returning to the caller, though, we clean up by calling the `dispose` method on the context object.

At this point, a discussion of the mysterious "properties object" passed into the `getQueueConnectionFactory` call is warranted.

The QSpace Properties File

Each space is configured through a properties file. A properties file is a file containing name/value pairs in the form, `name = value`. Each new line indicates a new property. Comments begin with a `"#"` character in the first column of a line.

One property in this properties file is the fully qualified¹⁷ name of the class that implements the `QSpaceFactory` interface. As we already know, there is one such class for each JMS implementation supported by the space. By doing this, changing the underlying message queue used by the space is simply a matter of changing the name of the Java class in the properties file for the space. Therefore, if one vendor's message queue does not live up to our expectations we can quickly switch to another one.

The properties in the properties file can be categorized into two categories: 1) properties used by the space implementation itself and 2) properties used by the factory class, such as the `SunQSpaceFactoryImpl` class discussed previously. When the space implementation calls the `getQueueConnectionFactory` method on the factory class, it passes a properties collection to the method. This properties collection contains all the properties in the properties file. This allows the factory class to get at any and all the configuration information that it needs.

An example properties file that I use is shown below:

```
# Filename: qspace.properties

QSpaceName=ModiSpace
AllowFilter=true
Persistent=false

Debug=true

# The factory to use to get the initial Connection Factory
#QSpaceFactory=SonicQSpaceFactoryImpl
#QSpaceFactory=SunQSpaceFactoryImpl
QSpaceFactory=FioranoQSpaceFactoryImpl

# Sonic MQ specific property
SpaceBroker=localhost:2506

# Fiorano MQ specific property
QCFactoryName=primaryqcf
```

¹⁷ name of the class with the package names included in dot notation

Don't worry if a few of the properties don't make sense right now. I'll discuss each one below.

The "QSpaceFactory" Property

First let's focus on the `QSpaceFactory` property. As discussed earlier, this property instructs the space on which factory class to use. Currently, the space is configured to use Fiorano's messaging product. To use another messaging product, for example, Progress' SonicMQ, simply comment out the current `QSpaceFactory` property and uncomment the `QSpaceFactory` property corresponding to the `SonicQSpaceFactoryImpl` class as follows:

```
# The factory to use to get the initial Connection Factory
QSpaceFactory=SonicQSpaceFactoryImpl
#QSpaceFactory=SunQSpaceFactoryImpl
#QSpaceFactory=FioranoQSpaceFactoryImpl
```

The `SpaceBroker` property is specific to SonicMQ and is only used by the `SonicQSpaceFactoryImpl` class. The space does not know or care about this property. Similarly, the `QCFacoryName` property is specific to Fiorano.

Let's return to the discussion that we left off about adding properties that the `config` object for Sun's Java Message Queue would need to connect to a "message router" on another machine, "Newton" on port 4096. Simply add a property as shown below to the properties file:

```
SunConfiguration=-- -sNewton -n-4096
```

And modify the `SunQSpaceFactoryImpl` class as follows:

```
public class SunQSpaceFactoryImpl implements QSpaceFactory {
    public javax.jms.QueueConnectionFactory
        getConnectionFactory(java.util.Properties props)
        throws javax.jms.JMSEException {
    javax.jms.QueueConnectionFactory factory =
        new com.sun.messaging.QueueConnectionFactory(
            props.getProperty("SunConfiguration"));
        return(factory);
    }
}
```

That's it! Pretty easy.

The "QSpaceName" Property

Each space instance has a unique name that identifies that instance. This name corresponds to the `QspaceName` property in the properties file. In the properties file shown above the space name is "ModiSpace".

The "Persistent" Property

Each space can also be configured to be persistent or not. If a space is persistent, then each message placed inside that space will be persisted to permanent¹⁸ storage until it is taken out. This provides the space with its fault tolerance. However, not all spaces will require this level of fault tolerance and if so, the persistence will only be a bottleneck for those spaces. So, the persistence can also be turned off on a space by space basis. This is achieved via the `Persistent` property in the properties file. If the value of this property is "true", the space instance will be persistent. In the properties file shown above the space is not persistent.

The "AllowsFilter" Property

¹⁸ at least more permanent than RAM. Examples include writing out to a database on the disk.

Each space can also be configured to allow filtering or not. Filtering allows clients of the space to specify the criteria (i.e. message selector in our case) that must be satisfied for a message to be returned to that client. However, if this feature is not required for a particular instance of a space, it will be a performance bottleneck. So, once again, I have provided the ability to turn this feature on/off on a space by space instance basis. This is achieved via the `AllowFilter` property in the properties file. If the value of this property is "true", the space instance will support filtering. In the properties file shown above the space allows filtering.

The "Debug" Property

The properties file also includes a `Debug` property. If the value of this property is "true" then "informational" messages will be seen on the console window of the space. These messages are useful to trace through and understand the workings of the space. In the properties file shown above "debugging" is turned on.

Note:

Our implementation of the space gains all of its persistence and filtering capabilities from the underlying messaging product. The space relies on the messaging product's Point-to-Point messaging support and is the only client of the message queue.

The QSpace Interface

I am a big fan of "interface-based" programming. Rather than being tied to an implementation, interface-based programming ties you to a firm behavioral contract between your program and the interface. The primary advantage of this is that it allows you to swap implementations out with better versions without changing all the other programs that depend on it. Isn't this exactly what JMS allows us to do? Yes, it is and as I've mentioned several times in this book, that's one of the reasons it's so powerful.

Each instance of QSpace will implement the following interface

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Properties;

public interface QSpace extends Remote {

    // Methods to put "things" in the space
    public void write(byte[] blob)
        throws QSpaceException, RemoteException;
    public void write(byte[] blob, Properties filters)
        throws QSpaceException, RemoteException;

    // Methods to get "things" out of the space
    public byte[] take()
        throws QSpaceException, RemoteException;
    public byte[] take(String filterAsString)
        throws QSpaceException, RemoteException;
    public byte[] take(long timeout)
        throws QSpaceException, RemoteException;
    public byte[] take(String filterAsString, long timeout)
        throws QSpaceException, RemoteException;

    // Methods to get the space characteristics
    public String getName() throws RemoteException;
    public boolean isPersistent() throws RemoteException;
    public boolean allowsFilter() throws RemoteException;

    // Shutdown method
    public void shutdown() throws RemoteException;
};
```


Notice how the `write` and `take` methods take an array of bytes as their parameter. Hence anything that can be represented as a stream of bytes, such as a CORBA object IOR, a serialized Java object, an XML document, etc. can be stored in the space and retrieved. This is in tune with our requirement that the space not limit what can be put into it. Also, note that all versions of the `write` and `take` methods throw a `QSpaceException` in addition to the standard `RemoteException` that is required to be thrown by all RMI remote methods. The definition of `QSpaceException` is shown below:

```
public class QSpaceException extends java.lang.Exception {
    public QSpaceException(String s) {
        super(s);
    }
}
```

Let's look at the semantics of each method before moving on to the actual implementation.

The write methods

There are two version of this method. These methods are used by a client of the space to put a message into the space. If the space supports filters the second version may be used to specify filtering properties and their values. Other clients can then filter based on these filtering properties. However, if the space does not support filtering, a `QSpaceException` must be thrown with the message *"Filters not Allowed in this space"*.

The take methods

There are four versions of this method. These methods range from the simplest one, which merely blocks till any message becomes available in the space, to the version that accepts both a time out value and a filtering string. However, if the space does not support filtering, a `QSpaceException` must be thrown with the message *"Filters not Allowed in this space"*, if one of the versions that take a filtering string is called. Also, if the method times out, a `QSpaceException` exception with the message *"No message to Take"* must be thrown.

The getName, isPersistent, and allowsFilter methods

The `getName` method returns the name of the space instance. The `isPersistent` and `allowsFilter` methods return `true` if the space is persistent and supports filtering respectively.

The shutdown method

This method is called to shutdown the space. The shutdown is clean, which means the registration with the RMI registry is removed and all JMS related cleanup is performed.

The QSpace Implementation

The `QSpace` implementation implements the `QSpace` interface described in the previous section. The space implementation itself is an RMI object. During start-up each space installs itself in the RMI registry running on its machine at the default port i.e. 1099. The space uses its name, such as "ModiSpace", as the binding name in the RMI registry. Thus interested applications/processes can find a space by using the well-known name from the RMI registry.

The main method

The `main` method of the implementation does the following, in order:

1. Install a security manager. All RMI applications must have a security manager installed. This is done as follows:

```
if( System.getSecurityManager() == null )
    System.setSecurityManager(new RMISecurityManager());
```

2. Get the configuration properties from the specified properties file. The first step is to get the name of the properties file as follows:

```
String propertiesFile;  
propertiesFile=QSpaceUtils.getPropertiesFile(args))
```

QSpaceUtils is a class that provides five static methods. These methods can be used by clients of QSpace instances i.e. spaces and by the space implementation itself as shown above. The five methods are: `getBytes`, `getObject`, `getPropertiesFile`, `getProperties`, and `getParam`. I will discuss the methods of this class as and when they are used. I will use all five of these, so all five methods will be discussed. The constructor of the QSpaceUtils class is private, so instances of this class cannot be created. The `getPropertiesFile` method of this class is shown below:

```
// if one of the args is -PropertiesFile=XYZ, return XYZ  
public static String getPropertiesFile(String[] args) {  
    int numArgs = args.length;  
    if( numArgs > 0 ) {  
        for( int i=0; i<numArgs; i++ ) {  
            int index;  
            // Find a command line arg that begins with  
            // -PropertiesFile=  
            index = args[i].indexOf("-PropertiesFile=");  
            // not found, then move onto the next "arg"  
            if( index == -1 )  
                continue;  
            index = index + new String("-PropertiesFile=").length();  
            String propertiesFilename = args[i].substring(index).trim();  
            return(propertiesFilename);  
        }  
    }  
    // No properties filename to return  
    return(null);  
}
```

The next step is to get the properties from this properties file, which is done as follows:

```
Properties props = QSpaceUtils.getProperties(propertiesFile,args);
```

Once again, `getProperties` is a static method of the QSpaceUtils class and is shown below:

```
// Return a Properties collection that loads all properties  
// from the  
// given properties file and then overrides any of those  
// properties  
// with values specified on the command line.  
// For example, if the properties file contained  
// color=blue  
// and the command line had a paramter  
// -color=red  
// then the returned properties collection would map "color" to  
// "red"  
//  
public static Properties getProperties(  
    String propertiesFile, String[] args) {  
    Properties props = new Properties();  
    try {  
        // Read the file into the properties collection  
        FileInputStream in = new FileInputStream(propertiesFile);  
        props.load(in);  
    }  
}
```

```

    in.close();
}
catch( java.io.IOException e ) {
    e.printStackTrace();
    return(null);
}

// No overriding arguments provided, so we're done.
if( args == null )
    return(props);

// Go thru each "arg"
int numArgs = args.length;
if( numArgs > 0 ) {
    for( int i=0; i<numArgs; i++ ) {
        int index;

        // if the arg does not start with "-" ignore it.
        index = args[i].indexOf("-");
        if( index == -1 )
            continue;

        // if the arg does not contain "=" ignore it.
        index = args[i].indexOf("=");
        if( index == -1 )
            continue;

        String propertyName;
        propertyName = args[i].substring(1,index).trim();

        // Special case
        if( propertyName.equals("PropertiesFile") )
            continue;

        if( props.containsKey(propertyName) ) {
            String propertyValue;
            propertyValue = args[i].substring(index+1).trim();
            props.put(propertyName,propertyValue);
        }
        else
            System.err.println("Unknown Property: " + propertyName);
    }
}

// Finally, return the properties collection.
return(props);
}

```

3. If the "Debug" property's value is "true", set the `_debug` static variable to true as follows:

```

    if( props.getProperty("Debug").trim().equals("true") ) {
        _debug = true;
        System.out.println("Debugging is turned on...");
    }

```

4. Create an instance of the `QSpaceImpl` class passing in the properties collection that we got in step 2. Let's look at the `QSpaceImpl` constructor. The constructor first calls the default base constructor and keeps a reference to the properties for future use. It then checks to if this instance of the space is persistent as follows:

```

// Is this space persistent?
if( props.getProperty("Persistent").equalsIgnoreCase("false") )
    persistenceMode = javax.jms.DeliveryMode.NON_PERSISTENT;
else

```

```
persistenceMode = javax.jms.DeliveryMode.PERSISTENT;
```

The `persistenceMode` variable will be used in the `send` method call on the `javax.jms.QueueSender` object. The constructor also checks to see if this instance of the space supports filtering as follows:

```
// Does this space allow filters?
if(props.getProperty("AllowFilter").equalsIgnoreCase("false"))
    bAllowFilter = false;
else
    bAllowFilter = true;
```

The constructor also gets the queue connection factory by getting the appropriate `QSpaceFactory` class and calling `getQueueConnectionFactory` on it passing in the properties collection.

```
// Get the queue connection factory...
QSpaceFactory sf = (QSpaceFactory)(Class.forName(
    props.getProperty("QSpaceFactory")).newInstance());
queueConnectionFactory = sf.getQueueConnectionFactory(props);
```

I've already discussed what happens at this point in great detail in the section "Getting the Initial Queue Connection Factory" earlier in this chapter.

5. Register the space in the RMI registry as follows:

```
// Make the space available in the RMI
// registry on [the default] port 1099
Naming.bind(theSpace.getName(),theSpace);
```

The above code snippet shows just how easy RMI is to use. As mentioned earlier, the space is made available in the local RMI registry available at port 1099. The space is bound in the registry with the name of the space itself (very convenient).

6. Now wait for the space to shutdown as shown below:

```
// Wait here until the space is asked to shutdown...
synchronized(theSpace) {
    while( theSpace.running() ) {
        try {
            theSpace.wait();
        }
        catch( Exception e ) {
        }
    }
}
```

When the space shuts down (i.e. the shutdown method is called) it will call the `notify` method on itself, which will wake up the main thread. At that point the `running` method on the space will return false and so the while loop will no longer be executed.

7. Unregister the space from the RMI registry and exit as follows:

```
// remove all traces of the space...
try {
    Naming.unbind(theSpace.getName());
}
catch( Exception e ) {
    e.printStackTrace();
}
```

```
// done
System.exit(0);
```

The QSpace interface implementation

Finally, let's look at the implementation of the QSpace interface, starting with the easiest methods first.

The isPersistent, allowsFilter, and getName methods

These methods are so easy they need no explanation at all!

```
public boolean isPersistent() throws RemoteException{
    return(persistenceMode == javax.jms.DeliveryMode.PERSISTENT);
}

public boolean allowsFilter() throws RemoteException{
    return(bAllowFilter);
}

public String getName() throws RemoteException{
    return(props.getProperty("QSpaceName"));
}
```

The shutdown method

Next, let's look at the shutdown method:

```
synchronized public void shutdown() throws RemoteException
{
    // if we've already shutdown, just return.
    // This could happen if two clients asked to shutdown
    // the space at the same time.
    if(bShutdownFlag)
        return;
    try {
        if( _debug )
            System.out.println("Shutting down the space " +
                               getName() + "...");

        queueSender.close();
        queueReceiver.close();
        queueSession.close();
        queueConnection.close();
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
    finally {
        // Set the shutdown flag to true...
        bShutdownFlag = true;
        // and notify the "main" thread of the shutdown.
        notifyAll();
    }
}
```

This method is synchronized to prevent multiple threads trying to shutdown at the same time. The method first checks if the space has already been shutdown and if it has, simply returns. This method closes the queue sender, the receiver, the session, and the connection. It then sets the shutdown flag, `bShutdownFlag`, to `true` and notifies all threads of the shutdown by calling `notifyAll`. Remember, this will also wake up the main thread, which will then call the running method shown below:

```
// if the space has not shutdown then return true.
public boolean running() {
    return(!bShutdownFlag);
}
```

```
}
```

The write methods

Now, let's look at the `write` methods. There are two versions of this method. The first version only takes an array of bytes. The implementation of this method delegates to the second version as shown below:

```
write(blob,null);
```

This second version also takes a properties collection of filtering properties, which in this case will be `null`. The steps taken by this method are outlined below:

1. Make sure that the space hasn't been asked to shutdown. If it has, the method raises an exception as follows:

```
if(bShutdownFlag)
    throw new QSpaceException(
        "This space has been shutdown and must be restarted.");
```

2. If a properties collection of filtering properties is specified, the method checks if filters are allowed in this space as follows:

```
if( (filters != null) && !bAllowFilter )
    throw new QSpaceException(
        "Filters not Allowed in this space");
```

3. Obtain the lock associated with this instance of the space as follows:

```
synchronized(this)
```

This is very important because we only want one thread accessing the queue at a time. Then I repeat step 1 again. This is to make sure that the space has not shutdown in the time it took to acquire the lock and is very important. This step is so important that it has been identified as a design pattern in itself and is known as the *"Double Checked Locking"*¹⁹ pattern.

4. Create the message as follows:

```
// Create a ByteMessage instance and store the message
// with its length in this message
ByteMessage msg = queueSession.createByteMessage();
msg.writeInt(blob.length);
msg.writeBytes(blob);
```

Remember from chapter 5, `ByteMessages` are useful in situations where one needs to read in raw data, for example, from a disk file, and transfer it "as is" (without any conversion at all, such as Big Endian/little Endian, etc.) to another machine and/or location. In our case, we are receiving raw byte data from the client that we need to store and allow another (or same) client get access to, possibly from another machine.

5. Add any filtering properties if required:

```
// Add all the filtering properties to the message
if( filters != null ) {
    java.util.Enumeration enum = filters.keys();
    while( enum.hasMoreElements() ) {
        String name = (String)enum.nextElement();
```

¹⁹ "Double-Checked Locking" by Doug Schmidt and Tim Harrison in *Pattern Languages of Program Design 3*

```

msg.setStringProperty(name,
                      filters.getProperty(name).trim());
    if( _debug )
        System.out.println("Added property ["
            + name + "=" + filters.getProperty(name) + "]);
    }
}

```

All filtering properties are added in as "String" properties.

6. Put the message in the queue and notify all threads that a new message is available. The reason for this will become obvious as we look at the code for the `take` methods. Note that the `persistenceMode` member variable is used to indicate the persistence status. The priority of the message is set to 9 (highest) and the message will not timeout. A better implementation of the space may have a configuration property specified in the properties file to set a timeout value. This is a minor change and will not be shown here.

```

queueSender.send(msg,persistenceMode,9,0);

// Important:
// Let all threads waiting for "something" to "take" know
// that their wait may be over...
notifyAll();

```

The take methods

Finally, let's look at the `take` methods. There are four versions of these methods. The most versatile of the four takes two parameters, a time out in milliseconds and a filter string. The other three simply delegate to this version. For example, the version of `take` that has no parameters delegates as follows:

```
take(null,-1);
```

So, let's look at the most versatile of the four `take` methods. The steps taken by it are described below:

1. Make sure that the space hasn't been asked to shutdown. If it has, the method raises an exception as follows:

```

if(bShutdownFlag)
    throw new QSpaceException(
        "This space has been shutdown and must be restarted.");

```

2. If a filter string is specified, the method checks if filters are allowed in this space as follows:

```

if((selection != null) && !bAllowFilter )
    throw new QSpaceException(
        "Filters not Allowed in this space");

```

3. Make sure that the time out value specified is valid as shown below:

```

if( timeout < -1 )
    throw new QSpaceException("Invalid Timeout specified");

```

This value is a time duration in milliseconds, with `-1` being a special value that indicates an infinite wait.

4. Obtain the lock associated with this instance of the space as follows:

synchronized(this)

This is very important because we only want one thread accessing the queue at a time. Then I repeat step 1 again. This is to make sure that the space has not shutdown in the time it took to acquire the lock and is very important. As mentioned earlier this is the *Double Checked Locking* pattern.

5. Based on the duration specified, compute the "end time", which is the current time plus the duration. Note that this does not take into account the time spent during the processing in steps 1 through 4. The design of the space is such that acquiring the lock should be a minimal wait, since no thread holds the lock for any more time than absolutely necessary. Also, note that this is not a time critical application.
6. If this space allows filters, we must call the private `reconnect` method. This method is shown below:

```
// close queue receiver and recreate...
private void reconnect(String selection) throws
QSpaceException
{
    try {
        queueReceiver.close();
        if( selection.equals("") )
            queueReceiver =
                queueSession.createReceiver(queue);
        else
            queueReceiver =
                queueSession.createReceiver(queue,
                                            selection);
    }
    catch( Exception e ) {
        e.printStackTrace();
        throw new QSpaceException(
            "Exception, could not connect to the space: "
            + e.getMessage());
    }
}
```

The `reconnect` method closes the existing queue receiver and recreates it with the proper selection string if necessary. If the space does not support filters then this additional step is not necessary, since a filter string will never be specified and so the queue receiver will never need to be changed. Note that we have to close the existing queue receiver before creating a new one because **multiple receivers are not allowed in the point-to-point model**.

JMS Note

Actually, the JMS does not specify anything on whether the point-to-point messaging style allows multiple receivers. So individual JMS providers are free to do whatever they desire. For portability, it is safer to assume that this is not possible.

7. Try to get a message from the queue by calling the non-blocking receive method on the queue as shown below:

```
BytesMessage msg =
    (BytesMessage)queueReceiver.receiveNoWait();
```

If a message is found then the method returns it to the caller.

8. If no message was found then the method waits if necessary. This happens either if the current time [at this instant] is less than [before] the end time calculated in step 5, or if the time out value specified was -1. Otherwise, the method raises an exception with the exception message as "No message to Take" as required by the QSpace interface contract.
9. If step 8 resulted in a wait, then this wait can end either by a time out of the wait or by another thread calling notifyAll. In either case, the sequence from step 6 onwards gets repeated.

The entire implementation of the space is shown below:

```
import java.util.Properties;
import java.util.HashMap;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Naming;
import javax.jms.*;

public class QSpaceImpl extends UnicastRemoteObject implements QSpace {

    public static void main(String args[]) {

        // Install a security manager
        if( System.getSecurityManager() == null )
            System.setSecurityManager(new RMISecurityManager());

        // Get the configuration properties
        Properties props = getProperties(args);
        if( props == null )
            System.exit(-1);

        // Is debugging on?
        if( props.getProperty("Debug").trim().equals("true") ) {
            _debug = true;
            System.out.println("Debugging is turned on...");
        }

        // Create the space...
        QSpaceImpl theSpace = null;
        try {
            theSpace = new QSpaceImpl(props);
            if( _debug )
                System.out.println(theSpace.toString());

            // Make the space available in the RMI registry on
            // [the default] port 1099
            Naming.bind(theSpace.getName(),theSpace);
            if( _debug )
                System.out.println(
                    "The space is ready and is available as '"
                    + theSpace.getName() + "' in the RMI Registry
                    on this machine at port 1099");
        }
        catch( Exception e ) {
            e.printStackTrace();
            System.exit(-1);
        }

        // Wait here until the space is asked to shutdown...
        synchronized(theSpace) {
            while( theSpace.running() ) {
                try {
                    theSpace.wait();
                }
            }
        }
    }
}
```

```

        }
        catch( Exception e ) {
        }
    }

    // remove all traces of the space...
    try {
        Naming.unbind(theSpace.getName());
    }
    catch( Exception e ) {
        e.printStackTrace();
    }

    // done
    System.exit(0);
}

public QSpaceImpl(Properties props)
    throws QSpaceException, RemoteException
{
    // important: call base class constructor.
    super();

    // keep a reference to props.
    this.props = props;

    try {
        // Is this space persistent?
        if( props.getProperty("Persistent").equalsIgnoreCase("false") )
            persistenceMode = javax.jms.DeliveryMode.NON_PERSISTENT;
        else
            persistenceMode = javax.jms.DeliveryMode.PERSISTENT;

        // Does this space allow filters?
        if( props.getProperty("AllowFilter").equalsIgnoreCase("false") )
            bAllowFilter = false;
        else
            bAllowFilter = true;

        // Get the queue connection factory...
        QSpaceFactory sf = (QSpaceFactory)(Class.forName(
            props.getProperty("QSpaceFactory")).newInstance());
        queueConnectionFactory = sf.getConnectionFactory(props);

        // Now use the queue conn. factory to
        // get the queue connection,
        // use the queue connection to get the session, and
        // use the session to get a sender and receiver
        queueConnection =
            queueConnectionFactory.createQueueConnection();
        queueSession = queueConnection.createQueueSession(false,1);
        queue = queueSession.createQueue(getName());
        queueSender = queueSession.createSender(queue);
        queueReceiver = queueSession.createReceiver(queue);
        queueConnection.start();
    }
    catch( Exception e ) {
        e.printStackTrace();
        throw new QSpaceException(
            "Exception, could not initialize the space: " +
            e.getMessage());
    }
}

// Get "displayable" information about the space
public String toString() {
    try {

```

```

String s1;
if( isPersistent() )
    s1 = "persistent";
else
    s1 = "not persistent";

String s2;
if( allowsFilter() )
    s2 = "allows filters";
else
    s2 = "does not allow filters";

StringBuffer sb = new StringBuffer(300);
sb.append("This space is called ").append(
    getName()).append(", is ");
sb.append(s1).append(", and ").append(s2);
return(sb.toString());
}
catch(Exception e)
{
}

return "";
}

// This method returns "true" as long as the space
// hasn't been asked to shutdown.
public boolean running() {
    return(!bShutdownFlag);
}

////////////////////////////////////
// Implementation of the QSpace Interface
////////////////////////////////////
public void write(byte[] blob)
    throws QSpaceException, RemoteException
{
    // delegate...
    write(blob,null);
}

public void write(byte[] blob, Properties filters)
    throws QSpaceException, RemoteException
{
    // Has the space been asked to shutdown?
    if(bShutdownFlag)
        throw new QSpaceException(
            "This space has been shutdown and must be restarted.");

    // if a filter is specified and this space does not allow
    // filters, then throw an exception.
    if( (filters != null) && !bAllowFilter )
        throw new QSpaceException("Filters not Allowed in this space");

    // only one thread at a time
    synchronized(this) {

        // The "Double Checked Pattern" in action.
        if(bShutdownFlag)
            throw new QSpaceException(
                "This space has been shutdown and must be restarted.");

        try {
            if( _debug )
                System.out.println(
                    "Writing message[" + blob.length + " bytes]...");

            // Create a BytesMessage and put the message

```

```

// in it.
BytesMessage msg = queueSession.createBytesMessage();
msg.writeInt(blob.length);
msg.writeBytes(blob);

// Add all the filtering properties to the message
if( filters != null ) {
    java.util.Enumeration enum = filters.keys();
    while( enum.hasMoreElements() ) {
        String name = (String)enum.nextElement();
        msg.setStringProperty(name,
            filters.getProperty(name).trim());

        if( _debug )
            System.out.println("Added property [" + name + "=" +
                filters.getProperty(name) + "]");
    }
}

// Send the message to the message queue.
// Note the use of the persistenceMode
// property, a priority of 9, and no timeout.
queueSender.send(msg,persistenceMode,9,0);

// Important:
// Let all threads waiting for "something" to
// "take" know
// that their wait may be over...
notifyAll();
}
catch( JMSEException e ) {
    if( _debug )
        System.err.println("JMSEException in write()");
    throw new QSpaceException(e.getMessage());
}
}
}

public byte[] take() throws QSpaceException, RemoteException
{
    // delegate
    return(take(null,-1));
}

public byte[] take(long timeout)
    throws QSpaceException, RemoteException
{
    // delegate
    return(take(null,timeout));
}

public byte[] take(String selection)
    throws QSpaceException, RemoteException
{
    // delegate
    return(take(selection,-1));
}

public byte[] take(String selection, long timeout)
    throws QSpaceException, RemoteException
{
    // Has the space been asked to shutdown?
    if(bShutdownFlag)
        throw new QSpaceException(
            "This space has been shutdown and must be restarted.");

    // if a filter is specified and this space does not allow
    // filters, then throw an exception.

```

```

if( (selection != null) && !bAllowFilter )
    throw new QSpaceException(
        "Filters not Allowed in this space");

// must have a valid timeout
if( timeout < -1 )
    throw new QSpaceException(
        "Invalid Timeout specified");

if( _debug && (selection != null) )
    System.out.println("Selection criteria: " +
        selection);

// only one thread at a time...
synchronized(this) {

    // The "Double Checked Pattern" in action.
    if(bShutdownFlag)
        throw new QSpaceException("This space has been
            shutdown and must be restarted.");

    try {
        // if a timeout has been specified then
        // compute the end time.
        long endtime = 0;
        if( timeout != -1 )
            endtime = System.currentTimeMillis() +
                timeout;

        while( true ) {

            // if this space allows filters
            // we need to call the reconnect
            // method. This method recreates
            // the receiver with the "right"
            // message selector.
            if( bAllowFilter ) {
                if( selection != null )
                    reconnect(selection);
                else
                    reconnect("");
            }

            // Any messages available?
            BytesMessage msg = (BytesMessage)
                queueReceiver.receiveNoWait();

            // Yep!, return the message.
            if( msg != null ) {
                int len = msg.readInt();
                byte[] theBytes = new byte[len];
                len = msg.readBytes(theBytes);
            }

            if( _debug ) {
                System.out.println("Took message [" + len + " bytes]...");
                if(selection != null) {
                    java.util.Enumeration e = msg.getPropertyNames() ;
                    while( e.hasMoreElements() ) {
                        String name = (String)e.nextElement();
                        if(!name.startsWith("JMS"))
                            System.out.println( name + "=" +
                                msg.getStringProperty(name));
                    }
                }
            }

            // return the bytes taken?
            return(theBytes);
        }
    }
}

```

```

        // Nope, no message available!

        // Check to see if we timed out
        // If so, then return.
        if( timeout != -1 &&
            System.currentTimeMillis() > endtime ) {
            if( _debug )
                System.out.println("No message to Take");
            throw new QSpaceException("No message to Take");
        }

        // Important:
        // "wait" will release the lock and allow
        // other threads in.
        // This is key to the proper operation of
        // the space
        try {
            if( timeout == -1 )
                wait();
            // Normally will not happen...
            else if( timeout == 0 )
                wait(1);
            else
                wait(Math.abs(endtime - System.currentTimeMillis()));
        }
        catch( InterruptedException e ) {
        }

        // This thread just woke up.
        // In the mean time, the space
        // could have been shutdown?
        if(bShutdownFlag)
            throw new QSpaceException("This space
                has been shutdown and must be restarted.");

    } // while(true) loop continues?
}
catch( JMSEException e ) {
    if( _debug )
        System.err.println("JMSEException in take()");
    e.printStackTrace();
    throw new QSpaceException(e.getMessage());
}
}
}

synchronized public void shutdown() throws RemoteException
{
    // If we've already shutdown, just return..
    if(bShutdownFlag)
        return;

    try {
        if( _debug )
            System.out.println("Shutting down the space " +
                getName() + "...");

        queueSender.close();
        queueReceiver.close();
        queueSession.close();
        queueConnection.close();
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
    finally {
        // Set the shutdown flag to true...

```

```

        bShutdownFlag = true;

        // Notify all waiting threads that the space
        // has shut down. This will also wake up the main
        // thread, which will then exit the JVM.
        notifyAll();
    }
}

public boolean isPersistent() throws RemoteException{
    return(persistenceMode == javax.jms.DeliveryMode.PERSISTENT);
}

public boolean allowsFilter() throws RemoteException{
    return(bAllowFilter);
}

public String getName() throws RemoteException{
    return(props.getProperty("QSpaceName"));
}

// Private Methods...

// Get the configuration properties
private static Properties getProperties(String[] args) {
    String propertiesFile = null;
    if( (propertiesFile=QSpaceUtils.getPropertiesFile(args)) == null ){
        System.err.println(
            "Fatal Error: You must specify a properties file");
        return(null);
    }

    System.out.println("Using Properties File " + propertiesFile);

    Properties props = QSpaceUtils.getProperties(propertiesFile,args);
    if( props == null )
        System.err.println(
            "Fatal Error: Could not get the QSpaceImpl Properties," +
            " Shutting down...");

    return(props);
}

// close all queue related stuff and recreate...
private void reconnect(String selection) throws QSpaceException
{
    try {
        queueReceiver.close();
        if( selection.equals("") )
            queueReceiver = queueSession.createReceiver(queue);
        else
            queueReceiver = queueSession.createReceiver(queue,selection);
    }
    catch( Exception e ) {
        e.printStackTrace();
        throw new QSpaceException(
            "Exception, could not connect to the space: " +
            e.getMessage());
    }
}

// Private Member Variables

private static boolean _debug = false;

private QueueConnectionFactory    queueConnectionFactory;
private QueueConnection           queueConnection;

```

```

private QueueSession      queueSession;
private QueueSender      queueSender;
private QueueReceiver    queueReceiver;
private Queue            queue;
private Properties props;
private boolean bAllowFilter;
private int persistenceMode;
private boolean bShutdownFlag = false;
}

```

If I haven't made it obvious yet, let me do so right now. At any point in time, only one thread is using the queue session, queue sender, and queue receiver objects. It may seem that this is not true since multiple threads can take from and write to the space simultaneously. But remember, the `write` and `take` methods are synchronized. Also, the `take` methods always call the `receiveNoWait` method on the queue receiver. If a message is retrieved, the method returns the message to the caller. On the other hand if no message is retrieved, the method calls the `wait` method to release its lock so that another thread can take its turn. The `write` method always finishes sending the message to the queue and returns. This logic is extremely important since the queue session, sender, and receiver objects are all single-threaded. In all your work with JMS you must make sure that you never violate this principle since it may lead to extremely hard to track down bugs.

Compiling the QSpace Application

Compiling the space requires the following:

- Setting up the right classpath
- Compiling the right QSpaceFactory class
- Generating the stubs and skeletons

For example, let's consider compiling with Progress Software's SonicMQ messaging product (on a Windows platform):

1. Set the home directory for SonicMQ

```
set JMQ_HOME=E:\Program Files\Progress_SonicMQ
```

2. Set the classpath

```
set JMQ_CPATH=%JMQ_CPATH%;%JMQ_HOME%\lib\jms.jar;
%JMQ_HOME%\lib\broker.jar;%JMQ_HOME%\lib\jndi.jar
```

3. Set the files to compile

```
set TARGET= QSpaceException.java QSpaceFactory.java QSpace.java
QSpaceUtils.java QSpaceImpl.java TestObject.java QSpaceClient.java
set TARGET= %TARGET% SonicQSpaceFactoryImpl.java
```

4. Compile the target files using the classpath JMQ_CPATH

```
javac -d . -g -classpath "%JMQ_CPATH%" %TARGET%
```

5. Create and compile the RMI stubs and skeletons using the RMI compiler

```
rmic -d . QSpaceImpl
```

Starting a QSpace

Starting a space requires the following:

- Setting up the right classpath
- Passing the right system properties to the JVM
- Specifying a properties file for the space
- Setting up a codebase
- Creating a policy file
- Starting an RMI registry at port 1099. This is done by just running the rmiregistry program that comes with the JRE. Make sure the classpath is set to nothing for that session.

My codebase is set up on "localhost" i.e. this machine, at port 9050. The following files will be needed in the codebase

- QSpace.class
- QSpaceException.class
- QSpaceImpl_Stub.class

I am also using a wide-open policy file (called policy.all) as shown below. This policy file should not be used in a production system. A policy file needs to be specified any time an application uses a security manager as in the case of RMI.

```
grant {
    permission java.security.AllPermission "", "";
};
```

Let's go through an example of starting the space with Fiorano's FioranoMQ messaging product (on a Windows platform):

1. Make sure Fiorano's FioranoMQ is running and the appropriate queue connection factory is set up along with a queue whose name corresponds to the QSpaceName property in the properties file. Refer to Fiorano's documentation for how to do these.
2. Start the RMI registry at port 1099 and a web server at port 9050.
3. Set the flags that will be passed to the Java VM

```
set FLAGS= -Djava.compiler=NONE -Djava.security.policy=policy.all
           -Djava.rmi.server.codebase=http://localhost:9050/
```

4. Set the home directory for SonicMQ

```
set JMQ_HOME=E:\Program Files\Fiorano\FioranoMQ
```

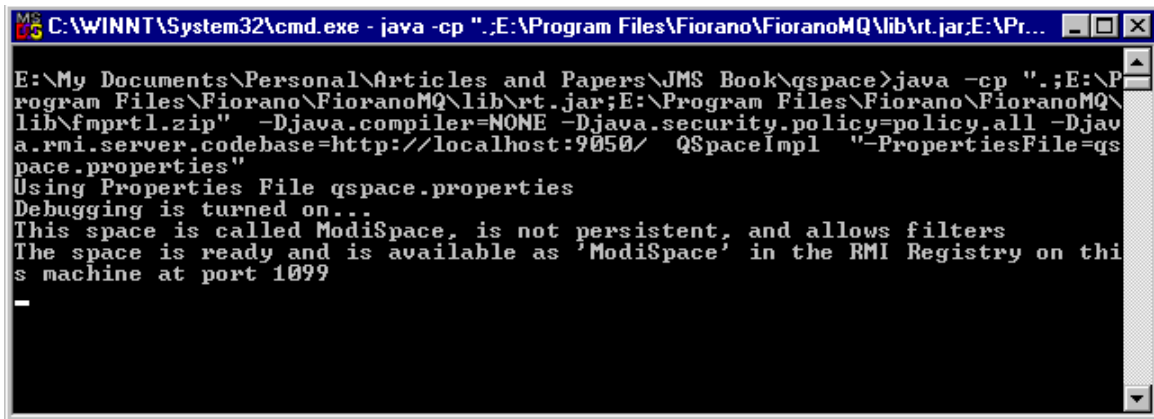
5. Set the classpath

```
set CPATH=
set CPATH=%CPATH%;%JMQ_HOME%\lib\rt.jar;%JMQ_HOME%\lib\fmprtl.zip
```

6. Start the space

```
java -cp "%CPATH%" %FLAGS% QSpaceImpl "-PropertiesFile=qspace.properties"
```

The combined results of executing these five steps are shown in figure 2.



```
C:\WINNT\System32\cmd.exe - java -cp ".;E:\Program Files\Fiorano\FioranoMQ\lib\rt.jar;E:\Pr...
E:\My Documents\Personal\Articles and Papers\JMS Book\qspace>java -cp ".;E:\P
rogram Files\Fiorano\FioranoMQ\lib\rt.jar;E:\Program Files\Fiorano\FioranoMQ\
lib\fmprtl.zip" -Djava.compiler=NONE -Djava.security.policy=policy.all -Djav
a.rmi.server.codebase=http://localhost:9050/ QSpaceImpl "-PropertiesFile=qspace.properties"
Using Properties File qspace.properties
Debugging is turned on...
This space is called ModiSpace, is not persistent, and allows filters
The space is ready and is available as 'ModiSpace' in the RMI Registry on thi
s machine at port 1099
-
```

Figure 3: Starting the QSpace using Fiorano's FioranoMQ

A Test Program

Let's look at a client of our space that can exercise every feature.

Note:

Make sure that QSpace.class, QSpaceUtils.class, and QSpaceException.class are in the classpath before running the test program.

The usage for the client application is summarized as follows:

```
java -Djava.security.policy=policy.all QSpaceClient space-name write
    [message:<message>] [filter:<properties>]
java -Djava.security.policy=policy.all QSpaceClient space-name take
    [timeout:<timeout>] [filter:<string>]
java -Djava.security.policy=policy.all QSpaceClient space-name shutdown
```

When the action is "write", the filter parameter must be the name of a properties file that specifies the filtering properties and their values. For example, the file may look like:

```
# filename: filter.properties
year=1999
manufacturer=honda
color=red
make=civic
model=ex
```

When the action is "take", the filter parameter is a string that conforms to the message selector syntax discussed in chapter 5, for example, "year='1999' And model='ex'".

The program in its entirety is shown below. While compiling this program make sure that QSpace.class is in the classpath.

```
// A comprehensive test program to test the QSpace implementation
import java.util.Properties;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.io.FileInputStream;

public class QSpaceClient {
```



```

else {
    System.err.println("Error: Invalid action.
        Action must be 'write' or 'take'.");
    System.exit(-1);
}

QSpace space = null;
try {
    space = (QSpace)Naming.lookup(spaceName);
}
catch( Exception e ) {
    e.printStackTrace();
    System.exit(-1);
}

if( action.equals("write") ) {
    try {
        if( filterProps != null )
            space.write(message.getBytes(),filterProps);
            //space.write(QSpaceUtils.getBytes(
            //new TestObject("This is the Test Object message...")),
            //filterProps);
        else
            space.write(message.getBytes());
            //space.write(QSpaceUtils.getBytes(
            //new TestObject("This is the Test Object message...")));
    }
    catch( Exception e ) {
        System.err.println(e.getMessage());
    }
}
else if( action.equals("take") ) {
    try {
        if( filterString != null )
            System.out.println(
                new String(space.take(filterString, timeout)));
            //System.out.println(((TestObject)
            //QSpaceUtils.getObject(space.take(
            //filterString,timeout))).getMessage());
        else
            System.out.println(new String(space.take(timeout)));
            //System.out.println(((TestObject)
            //QSpaceUtils.getObject(space.take(timeout))).getMessage());
    }
    catch( Exception e ) {
        System.err.println(e.getMessage());
    }
}
else if( action.equals("shutdown") ) {
    try {
        space.shutdown();
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
}

// done
System.exit(0);
}
}

```

The client program takes two mandatory parameters, which must be the first two and in a fixed order. The first parameter is the name of the space to use. The second parameter is the action to perform, which can be either "write", "take", or "shutdown". Based on this action, additional parameters can be specified in the form *parameter:value*.

To read these parameter values, the client program uses the `getParam` static method from the `QSpaceUtils` class. This method is shown below:

```
// Get the value of the "param" from the array "args"
// For example, if one of the Strings in args was
// name:"Tarak Modi"
// and "param" was "name" then
// this method would return "Tarak Modi"
public static String getParam(String[] args, String param) {
    int numArgs = args.length;
    if( numArgs > 0 ) {
        String clParam = param + ":";
        for( int i=0; i<numArgs; i++ ) {
            int index;
            // Find a command line arg that begins with
            // whatever clParam is...
            index = args[i].indexOf(clParam);
            if( index == -1 )
                continue;
            index = index + clParam.length();
            return(args[i].substring(index).trim());
        }
    }
    return(null);
}
```

Once the parameter values are read, the client tries to connect to the space indicated by the space-name parameter. This is done as follows:

```
// The first parameter is the space name
String spaceName = args[0];

QSpace space = null;
try {
    space = (QSpace)Naming.lookup(spaceName);
}
catch( Exception e ) { // Remaining code not shown?
```

For example, if `spaceName` is "ModiSpace", then the static `lookup` method on the `java.rmi.Naming` class will lookup the "ModiSpace" parameter value in the RMI registry on port 1099 on the local machine and return the remote object corresponding to it from the registry.

Now let's walk through the three different usages of the client program:

Usage 1

Putting a message in the space

```
java -Djava.security.policy=policy.all QSpaceClient space-name write [message:<message>]
[filter:<properties>]
```

For example,

- `java -Djava.security.policy=policy.all QspaceClient ModiSpace write`
- `java -Djava.security.policy=policy.all QspaceClient ModiSpace write message:"This is a test message?"`
- `java -Djava.security.policy=policy.all QspaceClient ModiSpace write filter:filter.properties`
- `java -Djava.security.policy=policy.all QspaceClient ModiSpace write message:"This is a test message?" filter:filter.properties`

The order of the filter and message parameters does not matter. Any combination of these two parameters may be specified. If a filter parameter is specified it is loaded into a properties collection as follows:

```
temp = QSpaceUtils.getParam(args, "filter");
filterProps = new Properties();
try {
    filterProps.load(new FileInputStream(temp));
}
catch( // Remaining code not shown?
```

If a message parameter is specified that message will be used instead of the default "Hello" message.

Finally, the program will put the message in the space as follows:

```
if( filterProps != null )
    space.write(message.getBytes(), filterProps);
else
    space.write(message.getBytes());
```

If a filter was specified then the two parameter version of `write` is used. Also note the use of the `getBytes` method on the message string. This is because the space only accepts an array of bytes.

Usage 2

Taking a message out from the space

```
java -Djava.security.policy=policy.all QSpaceClient space-name take [timeout:<timeout>]
[filter:<string>]
```

For example,

- `java -Djava.security.policy=policy.all QspaceClient ModiSpace take`
- `java -Djava.security.policy=policy.all QspaceClient ModiSpace take timeout:5000`
- `java -Djava.security.policy=policy.all QspaceClient ModiSpace take filter:"make='civic' AND (model='ex' OR year='1999')"`
- `java -Djava.security.policy=policy.all QspaceClient ModiSpace take timeout:5000 filter:"make='civic' AND (model='ex' OR year='1999')"`

The order of the filter and timeout parameters does not matter. Any combination of these two parameters may be specified. If a filter parameter is specified it must be a string that conforms to the message selector syntax discussed in chapter 5. The timeout parameter determines how long the `take` method on the space will wait for a message. It is specified in milliseconds. The default timeout value set in this program is `-1`, which is a special value that causes the `take` method to block until a message arrives or the space shuts down.

Once the parameters are obtained the client program will invoke the `take` method on the space as follows:

```
if( filterString != null )
    System.out.println(new String(space.take(filterString, timeout)));
else
    System.out.println(new String(space.take(timeout)));
```

If a filter was specified then the two parameter version of `take` is used.

Usage 3: java QSpaceClient space-name shutdown

For example,

```
java QSpaceClient ModiSpace shutdown
```

This will instruct the client program to shutdown the space "ModiSpace". It does this by calling the shutdown method on the space as follows:

```
try {
    space.shutdown();
}
catch( Exception e ) {
    e.printStackTrace();
}
```

Client Miscellania

The client has four "interesting" lines commented out. One of these lines is

```
//space.write(QSpaceUtils.getBytes(new TestObject(
//"This is the Test Object message...")));
```

TestObject is a simple, serializable class as shown below:

```
// A test object to test putting any
// Serializable object in a qspace
public class TestObject implements java.io.Serializable {
    private String _message;
    public TestObject() {
    }
    public TestObject(String s) {
        _message = s;
    }
    public String getMessage() {
        return(_message);
    }
}
```

As I mentioned before, our space implementation can contain any bytestream, including a serialized Java object. Once again, the class `QSpaceUtils` comes to our rescue with a pair of helpers: `getBytes` and `getObject`. The `getBytes` static method takes a reference to a `Serializable` object and returns an array of bytes. This method is shown below:

```
// Get the serialized bytestream for the
// given Serializable object
public static byte[] getBytes(java.io.Serializable object) {
    java.io.ByteArrayOutputStream bos =
        new java.io.ByteArrayOutputStream();
    try {
        java.io.ObjectOutputStream oos =
            new java.io.ObjectOutputStream(bos);
        oos.writeObject(object);
        return(bos.toByteArray());
    }
    catch( Exception e ) {
        e.printStackTrace();
        return(null);
    }
}
```

Another interesting "commented out" line is:

```
..//System.out.println(((TestObject)QSpaceUtils.getObject(
```

```
//space.take(timeout)).getMessage());
```

The `getObject` static method takes an array of bytes and returns a reference to a `Serializable` object. In this case, I typecast it to a `TestObject` and call `getMessage` on it. The `getObject` method is shown below:

```
// Get the object corresponding to a serialized bytestream
public static java.io.Serializable getObject(byte[] bytes) {
    java.io.ByteArrayInputStream bis =
        new java.io.ByteArrayInputStream(bytes);
    try {
        java.io.ObjectInputStream ois =
            new java.io.ObjectInputStream(bis);
        return (java.io.Serializable)ois.readObject();
    }
    catch( Exception e ) {
        e.printStackTrace();
        return(null);
    }
}
```

I suggest that you uncomment the four lines of code that are currently commented out and comment out the other corresponding four lines of code i.e. the line of code just above each line of code currently commented out. Now compile and run the client again and observe the outputs.

Using QSpace to Solve Problems

Before wrapping up this chapter, let's take a look at a client/server application that uses QSpace as its communication backbone. The client program creates 10 computation tasks and puts each one in the space named "ModiSpace". A computation task message consists of an instance of the `ComputeTask` class and a filtering property "type" set equal to "Task". The `ComputeTask` class is defined as follows:

```
public class ComputeTask implements java.io.Serializable {

    // the id of the operation
    public int id;

    // what operation.
    // Valid values are "ADD", "SUBTRACT", and "MULTIPLY".
    public String operation;

    // The operands.
    public double value1;
    public double value2;

    public ComputeTask() {
    }
}
```

Note that the class is serializable, since we will need to put an instance of it in the space. The client then waits for each task to be completed. The client does not care about the order in which the computation tasks are completed or which server completes them. The client program is as follows:

```
import java.util.Properties;
import java.rmi.Naming;
import java.rmi.RMI SecurityManager;

public class Client {
    public static void main(String args[]) {
        // Install a security manager.
        if( System.getSecurityManager() == null )
```



```

        System.setSecurityManager(new RMISecurityManager());

// Find "ModiSpace"
QSpace space = null;
try {
    space = (QSpace)Naming.lookup("ModiSpace");
}
catch( Exception e ) {
    e.printStackTrace();
    System.exit(-1);
}

// Create the compute tasks.
// Tasks 0, 5, and 9 are "ADD" tasks
// Tasks 1, 3, and 8 are "SUBTRACT" tasks
// Tasks 2, 4, 6, and 7 are "MULTIPLY" tasks
System.out.println("Tasks...");
try {
    for( int i=0; i<10; i++ ) {
        ComputeTask task = new ComputeTask();
        task.id = i;
        if( i==0 || i==5 || i==9 )
            task.operation = "ADD";
        else if( i==1 || i==3 || i==8 )
            task.operation = "SUBTRACT";
        else
            task.operation = "MULTIPLY";
        task.value1 = Math.round(Math.random()*10);
        task.value2 = Math.round(Math.random()*10);

        // Show the task
        System.out.println("Task #" + i + " [" +
            task.operation + "," + task.value1 + "," + task.value2 + "]");

        // Put the task in the space
        // Mark it as a task using a filter property "type"
        Properties f = new Properties();
        f.put("type", "Task");

        space.write(QSpaceUtils.getBytes(task), f);
    }

    // Get the results
// Results are marked by setting the filter
// property "type" to "Result"
    System.out.println("\nResults...");
    int i=0;
    while( i<10 ) {
        ComputeResult result = (ComputeResult)QSpaceUtils.getObject(
            space.take("type='Result'", -1));
        i++;

        // Show the result
        System.out.println("Task #" + result.id +
            " [" + result.value + "]");
    }
}
catch( Exception e ) {
    e.printStackTrace();
}

// Done
System.exit(0);
}
}

```

The server program finds "ModiSpace" and waits for computation tasks from any client. When any such tasks are found, the server carries out the requested computation and puts a computation result message back in the space. The computation result message consists of an instance of the ComputeResult class and a filtering property "type" set equal to "Result". The ComputeResult class is as follows:

```
public class ComputeResult implements java.io.Serializable {

    // The id of the request task.
    public int id;

    // The result
    public double value;

    public ComputeResult() {
    }
}
```

Note that the class is serializable, since we will need to put an instance of it in the space. The server program is as follows:

```
import java.util.Properties;
import java.rmi.Naming;
import java.rmi.RMISecurityManager;

public class Server {
    public static void main(String args[]) {

        // Install a security manager
        if( System.getSecurityManager() == null )
            System.setSecurityManager(new RMISecurityManager());

        // Find "ModiSpace"
        QSpace space = null;
        try {
            space = (QSpace)Naming.lookup("ModiSpace");
        }
        catch( Exception e ) {
            e.printStackTrace();
            System.exit(-1);
        }

        System.out.println("Server is ready to accept " +
            " ComputeTasks...");

        // Loop forever, waiting for compute tasks.
        while( true ) {
            try {
                // "Take" a task from the space.
                // Note the use of the selection string "type='Task'"
                // Use getObject method on QSpaceUtils
                // to get the task object from the bytes
                ComputeTask task = (ComputeTask)QSpaceUtils.getObject(
                    space.take("type='Task'",-1));

                // Show the task that we got from the space
                System.out.println("Got Task #" + task.id
                    + " [" + task.operation + "," +
                    task.value1 + "," + task.value2 + "]);

                // Compute the results
                // And put the result back in ModiSpace".
                // Mark the result by setting the filter
                // property "type" to "Result"
                double value = 0;
                if( task.operation.equals("ADD") ) {
```

```

        value = task.value1 + task.value2;
    }
    else if(task.operation.equals("SUBTRACT") ) {
        value = task.value1 - task.value2;
    }
    else if( task.operation.equals("MULTIPLY") ) {
        value = task.value1 * task.value2;
    }

    // Put the result back in the space
    // Note that we add a filter property
    // Also note the use of the getBytes
    // method on QSpaceUtils to put the
    // Serializable result object in the
    // space.
    ComputeResult result = new ComputeResult();
    result.id = task.id;
    result.value = value;
    Properties f = new Properties();
    f.put("type", "Result");
    space.write(QSpaceUtils.getBytes(result),f);
}
catch( Exception e ) {
}
}
}
}

```

Running the Client/Server Application

The steps required to run the client/server application are as follows:

1. Compile the client and server programs. Make sure that QSpace.class is in the classpath.
2. Start a space named "ModiSpace" that supports filtering (as done earlier).
3. Start a server application as shown below. Make sure that QSpace.class and QSpaceException.class are in the classpath.

```

java -Djava.compiler=NONE -Djava.security.policy=policy.all
    -cp E:\dev\test\qspace\;. Server

```

This is shown in figure 3.

4. Start client application as shown below. Make sure that QSpace.class is in the classpath.

```

java -Djava.compiler=NONE -Djava.security.policy=policy.all
    -cp E:\dev\test\qspace\;. Client

```

This is shown in figure 4.

5. Try running several instances of the server application at the same time. Notice a difference in how fast the client completes? The client has no idea of how many servers are in existence or even where they are. This is an example of a scalable, loosely coupled, and (to some extent) fault tolerant distributed system.

Figures 3 – 5 illustrate the results of the steps 2 – 4.

```
MS-DOS C:\WINNT\System32\cmd.exe - java -Djava.compiler=NONE -Djava.security.policy=policy.all -cp...
E:\My Documents\Personal\Articles and Papers\JMS Book\qspace\example>java -Djava.compiler=NONE -Djava.security.policy=policy.all -cp E:\dev\test\qspace\;.
Server
Server is ready to accept ComputeTasks...
Got Task #0 [ADD,5.0,9.0]
Got Task #1 [SUBTRACT,6.0,4.0]
Got Task #2 [MULTIPLY,7.0,4.0]
Got Task #3 [SUBTRACT,4.0,2.0]
Got Task #4 [MULTIPLY,10.0,2.0]
Got Task #5 [ADD,3.0,8.0]
Got Task #6 [MULTIPLY,1.0,2.0]
Got Task #7 [MULTIPLY,7.0,8.0]
Got Task #8 [SUBTRACT,8.0,2.0]
Got Task #9 [ADD,1.0,1.0]
```

Figure 4: The Server window after a Client has executed

```
MS-DOS Select C:\WINNT\System32\cmd.exe
E:\My Documents\Personal\Articles and Papers\JMS Book\qspace\example>java -Djava.compiler=NONE -Djava.security.policy=policy.all -cp E:\dev\test\qspace\;.
Client
Tasks...
Task #0 [ADD,5.0,9.0]
Task #1 [SUBTRACT,6.0,4.0]
Task #2 [MULTIPLY,7.0,4.0]
Task #3 [SUBTRACT,4.0,2.0]
Task #4 [MULTIPLY,10.0,2.0]
Task #5 [ADD,3.0,8.0]
Task #6 [MULTIPLY,1.0,2.0]
Task #7 [MULTIPLY,7.0,8.0]
Task #8 [SUBTRACT,8.0,2.0]
Task #9 [ADD,1.0,1.0]

Results...
Task #0 [14.0]
Task #1 [2.0]
Task #2 [28.0]
Task #3 [2.0]
Task #4 [20.0]
Task #5 [11.0]
Task #6 [2.0]
Task #7 [56.0]
Task #8 [6.0]
Task #9 [2.0]

E:\My Documents\Personal\Articles and Papers\JMS Book\qspace\example>
```

Figure 5: The Client Application window after it has finished execution

```
MS-DOS C:\WINNT\System32\cmd.exe - java -cp ".;E:\Program Files\Fiorano\FioranoMQ\lib\rt.jar;E:\Pr...
E:\My Documents\Personal\Articles and Papers\JMS Book\qspace>java -cp ".;E:\P
rogram Files\Fiorano\FioranoMQ\lib\rt.jar;E:\Program Files\Fiorano\FioranoMQ\
lib\fmprtl.zip" -Djava.compiler=NONE -Djava.security.policy=policy.all -Djav
a.rmi.server.codebase=http://localhost:9050/ QSpaceImpl "-PropertiesFile=qs
pace.properties"
Using Properties File qspace.properties
Debugging is turned on...
This space is called ModiSpace, is not persistent, and allows filters
The space is ready and is available as 'ModiSpace' in the RMI Registry on thi
s machine at port 1099
Selection criteria: type='Task'
Writing message[114 bytes]...
Added property [type=Task]
Writing message[119 bytes]...
Added property [type=Task]
Took message [114 bytes]...
type=Task
Writing message[119 bytes]...
Added property [type=Task]
Writing message[119 bytes]...
Added property [type=Task]
Writing message[119 bytes]...
Added property [type=Task]
Writing message[114 bytes]...
Added property [type=Task]
Writing message[119 bytes]...
Added property [type=Task]
Writing message[59 bytes]...
Added property [type=Result]
Writing message[119 bytes]...
Added property [type=Task]
Selection criteria: type='Task'
Writing message[119 bytes]...
Added property [type=Task]
Took message [119 bytes]...
type=Task
Writing message[114 bytes]...
Added property [type=Task]
Selection criteria: type='Result'
Writing message[59 bytes]...
Added property [type=Result]
Selection criteria: type='Task'
Took message [59 bytes]...
type=Result
Selection criteria: type='Result'
Took message [119 bytes]...
type=Task
Took message [59 bytes]...
type=Result
Writing message[59 bytes]...
Added property [type=Result]
Selection criteria: type='Result'
Selection criteria: type='Task'
Took message [59 bytes]...
type=Result
Selection criteria: type='Result'
Took message [119 bytes]...
```

Figure 6: (Partial Output) The QSpace window after the Client has executed.

Perusing through the output of figure 5 shows that the Server is processing Task 1 while the Client is adding other Tasks. By the time the Client puts in the seventh Task, The Server puts in the Results of the first Task and gets the next one.

Summary

Distributed applications can be notoriously difficult to design, build, and debug. The distributed environment introduces many complexities, which are not present while writing standalone applications. Some of these challenges include network latency, synchronization and concurrency, and partial failure. In this chapter, I've discussed space-based programming, which although not a "silver bullet", is an excellent concept that leads towards an elegant solution to these problems.

Space-based programming takes us one step further towards achieving our goals in a distributed system, namely those of scalability, high availability, loose coupling, and performance. It also helps us in facing the challenges mentioned above. Best of all, as I've shown in this chapter, you do not have to buy an expensive implementation to get started with this excellent concept. It's fairly easy to create a homegrown implementation using a JMS compliant messaging product that satisfies your requirements, and it's fun too. Let's wrap this chapter up by considering the benefits that JMS gives us in the space implementation discussed here. In my opinion, the most important benefit is providing the ability to easily switch between JMS providers (I showed you support for three such JMS providers). More importantly, we did not handicap ourselves by "sticking to" the JMS specification.

Chapter 9

Creating a JMS Protocol Handler

Throughout this book, I've presented JMS as being an excellent foundation for enterprise applications upon. But it is not very appealing to force every developer in your organization to learn JMS²⁰. One of the most common ways to avoid that is to create a reusable library that encapsulates all the JMS related code/knowledge. Unfortunately, even this library would require a learning curve, albeit a smaller one (hopefully). In design pattern lingo this library is referred to as a facade²¹. One possible definition of a facade is as follows:

"A higher-level interface that provides a unified way of accessing a subsystem and as a result makes the subsystem easier to use."

Thus, in contrast to most design patterns that help break the system up into subsystems, the facade design pattern rolls up a complex subsystem into one, easy-to-use system. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

In this chapter, I will present an alternative facade based on the Java protocol handler architecture. The major advantage of this approach is that most Java developers are all already familiar with using this architecture via the `URL` class in the `java.net` package. For example:

```
URL url = new URL("http://www.javasoft.com/index.html");
```

An additional and by no means minor benefit is that this is a time-tested architecture built into the Java language/platform itself. Best of all, this architecture is flexible enough to allow the "plug-in" of new protocol [handlers]. That is exactly what I will show you how to do in this chapter by creating a protocol handler for JMS.

For example, using the JMS protocol handler that we will create in this chapter, programming with JMS can be as simple as:

```
// create a new URL with our custom "jms" protocol.
URL url = new URL("jms://Queue/ModiQueue");
URLConnection uc = url.openConnection();

// Send a message
DataOutputStream dos =
    new DataOutputStream(uc.getOutputStream());
dos.writeUTF("Hello!");
dos.flush();
.
.

// Receive the message
DataInputStream dis = new DataInputStream(uc.getInputStream());
String message = dis.readUTF();

// close the streams.
dos.close();
```

²⁰ Actually, developers would probably love the idea of learning a new API, but managers would be worried about the cost and time involved.

²¹ For more details on this very useful design pattern refer to "Elements of Reusable Object-Oriented Software" by Eric Gamma, et al.

```
dis.close();
```

An Overview of the Protocol Handler Architecture

The gateway to this architecture is the `java.net.URL` class, which encapsulates a URL string. The general form of a URL is

```
protocol://host:port/filepath#ref
```

Examples include "`http://www.javasoft.com/index.html`" or "`file:///C:/temp/junk.txt`"²². In the first example, the protocol is `http`, the host name is `www.javasoft.com`, and the filepath is `index.html`. Since no port number has been specified, the protocol handler will use a protocol specific/default port, which in this case will be port number 80. In the second example, the protocol is `file` and the filepath is `C:/temp/junk.txt`. Note that in this case neither the host name nor the port number have been specified (which would have been between the first and second slashes), so the file protocol handler will use protocol specific/default values for these, such as "`localhost`" for the host name.

As an aside, the format of the URL string for the `jms` protocol will be as follows.

```
jms://Queue/<QueueName>
      or
jms://Topic/<TopicName>
```

Note that the `jms` protocol does not have a concept of a host name, port number, or filepath. Instead the host name is actually the messaging style and the filepath is the destination.

The `URL` class itself does not know how to access the resource stream represented by the URL string. Instead, the `URL` class relies on a set of other classes to handle this. When a new `URL` class instance is created it resolves²³ the URL string to a protocol specific handler i.e. the protocol handler class. This protocol handler knows how to create a connection to the resource represented by the URL string and return an object corresponding to this connection. Since this resolution occurs at construction time, any attempt to construct an instance of `URL` with an unknown/invalid protocol will throw a `MalformedURLException` during the construction itself. The relevant portion of the `URL` constructor is shown below.

```
if(handler == null &&
    (handler = getURLStreamHandler(protocol)) == null) {
    throw new MalformedURLException(
        "unknown protocol: " + protocol);
}
```

I will discuss the `getURLStreamHandler` method in detail later in this chapter.

Sun provides protocol handlers for several standard and widely used protocols such as `http`, `ftp`, `mailto`, and `gopher`. Protocol handlers must follow a strict naming convention. The class must always be named *Handler*. The package name must always have the protocol name as its last part. For example, Sun's protocol handler for the `http` protocol is called `Handler` and is in the package `sun.net.www.protocol.http`. Note that the package name ends with "`http`". In our case the "`jms`" protocol handler will be in the `jmsbook.jms` package and will be called `Handler`. To make the Java runtime aware of your own protocol handlers you must use the `java.protocol.handler.pkgs` system property. This property is set equal to a "|" delimited list of package name prefixes. These prefixes will be used to resolve the specified protocol name to a protocol handler object. Note that these package names must not include the last part i.e. the

²² The three slashes '///' is not an error. You will find out why in a moment.

²³ It actually uses a `URLStreamHandlerFactory` if one is specified. The details of this case are beyond the scope of this book.

protocol name. So, in our case this property will be set to `jmsbook` (and not `jmsbook.jms`), as follows:

```
java.protocol.handler.pkgs=jmsbook
```

The `getURLStreamHandler` method

So why does Java impose such a strict naming convention for protocol handlers? The answer is found by examining the URL class source code, more specifically the `getURLStreamHandler` method implementation, which is called to resolve a protocol name to the corresponding protocol handler. The relevant portion of this method is shown below for reference. Read the inline comments for the explanation of the code fragment.

```
.
.
.
// Get the list of package prefixes
// protocolPathProp has been defined as
// "java.protocol.handler.pkgs"
String packagePrefixList = null;
PackagePrefixList = (String)
    java.security.AccessController.doPrivileged(
        new sun.security.action.GetPropertyAction(
            protocolPathProp, ""));

// Add the standard protocols package to the list!
// First, if any package prefixes were found, append
// another delimiter, "|", to the end.
if (packagePrefixList != "") {
    packagePrefixList += "|";
}

// and now append "sun.net.www.protocol" to the end.
// Important:
// Since this package is appended at the end of user
// specified packages, a user can override any of the
// Sun provided protocol handler implementations, such
// as the one for http.
packagePrefixList += "sun.net.www.protocol";

// And now parse through the list?
// Remember, "|" is the delimiter.
StringTokenizer packagePrefixIter =
    new StringTokenizer(packagePrefixList, "|");

// Keep going until either we get a handler or
// no more tokens remain.
// Note that there will always be at least
// one token, sun.net.www.protocol.
while (handler == null && packagePrefixIter.hasMoreTokens()) {

    // Get the next token
    String packagePrefix = packagePrefixIter.nextToken().trim();
    try {
        // Create the fully qualified class name.
        // Eg. jmsbook + jms + ".Handler"
        String clsName = packagePrefix + "." +
            protocol + ".Handler";
        Class cls = null;
        try {
            // Now try loading the class with that name.
            cls = Class.forName(clsName);
        }
        catch (ClassNotFoundException e) {
            ClassLoader cl =
                ClassLoader.getSystemClassLoader();
```

```

        if (cl != null) {
            cls = cl.loadClass(clsName);
        }
    }
    if (cls != null) {
        // create a new instance.
        handler = (URLConnectionHandler)cls.newInstance();
    }
}
catch (Exception e) {
    // any number of exceptions can get thrown here
    // move onto the next token?
}
} // while loop.
.
.
.

```

Only one protocol handler object is created per VM per protocol. A new protocol handler is created the first time it is required and is then cached for later use. This means that multiple threads may use the same protocol handler simultaneously. Thus, the protocol handler implementation must be thread safe. The URL class instance caches the protocol handler in a static hash table, which allows any URL instance to access this handler. To get a better feel for this, let's take a look at the remainder of the `getURLConnectionHandler` method. Once again, read the comments for the explanation.

```

// This is the static hash table used
// to cache the protocol handlers.
// All access to this table must be synchronized.
static Hashtable handlers = new Hashtable();

static synchronized URLConnectionHandler getConnectionHandler(
    String protocol) {

    // Have we already resolved this protocol?
    URLConnectionHandler handler =
        (URLConnectionHandler)handlers.get(protocol);

    // Maybe not?
    if (handler == null) {
        // Use the factory (if any)
        // We will not consider this case.
        // In a nutshell, a factory implements the
        // URLConnectionHandlerFactory interface and is
        // registered with the URL instance either during
        // construction or using the
        // setURLConnectionHandlerFactory method.
        // A factory can only be set once and similar to the
        // protocol handlers is shared by all URL instances.
        if (factory != null) {
            handler =
                factory.createURLConnectionHandler(protocol);
        }

        // still don't have a handler?
        if (handler == null) {
            // All the logic to resolve a protocol
            // string to a protocol handler.
            // Plug in the the implementation that
            // we saw above here
        }
    }
}

```

```

        // Cache the handler if one was found.
        if (handler != null) {
            handlers.put(protocol, handler);
        }
    }

    // Return the handler to the caller.
    return handler;
}

```

The openConnection and openStream methods

At this point the URL has successfully resolved the protocol string to a protocol handler. The `openConnection` method may be used to gain access to a connection object. A connection object must implement the `URLConnection` interface and is used to send and receive data to and from the resource stream, respectively. The URL instance merely delegates the `openConnection` method call to the protocol handler as shown below.

```

    public URLConnection openConnection() throws java.io.IOException {
        return handler.openConnection(this);
    }

```

The URL class also provides a helper method, `openStream`, for clients only interested in receiving data from the resource stream. This method is shown below.

```

    public final InputStream openStream() throws java.io.IOException {
        return openConnection().getInputStream();
    }

```

Our JMS Protocol Handler

As discussed above, the URL class serves as the gateway into Java's protocol handler architecture. The URL class itself has very limited functionality beyond resolving a protocol string into a protocol handler. Having said that, let's take a look at the implementation for our JMS protocol handler. A class diagram showing all the pieces of the JMS protocol handler architecture and how they fit together is shown in figure 1.

As with all protocol handlers, the JMS protocol handler conforms to the following rules.

1. The class name is `Handler`.
2. It extends the `URLStreamHandler` class and provides a concrete implementation of the `openConnection` abstract method.
3. Its package name has the protocol name as its last part i.e. it is in a package called `jmsbook.jms`

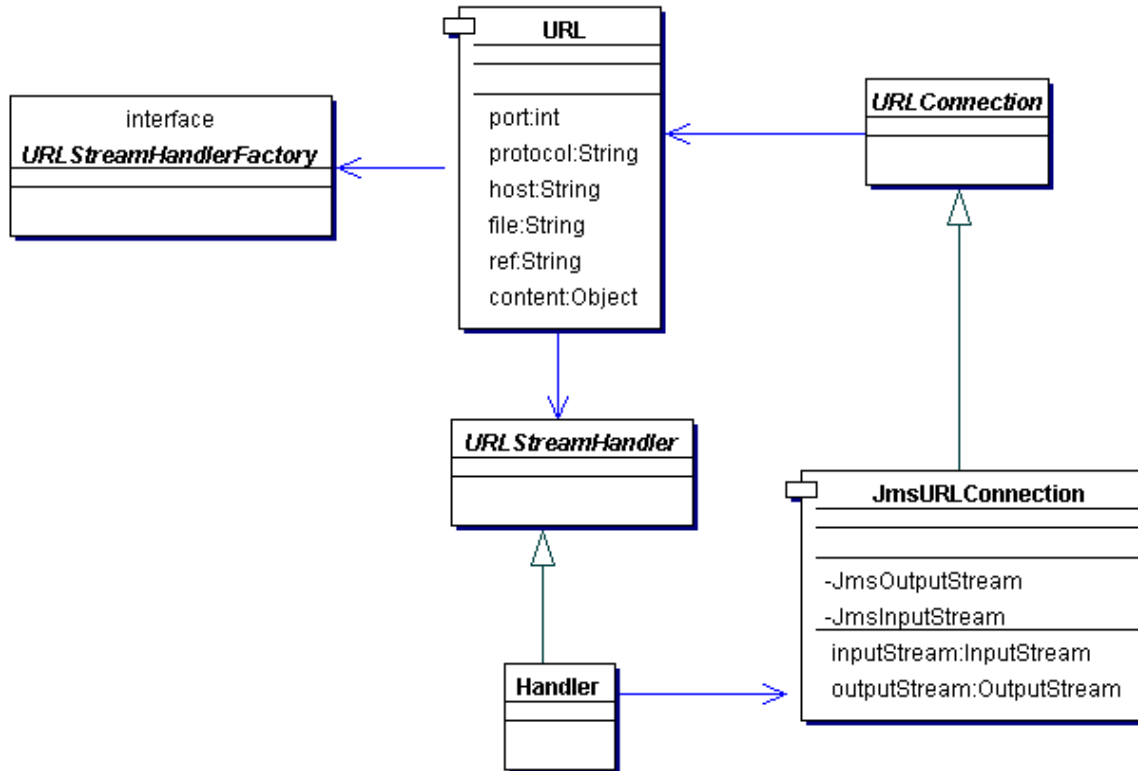


Figure 1: The JMS protocol handler architecture

The openConnection method

Since the JMS protocol handler extends the `URLStreamHandler` class, it must provide an implementation of the `openConnection` method. The `openConnection` method is responsible for returning a connection object to the caller. The caller may then use this connection object to access the resource stream. The JMS protocol is configurable and hence needs to be initialized. So, the first action taken by this method is to make sure that this instance has been initialized. This is shown below.

```

synchronized(this) {
if(!initialized)
    init();
}

```

Note that this code is synchronized since protocol handlers are shared entities and hence must be thread safe. This is the only portion of the handler code that must be synchronized in this case, since it is the only portion that results in changes to member variables²⁴. Next, based on the messaging style specified in the URL (i.e. the host name portion of the URL string), the `openConnection` method appropriately initializes our implementation of the connection object, `JmsURLConnection`, and returns it to the caller, as shown below.

```

if(u.getHost().equals("Queue"))
    return new JmsURLConnection(queueConnection,u);
else if(u.getHost().equals("Topic"))
    return new JmsURLConnection(topicConnection,u);

```

²⁴ If a handler does not have any member or class variables or if these do not change through the life of the handler then no synchronization is necessary since the handler is inherently thread safe, but that is not the case here.

```
else
    throw new IOException("Host name must be Topic or Queue.");
```

As I'll discuss in detail later, the `JmsURLConnection` class extends the `URLConnection` class and overrides a few key methods.

The initialization method – `init`

The `openConnection` method calls the `init` method if the handler instance has not been initialized. The JMS protocol handler class is configurable through a properties file, the name of which is specified in the `jmsbook.jms.propertiesFile` system property as shown below.

```
java -Djmsbook.jms.propertiesFile=C:/temp/jmsProtocol.properties
    . . . the rest of the java command
```

The `init` method checks to see if a properties file has been specified and if so loads it in. This is shown below.

```
// Has a properties file been specified?
String filename = System.getProperty("jmsbook.jms.propertiesFile");
// exception handling ignored in this snippet?
if(filename != null)
    props.load(new FileInputStream(filename));
```

The handler has two member variables that must be initialized; a reference to a queue connection and a reference to a topic connection. As we already know, connection objects in JMS are obtained from a vendor specific connection factory. I will use the same the same strategy as I did in chapter 8 for eliminating the need of this vendor specific code in the JMS protocol handler itself and to support any JMS provider. This is shown in figure 2 and in the code fragment from the `init` method below.

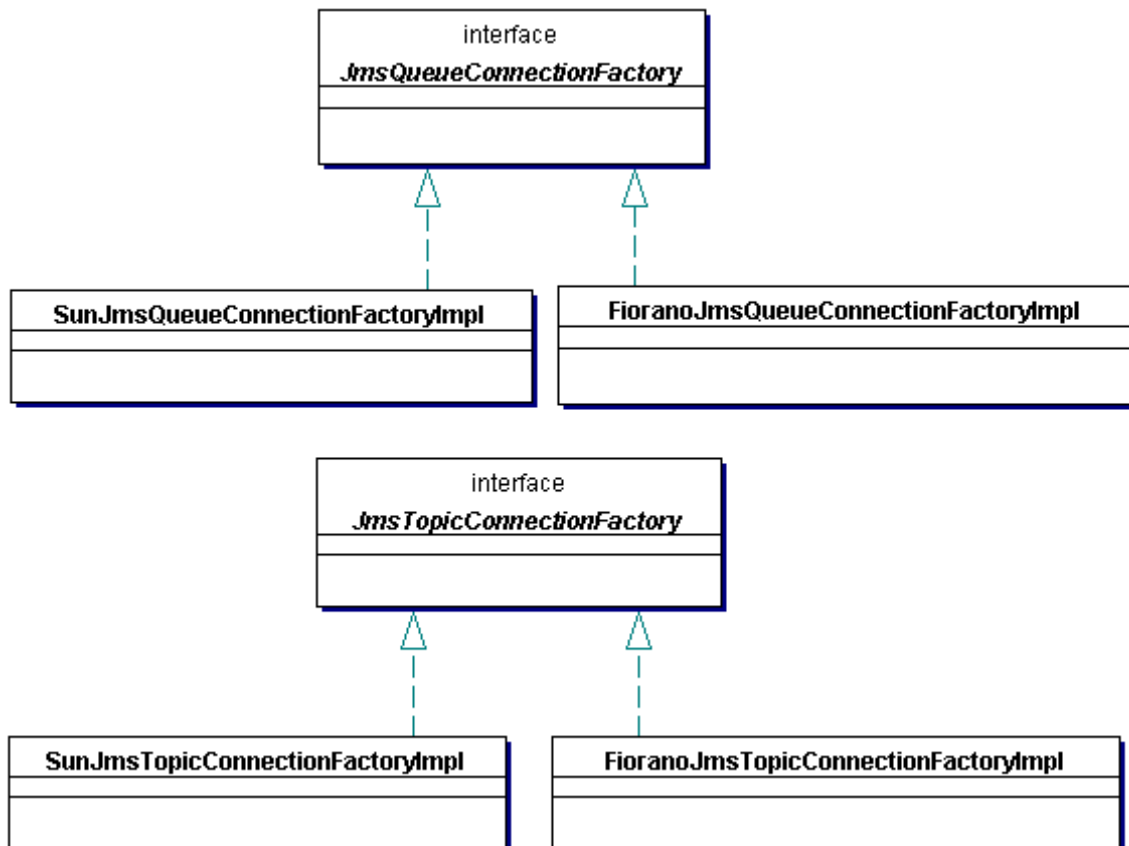


Figure 2: The JMS Protocol Queue and Topic Connection Factory Hierarchy

```
// Get the queue connection factory...
String factoryClass =
    props.getProperty("JmsQueueConnectionFactory");

// if the property JmsQueueConnectionFactory is not
// defined then use the default, which is
// Sun Java Message Queue
if(factoryClass == null)
    factoryClass = "jmsbook.jms.SunJmsQueueConnectionFactoryImpl";

// Create a new instance of the factory class
JmsQueueConnectionFactory jmsQcf = (JmsQueueConnectionFactory)
    (Class.forName(factoryClass).newInstance());

// use the factory class instance to the vendor
// specific Queue Connection factory.
QueueConnectionFactory qcf =
    jmsQcf.getQueueConnectionFactory(props);

// Get the topic connection using the connection factory
// and start it.
queueConnection = qcf.createQueueConnection();
queueConnection.start();

// Get the queue connection factory...
factoryClass = props.getProperty("JmsTopicConnectionFactory");

// if the property JmsTopicConnectionFactory is not
// defined then use the default, which is
// Sun Java Message Queue
if(factoryClass == null)
    factoryClass = "jmsbook.jms.SunJmsTopicConnectionFactoryImpl";

// Create a new instance of the factory class
JmsTopicConnectionFactory jmsTcf = (JmsTopicConnectionFactory)
    (Class.forName(factoryClass).newInstance());

// use the factory class instance to the vendor
// specific Topic Connection factory.
TopicConnectionFactory tcf =
    jmsTcf.getTopicConnectionFactory(props);

// Get the topic connection using the connection factory
// and start it.
topicConnection = tcf.createTopicConnection();
topicConnection.start();

// Initialization is complete.
// Set the flag so that init is not called again.
initialized = true;
```

A sample properties file for configuring the JMS protocol handler is shown below.

```
# The factory to use to get the initial Connection Factory
JmsQueueConnectionFactory=
    jmsbook.jms.FioranoJmsQueueConnectionFactoryImpl
JmsTopicConnectionFactory=
    jmsbook.jms.FioranoJmsTopicConnectionFactoryImpl

# Fiorano MQ specific property
QCFactoryName=primaryqcf
TCFactoryName=primarytcf
```

You'll see that this is very similar to the `qspace.properties` file used to configure QSpace in chapter 8. In the example shown above, the properties file configures the JMS protocol handler to work with Fiorano's FioranoMQ. The implementation of the factory class, `FioranoJmsQueueConnectionFactoryImpl`, is the same as the one in chapter 8 except that in chapter 8 this class was called `FioranoQSpaceFactoryImpl` and it implemented the `QSpaceFactory` interface. Take a look yourself.

```

package jmsbook.jms;
import fiorano.jms.rtl.*;
public class FioranoJmsQueueConnectionFactoryImpl
                implements JmsQueueConnectionFactory {

    public javax.jms.QueueConnectionFactory
    getQueueConnectionFactory(java.util.Properties props)
    throws javax.jms.JMSEException {

        //////////////////////////////////////
        // The same logic as in chapter 8. //
        //////////////////////////////////////
        fiorano.jms.rtl.FioranoInitialContext ic = null;
        ic = new fiorano.jms.rtl.FioranoInitialContext();
        ic.bind ();
        javax.jms.QueueConnectionFactory factory =
            (javax.jms.QueueConnectionFactory)ic.lookup(
            props.getProperty("QCFactoryName").trim());
        ic.dispose();
        return(factory);
    }
}

```

The factory class for getting the initial topic connection factory is similar. This was not shown in chapter 8 since it relates to publish-and-subscribe. So, I'll include an example for FioranoMQ below.

```

package jmsbook.jms;
import fiorano.jms.rtl.*;
public class FioranoJmsTopicConnectionFactoryImpl
                implements JmsTopicConnectionFactory {

    public javax.jms.TopicConnectionFactory
    getTopicConnectionFactory(java.util.Properties props)
    throws javax.jms.JMSEException {

        fiorano.jms.rtl.FioranoInitialContext ic = null;
        ic = new fiorano.jms.rtl.FioranoInitialContext();
        ic.bind ();
        javax.jms.TopicConnectionFactory factory =
            (javax.jms.TopicConnectionFactory)ic.lookup(
            props.getProperty("TCFactoryName").trim());
        ic.dispose();
        return(factory);
    }
}

```

Perusing through the `init` method you will notice that if no properties file is specified then it is assumed that Sun's Java Message Queue is being used. Also it is possible to have a mix and match of different messaging products. For example, there is no reason why you cannot use a different JMS provider for point-to-point messaging than the one used for publish-and-subscribe messaging. A properties file that uses FioranoMQ for point-to-point and Sun's Java Message Queue for publish-and-subscribe is shown below.

```
# The factory to use to get the initial Connection Factory
```

```

# Use FioranoMQ for point-to-point.
JmsQueueConnectionFactory=
    jmsbook.jms.FioranoJmsQueueConnectionFactoryImpl
# Use Sun's Java Message Queue for publish-and-subscribe.
JmsTopicConnectionFactory=
    jmsbook.jms.SunJmsTopicConnectionFactoryImpl

# Fiorano MQ specific property
QCFactoryName=primaryqcf

```

Also, the code indicates that this properties file is equivalent to the following properties file in which the topic connection factory implementation is not specified.

```

# The factory to use to get the initial Connection Factory
# Use FioranoMQ for point-to-point.
JmsQueueConnectionFactory=
    jmsbook.jms.FioranoJmsQueueConnectionFactoryImpl

# Fiorano MQ specific property
QCFactoryName=primaryqcf

```

This is because if a factory is not specified in the properties file then the factory class for Sun's Java Message Queue is used.

Look at how the init method handles `JMSEException`s. It checks for a "linked" exception and if one exists, the method throws a new `IOException` with the message from that [linked] exception instead of using the message from the `JMSEException`. This is shown below.

```

.
.
.
catch( JMSEException e ) {
    if( e.getLinkedException() != null )
        throw new IOException(
            e.getLinkedException().getMessage());
    else
        throw new IOException(e.getMessage());
}
.
.
.

```

The entire source for the `Handler` class is included below for reference.

```

package jmsbook.jms;
import java.net.URL;
import java.net.URLConnection;
import java.net.URLStreamHandler;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;
import javax.jms.*;

public class Handler extends URLStreamHandler {

    private QueueConnection queueConnection = null;
    private TopicConnection topicConnection = null;
    private boolean initialized = false;

    public Handler() {
    }

    protected URLConnection openConnection(URL u)
        throws IOException

```



```

{
    // Make the initialization thread safe.
    synchronized(this) {
        if( !initialized )
            init();
    }

    if( u.getHost().equals("Queue") )
        return new JmsURLConnection(queueConnection,u);
    else if( u.getHost().equals("Topic") )
        return new JmsURLConnection(topicConnection,u);
    else
        throw new IOException("Host name must be Topic or Queue.");
}

private void init() throws IOException
{
    try {
        Properties props = new Properties();
        String filename =
            System.getProperty("jmsbook.jms.propertiesFile");
        if( filename != null )
            props.load(new FileInputStream(filename));

        String factoryClass =
            props.getProperty("JmsQueueConnectionFactory");
        if( factoryClass == null )
            factoryClass =
                "jmsbook.jms.SunJmsQueueConnectionFactoryImpl";

        JmsQueueConnectionFactory jmsQcf =
            (JmsQueueConnectionFactory)(Class.forName(
                factoryClass).newInstance());

        QueueConnectionFactory qcf =
            jmsQcf.getQueueConnectionFactory(props);
        queueConnection = qcf.createQueueConnection();
        queueConnection.start();

        factoryClass =
            props.getProperty("JmsTopicConnectionFactory");
        if( factoryClass == null )
            factoryClass =
                "jmsbook.jms.SunJmsTopicConnectionFactoryImpl";

        JmsTopicConnectionFactory jmsTcf =
            (JmsTopicConnectionFactory)(Class.forName(
                factoryClass).newInstance());

        TopicConnectionFactory tcf =
            jmsTcf.getTopicConnectionFactory(props);
        topicConnection = tcf.createTopicConnection();
        topicConnection.start();

        initialized = true;
    }
    catch( JMSEException e ) {
        if( e.getLinkedException() != null )
            throw new IOException(
                e.getLinkedException().getMessage());
        else
            throw new IOException(e.getMessage());
    }
    catch( Exception e ) {
        throw new IOException(e.getMessage());
    }
}

```

```
}
```

The JmsURLConnection Class

As you've seen before, the `openConnection` method in the JMS protocol handler class returns an instance of the `JmsURLConnection` class. This class extends the `URLConnection` class and overrides the `set/getRequestProperty`, `getInputStream`, `getOutputStream`, and `connect` methods. I will discuss each one of these methods in a moment, but first let's look at the constructor. The one and only constructor takes two parameters: a `JMS Connection` object and a `URL` object. The type of connection passed in depends on the messaging style to be used. The JMS protocol handler makes this decision. At various points in the `JmsURLConnection` implementation you will see the code checking for the type of the connection object. The constructor is shown below for reference.

```
public JmsURLConnection(Connection connection, URL url)
{
    super(url);
    if(connection == null)
        throw new RuntimeException(
            "Cannot specify null connection.");
    this.connection = connection;
}
```

Now let's look at the methods that this class overrides from the base class.

The set/getRequestProperty methods

As we already know, messages in JMS are associated with a delivery mode, a priority, and a time-to-live. The `JmsURLConnection` class provides default values for three "request" properties as and allows the client (i.e. user) to configure these properties at any point in time. The default values of these properties are shown below.

```
// The delivery mode; default is persistent.
private int persistence = DeliveryMode.PERSISTENT;
// The priority; default is 9 (highest)
private int priority = 9;
// The time-to-live; default is 0 (forever)
private int ttl = 0;
```

A client can get the value of any of these properties at any time by calling the `getRequestProperty` method passing in the name of the property required. An example of querying the connection for the delivery mode is shown below.

```
// uc is a JmsURLConnection
String persistent = uc.getRequestProperty("persistent");
```

In the above code fragment, if the value of `persistent` is "true" after the statement is executed then the delivery mode is persistent. Note the name of the persistence property is "persistent". Similarly to get find out the priority and time-to-live properties call the `getRequestProperty` method with the property names "priority" and "timeToLive" respectively.

To change the value of any of these properties use the `setRequestProperty` method. For example, to change the delivery mode to non-persistent, change the priority to 4, and change the time-to-live to 10 seconds

```
setRequestProperty("persistent", "false");
setRequestProperty("priority", "4");
setRequestProperty("timeToLive", "10000");
```

If an invalid property name or value is passed in to either of these methods, a `RuntimeException` will be thrown.

The connect method

All this method does is set the `connected` member variable in the base class to `true`.

```
public void connect() throws IOException {
    this.connected = true;
}
```

The getInputStream method

This method checks to see if the JMS connection that it has is a queue connection or a topic connection and then based on this it creates the appropriate session and message consumer. It then creates a new instance of the `JmsInputStream` class passing in the session and the message consumer. By creating and passing in a new instance of the session and message consumer we ensure that we do not violate the single-threaded access requirement for these objects. This is shown below.

```
.
.
.
// is this a queue connection?
if( connection instanceof QueueConnection ) {
    // Yes?
    // Then create a queue session and receiver
    session = ((QueueConnection)connection).createQueueSession(
                                                false,1);
    Queue queue = ((QueueSession)session).createQueue(
        this.getURL().getFile().substring(1));
    consumer = ((QueueSession)session).createReceiver(queue);
}
else {
    // No? then it is a topic connection
    // So create a topic session and subscriber
    session = ((TopicConnection)connection).createTopicSession(
                                                false,1);
    Topic topic = ((TopicSession)session).createTopic(
        this.getURL().getFile().substring(1));
    consumer = ((TopicSession)session).createSubscriber(topic);
}
// And now return an input stream?
return(new JmsInputStream(session,consumer));
```

`JmsInputStream` is a private class that extends the `java.io.InputStream` class and provides a concrete implementation of the `read` method. When the client calls the `read` method (or a method that results in the `read` method being called, such as `readUTF`, `readDouble`, or any other <read> method on any decorating input stream.), it first checks if there are any more bytes in the buffer. If no more bytes exist, it calls the private method `readMessage`. This method checks if there are any more bytes remaining in a previous [partially read] message. If no such message exists it calls the `receive` method on the message consumer that was passed in during construction. The relevant portion of the `readMessage` method is shown below.

```
.
.
.
// The size of the buffer
int size = 1024;
try {
    // Does a previous partially
    // read message exist?
    if( msg != null ) {
        bytes = new byte[size];
        // Get the next 1024 (or less) bytes
        int n = msg.readBytes(bytes);
        // read 1024 bytes, just return?
        if( n == size )
```

```

        return;
    // read less than 1024 bytes.
    // we need to truncate our buffer to
    // the right size.
    else if( n != -1 && n < size ) {
        truncateTo(n);
        return;
    }
    // n == -1 means no more
    // bytes in the message.
    msg = null;
    bytes = null;
}
// Either this is the first message
// being read, or the previous message
// has no more bytes as determined above.
msg = (BytesMessage)consumer.receive();

// Does this message have a MessageLengthInBytes
// property. That way we know the exact size
// of the buffer required.
int length = size;
try {
    length = msg.getIntProperty("MessageLengthInBytes");
}
catch( Exception e ) {
    // Guess not?
}

// Create the buffer and read the first "length" bytes.
bytes = new byte[length];
int n = msg.readBytes(bytes);
if( n == size )
    return;
// This occurs if the MessageLengthInBytes property was
// not found and the message has less than 1024 bytes.
else if( n != -1 && n < size ) {
    truncateTo(n);
    return;
}
else
    throw    new IOException("Error receiving message.");
}
.
.
.

```

Let's quickly look at the `truncateTo` method, which is called when the `readBytes` method call on the message reads fewer than expected (1024) bytes. Note the use of the `System.arraycopy` method for improving the efficiency of the memory copy.

```

private void truncateTo(int n) {
    byte[] temp = new byte[n];
    System.arraycopy(bytes, 0, temp, 0, n);
    bytes = temp;
}

```

The `getOutputStream` method

This method checks to see if the JMS connection that it has is a queue connection or a topic connection and then based on this it creates the appropriate session and message producer. It then creates a new instance of the `JmsOutputStream` class passing in the session and the message producer. By creating and passing in a new instance of the session and message producer we ensure that we do not violate the single-threaded access requirement for these objects. This is shown below.

```

.
.
.
// is this a queue connection?
if( connection instanceof QueueConnection ) {
    // Yes?
    // Then create a queue session and receiver
    session = ((QueueConnection)connection).createQueueSession(
                                                false,1);

    Queue queue = ((QueueSession)session).createQueue(
        this.getURL().getFile().substring(1));
    producer = ((QueueSession)session).createSender(queue);
}
else {
    // No? then it is a topic connection
    // So create a topic session and publisher
    session = ((TopicConnection)connection).createTopicSession(
                                                false,1);

    Topic topic = ((TopicSession)session).createTopic(
        this.getURL().getFile().substring(1));
    producer = ((TopicSession)session).createPublisher(topic);
}
return(new JmsOutputStream(session,producer));

```

JmsOutputStream is a private class that extends the java.io.OutputStream class and provides a concrete implementation of the write method. When a client calls the write method (or a method that results in the write method being called, such as writeUTF, writeDouble, or any other <write> method on any decorating output stream.), it updates an internal byte stream/buffer as shown below.

```

public void write(int b) throws IOException {
    if( baos == null )
        baos = new ByteArrayOutputStream();
    baos.write(b);
}

```

A client can call write as many times as needed. Finally, when the entire message is written, the client must call flush to actually send the message. The flush method will call the private writeMessage method that contains all the logic for dealing with the message producer. The relevant portion of this method is shown below.

```

.
.
.
try {
    // Create a new bytes message
    BytesMessage msg =session.createBytesMessage();
    // and write the bytes to it?
    msg.writeBytes(bytes);

    // this will improve efficiency while reading.
    msg.setIntProperty("MessageLengthInBytes", bytes.length);

    // what type of producer is this?
    // depends on the messaging style.
    if( producer instanceof QueueSender )
        ((QueueSender)producer).send(msg,
            persistence,priority,ttl);
    else
        ((TopicPublisher)producer).publish(msg,
            persistence,priority,ttl);
}
.
.
.

```

For example, let's assume that a message consists of some basic information about a person, such as their name, age, and sex. Such a message could be sent as follows

```
// uc is a JmsURLConnection
// Wrap/Decorate the JmsOutputStream with a DataOutputStream
DataOutpputStream dos = new DataOutputStream(uc.getOutputStream());
// Write the name (string), sex (string)
// and age (long) to the stream.
dos.writeUTF(name);
dos.writeUTF(sex);
dos.writeLong(age);
// send the message.
dos.flush();
// done
dos.close();
```

A client could receive this message as follows.

```
// uc is a JmsURLConnection
// Wrap/Decorate the JmsInputStream with a DataInputStream
DataInpputStream dis = new DataInputStream(uc.getInputStream());
// Read the name (string), sex (string)
// and age (long) from the stream.
String name = dos.readUTF(name);
String sex = dos.readUTF(sex);
Long age = dos.readLong(age);
// done
dis.close();
```

Later on in this chapter I will show two complete examples of programs using the JMS Protocol handler.

The entire implementation of `JmsURLConnection` is shown below for reference.

```
package jmsbook.jms;
import java.net.URL;
import java.net.URLConnection;
import java.net.URLStreamHandler;
import java.io.*;
import java.util.Properties;
import javax.jms.*;

public class JmsURLConnection extends URLConnection {
    private Connection connection = null;
    private int persistence = DeliveryMode.PERSISTENT;
    private int priority = 9;
    private int ttl = 0;

    public JmsURLConnection(Connection connection, URL url) {
        super(url);
        if( connection == null )
            throw new RuntimeException(
                "Cannot specify null connection.");
        this.connection = connection;
    }

    public void setRequestProperty(String key, String value) {
        boolean invalidkey = false;
        try {
            if( key.equals("persistent") ) {
                if( value.equalsIgnoreCase("true") )
                    persistence = DeliveryMode.PERSISTENT;
                else if( value.equalsIgnoreCase("false") )
                    persistence = DeliveryMode.NON_PERSISTENT;
            }
        }
    }
}
```

```

        else
            throw new Exception();
    }
    else if( key.equals("priority") ) {
        priority = Integer.parseInt(value);
        if( priority < 0 || priority > 9 ) {
            priority = 9;
            throw new Exception();
        }
    }
    else if( key.equals("timeToLive") ) {
        ttl = Integer.parseInt(value);
        if( ttl < 0 ) {
            ttl = 0;
            throw new Exception();
        }
    }
    else
        invalidkey = true;
}
catch( Exception e ) {
    throw new RuntimeException(
        "Invalid request header value " + value +
        " for field " + key);
}

if( invalidkey )
    throw new RuntimeException(
        "Invalid request header " + key);
}

public String getRequestProperty(String key) {
    if( key.equals("persistent") ) {
        if( persistence == DeliveryMode.PERSISTENT )
            return(new String("true"));
        else
            return(new String("false"));
    }
    else if( key.equals("priority") )
        return(new Integer(priority).toString());
    else if( key.equals("timeToLive") )
        return(new Integer(ttl).toString());

    throw new RuntimeException(
        "Invalid request header " + key);
}

public void connect() throws IOException
{
    this.connected = true;
}

public InputStream getInputStream() throws IOException
{
    if( !connected )
        connect();

    MessageConsumer consumer = null;
    Session session = null;
    try {
        if( connection instanceof QueueConnection ) {
            session = ((QueueConnection)
                connection).createQueueSession(false,1);
            Queue queue = ((QueueSession)
                session).createQueue(this.getURL(
                    ).getFile().substring(1));

            consumer = ((QueueSession)

```

```

        session).createReceiver(queue);
    }
    else {
        session = ((TopicConnection)
            connection).createTopicSession(false,1);
        Topic topic = ((TopicSession)
            session).createTopic(this.getUrl(
                ).getFile().substring(1));
        consumer = ((TopicSession)
            session).createSubscriber(topic);
    }
    return(new JmsInputStream(session,consumer));
}
catch( JMSEException e ) {
    if( e.getLinkedException() != null )
        throw new IOException(
            e.getLinkedException().getMessage());
    else
        throw new IOException(e.getMessage());
}
}

public OutputStream getOutputStream() throws IOException
{
    if( !connected )
        connect();

    MessageProducer producer = null;
    Session session = null;
    try {
        if( connection instanceof QueueConnection ) {
            session = ((QueueConnection)
                connection).createQueueSession(false,1);
            Queue queue = ((QueueSession)
                session).createQueue(this.getUrl(
                    ).getFile().substring(1));
            producer = ((QueueSession)
                session).createSender(queue);
        }
        else {
            session = ((TopicConnection)
                connection).createTopicSession(false,1);
            Topic topic = ((TopicSession)
                session).createTopic(this.getUrl(
                    ).getFile().substring(1));
            producer = ((TopicSession)
                session).createPublisher(topic);
        }
        return(new JmsOutputStream(session,producer));
    }
    catch( JMSEException e ) {
        if( e.getLinkedException() != null )
            throw new IOException(
                e.getLinkedException().getMessage());
        else
            throw new IOException(e.getMessage());
    }
}

private class JmsOutputStream extends OutputStream {
    private ByteArrayOutputStream baos = null;
    private Session session = null;
    private MessageProducer producer = null;

    private JmsOutputStream(Session session,
        MessageProducer producer) {
        this.session = session;
        this.producer = producer;
    }
}

```



```

    }

    public void write(int b) throws IOException
    {
        if( baos == null )
            baos = new ByteArrayOutputStream();
        baos.write(b);
    }

    public void flush() throws IOException
    {
        if( baos == null )
            return;
        writeMessage(baos.toByteArray());
        baos.close();
        baos = null;
    }

    public void close() throws IOException
    {
        flush();
        try {
            producer.close();
            session.close();
            producer = null;
            session = null;
        }
        catch( JMSEException e ) {
            if( e.getLinkedException() != null )
                throw new IOException(
                    e.getLinkedException().getMessage());
            else
                throw new IOException(e.getMessage());
        }
    }

    private void writeMessage(byte[] bytes) throws IOException
    {
        try {
            BytesMessage msg = session.createBytesMessage();
            msg.writeBytes(bytes);

            // improve efficiency while reading.
            msg.setIntProperty("MessageLengthInBytes",
                bytes.length);
            if( producer instanceof QueueSender )
                ((QueueSender)producer).send(
                    msg,persistence,priority,ttl);
            else
                ((TopicPublisher)producer).publish(
                    msg,persistence,priority,ttl);
        }
        catch( JMSEException e ) {
            if( e.getLinkedException() != null )
                throw new IOException(
                    e.getLinkedException().getMessage());
            else
                throw new IOException(e.getMessage());
        }
    }
}

private class JmsInputStream extends InputStream {
    private Session session = null;
    private MessageConsumer consumer = null;
    private byte[] bytes = null;
    BytesMessage msg = null;
    private int index = 0;
}

```

```

private JmsInputStream(Session session,
                       MessageConsumer consumer) {
    this.session = session;
    this.consumer = consumer;
}

public int read() throws IOException
{
    if( bytes == null ) {
        readMessage();
        index = 0;
    }

    if( index < bytes.length ) {
        int retVal = bytes[index] & 0xff;
        if( index == bytes.length-1 ) {
            bytes = null;
            index = 0;
        }
        else
            index++;
        return(retVal);
    }
    else {
        return(-1);
    }
}

public int available() throws IOException
{
    if( bytes != null )
        return(0);
    return(bytes.length - index);
}

public void close() throws IOException
{
    try {
        consumer.close();
        session.close();
        session = null;
        consumer = null;
    }
    catch( JMSEException e ) {
        if( e.getLinkedException() != null )
            throw new IOException(
                e.getLinkedException().getMessage());
        else
            throw new IOException(e.getMessage());
    }
}

private void readMessage() throws IOException
{
    int size = 1024;
    try {
        if( msg != null ) {
            bytes = new byte[size];
            int n = msg.readBytes(bytes);
            if( n == size )
                return;
            else if( n != -1 && n < size ) {
                truncateTo(n);
                return;
            }
            msg = null;
            bytes = null;
        }
    }
}

```

```

    }
    msg = (BytesMessage)consumer.receive();
    int length = size;
    try {
        length = msg.getIntProperty(
            "MessageLengthInBytes");
    }
    catch( Exception e ) {
    }
    bytes = new byte[length];
    int n = msg.readBytes(bytes);
    if( n == size )
        return;
    else if( n != -1 && n < size ) {
        truncateTo(n);
        return;
    }
    else
        throw new IOException(
            "Error receiving message.");
    }
    catch( JMSEException e ) {
        if( e.getLinkedException() != null )
            throw new IOException(
                e.getLinkedException().getMessage());
        else
            throw new IOException(e.getMessage());
    }
    }
}

private void truncateTo(int n) {
    byte[] temp = new byte[n];
    System.arraycopy(bytes,0,temp,0,n);
    bytes = temp;
}
}
}

```

Creating Programs that use the JMS Protocol Handler

Let's create a pair of programs, one for sending messages and one for receiving messages, using the JMS protocol handler architecture.

The Sender Program

The goal of this program is to send a message on a queue called "ModiQueue". The program is shown in its entirety below. Note how different this program looks than traditional JMS programs. For example, there are no import statements importing classes from the `javax.jms` package. In fact apart from the name of the protocol (jms) in the URL string there is no indication of JMS at all. If for some reason, we wanted to send the "request/message" over http instead of a JMS provider, we simply change the URL; no other code changes are required in this program. Also note how easy it is to switch between the point-to-point and publish-and-subscribe messaging styles. Just change the URL string "jms://Queue/ModiQueue" to "jms://Topic/ModiTopic" assuming that "ModiTopic" is a valid JMS Topic.

```

import java.net.*;
import java.io.*;
public class Sender {
    public static void main(String[] args) {
        try {
            // This URL indicates a point-to-point
            // messaging style and the destination
            // "ModiQueue".

```

```

URL url = new URL("jms://Queue/ModiQueue");
//URL url = new URL("jms://Topic/ModiTopic");

// Get a connection
URLConnection uc = url.openConnection();

// Configure this instance of
// the URL connection
// We want the messages non-persistent and
// to expire in 10 seconds.
uc.setRequestProperty("persistent","false");
uc.setRequestProperty("timeToLive","10000");

// Get the output stream to write messages to.
OutputStream os = uc.getOutputStream();
DataOutputStream dos = new DataOutputStream(os);

// The message contains two strings.
dos.writeUTF("Hello!");
dos.writeUTF("Tarak Modi");

// This will actually send the message.
dos.flush();

// Done?
dos.close();
}
catch( Exception e ) {
    e.printStackTrace();
}
}
System.exit(-1);
}
}

```

The steps taken by this program are shown in figure 3.

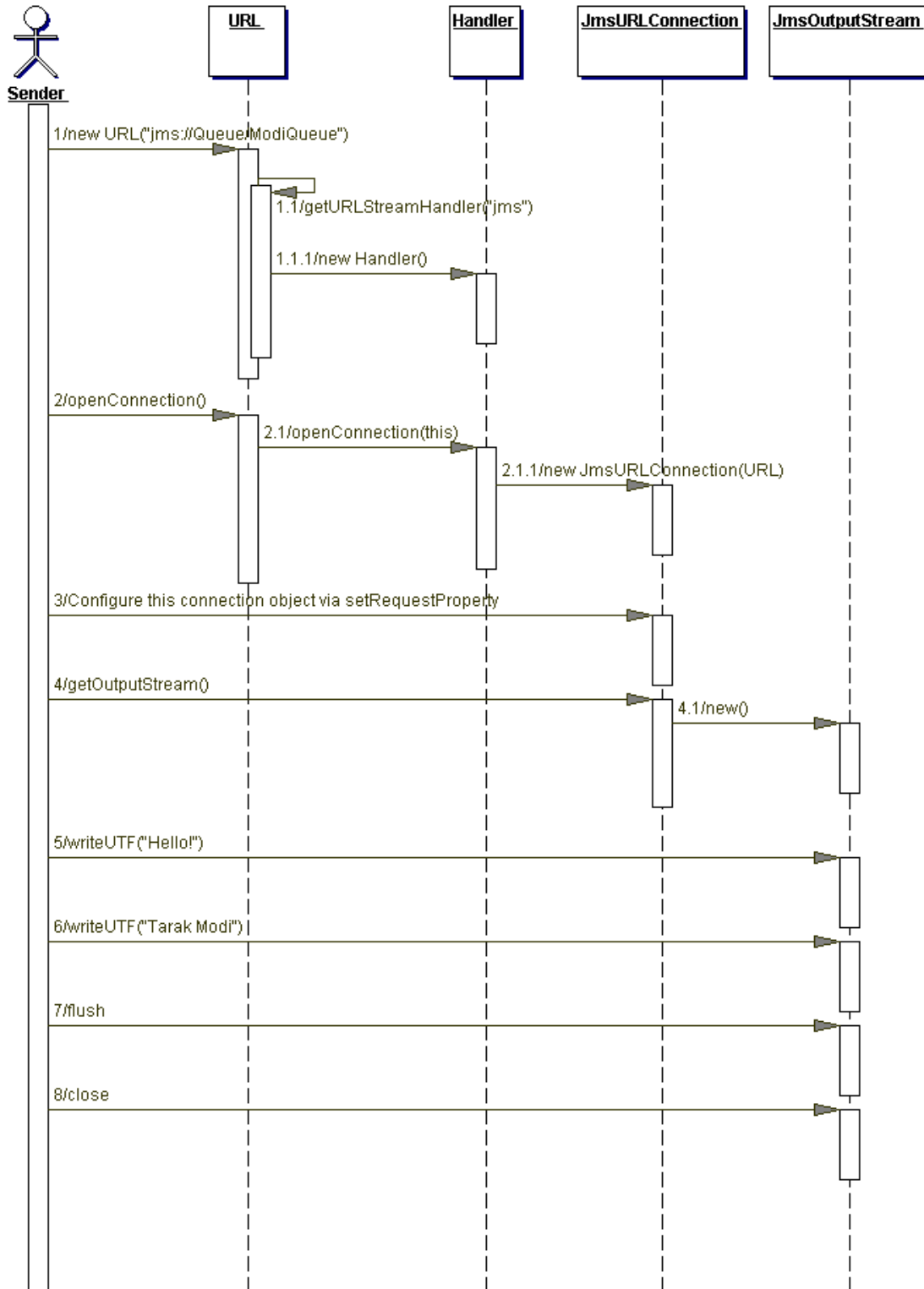


Figure 3: The Sender Sequence Diagram

The Receiver Program

```

import java.net.*;
import java.io.*;

```

```

public class Receiver {
    public static void main(String[] args) {
        try {
            // This URL indicates a point-to-point
            // messaging style and the destination
            // "ModiQueue".
            URL url = new URL("jms://Queue/ModiQueue");
            //URL url = new URL("jms://Topic/ModiTopic");

            // Get the input stream
            InputStream is = url.openStream();
            DataInputStream dis = new DataInputStream(is);

            // Read the message from the stream
            String message = dis.readUTF();
            // Should see "Hello!"
            System.out.println(message);
            message = dis.readUTF();
            // Should see "Tarak Modi"
            System.out.println(message);

            // Done?
            dis.close();
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
        System.exit(-1);
    }
}

```

The goal of this program is to receive messages from a queue called "ModiQueue". The program is shown in its entirety below. Once again note how different this program looks than traditional JMS programs and how easy it is to switch between the point-to-point and publish-and-subscribe messaging styles. Just change the URL string "jms://Queue/ModiQueue" to "jms://Topic/ModiTopic" assuming that ModiTopic is a valid JMS Topic.

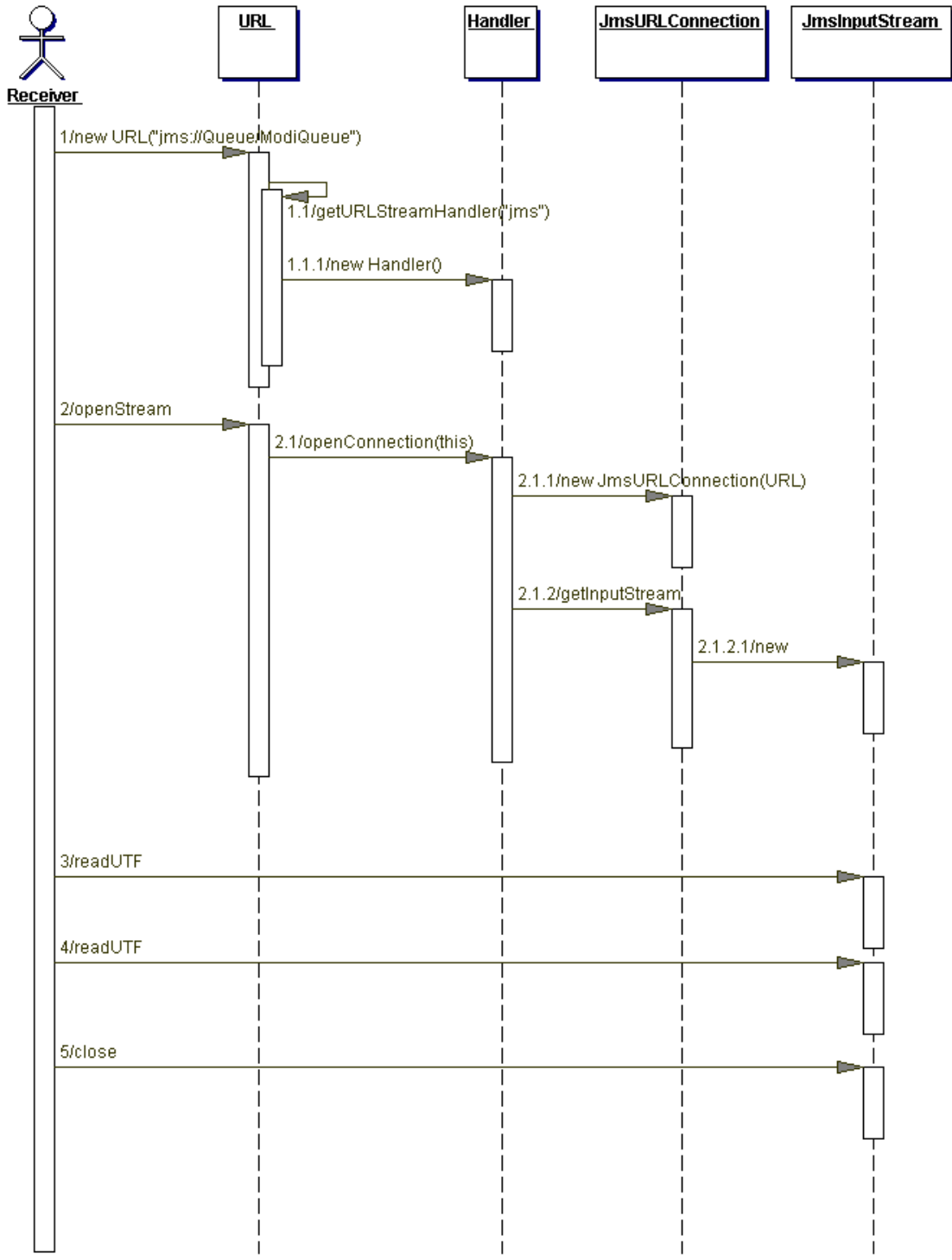


Figure 4: The Receiver Sequence Diagram

Since, our JMS protocol handler uses non-durable topic subscriptions, make sure that you start the Receiver before you run the Sender when using topics. Remember, JMS does not "remember" published messages for non-durable topic subscriptions. A useful enhancement to the protocol handler would to add the capability of having durable topic subscriptions. I recommend using

"request" properties to allow users to configure this, similar to how we allow persistence mode, time-to-live, and priority to be configured.

Summary

Throughout this book, I've emphasized on designing your enterprise applications without relying or getting married to a specific JMS provider. I've advocated total fidelity to the JMS specification. Now in this chapter, I've shown you a way to create a facade that decouples your application from JMS itself using Java's protocol handler architecture. An efficiency introduced as a result of this architecture is that only one queue and one topic connection will be created per VM since only one instance of a protocol handler exists per VM. Additional benefits of this include a reduced learning curve for most Java developers, since they are already familiar with using URLs and the ability to leverage a well-designed and time-tested architecture in the Java platform.

Chapter 10

Custom JSP tags for JMS²⁵

In this chapter, I continue my quest of isolating JMS from the end developer. This time, the focus is on the JSP developer. Most JSP developers that I've worked with love JSP because it is easy to use, yet gives them an awesome amount of power to design flexible webpages. These developers are excellent web designers because web designing is their passion. They are not interested in the details of n-tier architectures and distributed programming, let alone the intricacies and complexities of asynchronous processing with JMS. Having said that, it may be necessary to provide them with the power of asynchronous processing. The fact that both JSP and JMS are key pieces of J2EE reinforces this necessity.

JSP 1.1 introduced an extremely valuable new capability: the ability to define your own JSP tags. Such tags are called custom tags. Custom tags may be grouped into tag libraries and be reused in any number of JSP files. Custom tags allow complex programming logic to be boiled down into a set of simple and easy to use tags, which JSP developers can very easily use while developing content²⁶. Although custom tags require a little bit more setup work than would normally be required, the benefits definitely outweigh the costs.

In the previous chapter, I showed you how to create a custom protocol handler to abstract JMS from your organization and to save both time and money by not having to subject the entire development organization to a steep JMS learning curve. In this chapter, I will build upon this architecture and create custom JSP tags that allow JSP developers to use JMS without having to deal with JMS or even network programming as in the previous chapter.

For example, using the custom tag that I will create in this chapter, a JMS developer can send/publish a message as shown below

```
<jms:write destination="jms://Queue/ModiQueue" message="Hello World"/>
```

and can receive a message as follows

```
<jms:read destination="jms://Queue/ModiQueue"/>
```

Note the format of the `destination` attribute above. Remember, this is the format required by the underlying JMS protocol handler that the custom tags are based upon.

In general, a custom tag consists of three pieces:

1. A tag library descriptor that maps the XML tag to its implementation i.e. the tag handler class (see #2 below).
2. A tag handler class that defines the tag's behavior. When the web server comes across a custom tag, it relies on the handler class for that tag to do all the work.
3. JSP files that use the custom tag.

²⁵ To compile and execute the code in this chapter you will need a webserver that supports JSP 1.1, such as Tomcat 3.1

²⁶ JSP has allowed developers to access JavaBeans from the very beginning. So there has always been a way to isolate complex logic. However, custom tags have a couple of benefits over beans. First, beans cannot manipulate JSP content. Second, well-designed custom tags can allow a much simpler representation of complex operations.

The first two pieces are created by the "tag developer" and are only done once. The last piece is created by the JSP developer while designing the content and [re]uses the first two pieces.

The Custom Tags

The write Tag

We will create two custom tags. The first tag, `write`, is for sending/publishing messages to a destination. The definition of this tag is shown below.

```
<tag>
  <name>write</name>
  <tagclass>tags.jms.JmsWriteTag</tagclass>
  <info>Send/Publish a message</info>
  <attribute>
    <name>destination</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>message</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>ttl</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>priority</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>persistent</name>
    <required>false</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

The following information can be derived from the above tag definition.

- the tag name is `write`.
- the tag handler class is `tags.jms.JmsWriteTag`. This is the value of the `tagclass` element and must be the fully qualified class name of the tag handler implementation.
- the tag has five attributes.
 - `destination` is a mandatory attribute.
 - the other four attributes (`message`, `ttl`, `priority`, and `persistent`) are optional.
 - all the attributes can be JSP expressions of the form `<%= expression %>`, since the value of the `rtexprvalue` element is true for all attributes.

We've already seen that the value of the `destination` attribute is a valid `jms` protocol URL. This attribute must be specified. The message that is to be sent/published can be specified in two ways. The first way, which is also obvious from the above definition is via the `message` attribute.

For example,

```
<jms:write destination="jms://Queue/ModiQueue" message="Hello World"/>
```

The other way to specify the message is via the tag body. For example,

```
<jms:write destination="jms://Queue/ModiQueue">
    Hello World
</jms:write>
```

Note that in the latter case, I did not specify the `message` attribute. If I had specified the `message` attribute then the body would have been ignored. Why? Because that's the way I've coded the handler for this tag. I'll discuss the code in a few moments.

As we already know, a sender/publisher has control over the delivery mode, the priority, and the time-to-live of each message sent/published. The `write` tag allows the JSP developer to control these values via the `persistent`, `priority`, and `ttl` attributes. For example, suppose the JSP developer wants the message to expire in 60 seconds. This is achieved as follows.

```
<jms:write destination="jms://Queue/ModiQueue"
    message="Hello World" ttl="60000"/>
```

or,

```
<jms:write destination="jms://Queue/ModiQueue" ttl="60000">
    Hello World
</jms:write>
```

As mentioned above, these attributes are optional and their default values are shown in Table 1.

Attribute Name	Default Value
<code>persistent</code>	"true"
<code>priority</code>	"9"
<code>ttl</code>	"0"

Table 1: Optional attributes and their defaults

Now, let's take a look at the tag handler class for the `write` tag. When the web server comes across the `write` tag, this is the class that it will call to do all the work. A handler class must implement the `javax.servlet.jsp.tagext.Tag` interface. This is usually done by making the handler extend either the `javax.servlet.jsp.tagext.TagSupport` or `javax.servlet.jsp.tagext.BodyTagSupport` class. If the handler implementation needs to manipulate the tag body (as in this case), it needs to extend the `BodyTagSupport` class. After creating a new instance (or reusing an old instance from a pool) of a handler class, the web server informs the handler of all specified attributes. The handler class must follow the JavaBeans standard of naming property modifiers i.e. for an attribute called `destination`, the handler class must have a `setDestination` method defined as follows:

```
public void setDestination(String destination);
```

Accordingly, our handler implementation defines a 'setter' method for each attribute. The handler also overrides the `doAfterBody` method of the `BodyTagSupport` base class. This method is called by the web server when it is ready to process the body of the custom tag. Our implementation of this method checks to see if a message was specified via the `message` attribute and if so ignores the body. If no `message` attribute was specified then the body becomes the message as shown below.

```
// Was a message specified via the message attribute?
if( message == null ) {
    // No.
    // Get the body content
    BodyContent body = getBodyContent();
    // Set the message equal to this content.
```

```
        message = body.getString();
    }
```

Now, the `writeMessage` private method is called, which sends the message off to the specified destination. The `writeMessage` method implementation is similar to that of the `Sender` test program in previous chapter, so I'll not discuss it here again. The `doAfterBody` method returns the `SKIP_BODY` constant to inform the web server that the body has finished being processed.

Finally, the handler also overrides the `release` method. This method is called by the web server before reusing an instance of the handler class again. The `release` method is responsible for cleaning up this instance of the handler so that [the handler] can be used as if it were a brand new instance.

The complete implementation is shown below.

```
package tags.jms;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.net.*;

public class JmsWriteTag extends BodyTagSupport {

    private String destination = null;
    private String message = null;
    private String ttl = "0"; // live forever
    private String priority = "9"; // highest priority
    private String persistent = "true"; // persistent

    public void setDestination(String destination) {
        this.destination = destination;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public void setTtl(String ttl) {
        this.ttl = ttl;
    }

    public void setPriority(String priority) {
        this.priority = priority;
    }

    public void setPersistent(String persistent) {
        this.persistent = persistent;
    }

    public int doAfterBody() {
        if( message == null ) {
            BodyContent body = getBodyContent();
            message = body.getString();
        }
        writeMessage();
        return(this.SKIP_BODY);
    }

    public void release() {
        destination = null;
        message = null;
        ttl = "0";
        priority = "9";
        persistent = "true";
    }
}
```

```

    }
    private void writeMessage() {
        try {
            URL url = new URL(destination);
            URLConnection uc = url.openConnection();
            uc.setRequestProperty("persistent",persistent);
            uc.setRequestProperty("timeToLive",ttl);
            uc.setRequestProperty("priority",priority);
            OutputStream os = uc.getOutputStream();
            DataOutputStream dos =
                new DataOutputStream(os);
            dos.writeUTF(message);
            dos.flush();
            os.close();
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
    }
}

```

The read Tag

The second tag, `read`, is for receiving messages from a destination. Its definition is shown

```

<tag>
  <name>read</name>
  <tagclass>tags.jms.JmsReadTag</tagclass>
  <info>Receive a message</info>
  <attribute>
    <name>destination</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

```

This definition is similar to the definition for the `write` tag except that it is much simpler. The handler implementation for the `read` tag is `tags.jms.JmsReadTag`. It only has one mandatory attribute, `destination`. An example of using the `read` tag is shown below.

```
<jms:read destination="jms://Queue/ModiQueue"/>
```

Now let's take a look at the tag handler implementation for the `read` tag. Once again, this handler is very similar to that for the `write` tag, so I will only discuss the differences. Since the handler for this tag does not need to manipulate the tag body, it extends the `TagSupport` class and overrides the `doStartTag` method. The implementation of this method simply calls the private `readMessage` method and prints the results to the page as shown below.

```
pageContext.getOut().print(readMessage());
```

The `readMessage` method implementation is similar to that of the `Receiver` test program in the previous chapter and so I'll not discuss it here again. The method returns the `SKIP_BODY` constant to inform the web server to skip the body of the tag.

The complete implementation is shown below for reference.

```

package tags.jms;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
import java.net.*;

```

```

public class JmsReadTag extends TagSupport {
    private String destination = null;

    public void setDestination(String destination) {
        this.destination = destination;
    }

    public int doStartTag() {
        try {
            pageContext.getOut().print(readMessage());
        }
        catch( IOException e ) {
            e.printStackTrace();
        }

        // Skip tag body
        return(this.SKIP_BODY);
    }

    public void release() {
        destination = null;
    }

    private String readMessage() {
        try {
            URL url = new URL(destination);
            URLConnection uc = url.openConnection();
            InputStream is = url.openStream();
            DataInputStream dis = new DataInputStream(is);
            String message = dis.readUTF();
            is.close();
            return(message);
        }
        catch( Exception e ) {
            e.printStackTrace();
            return("");
        }
    }
}

```

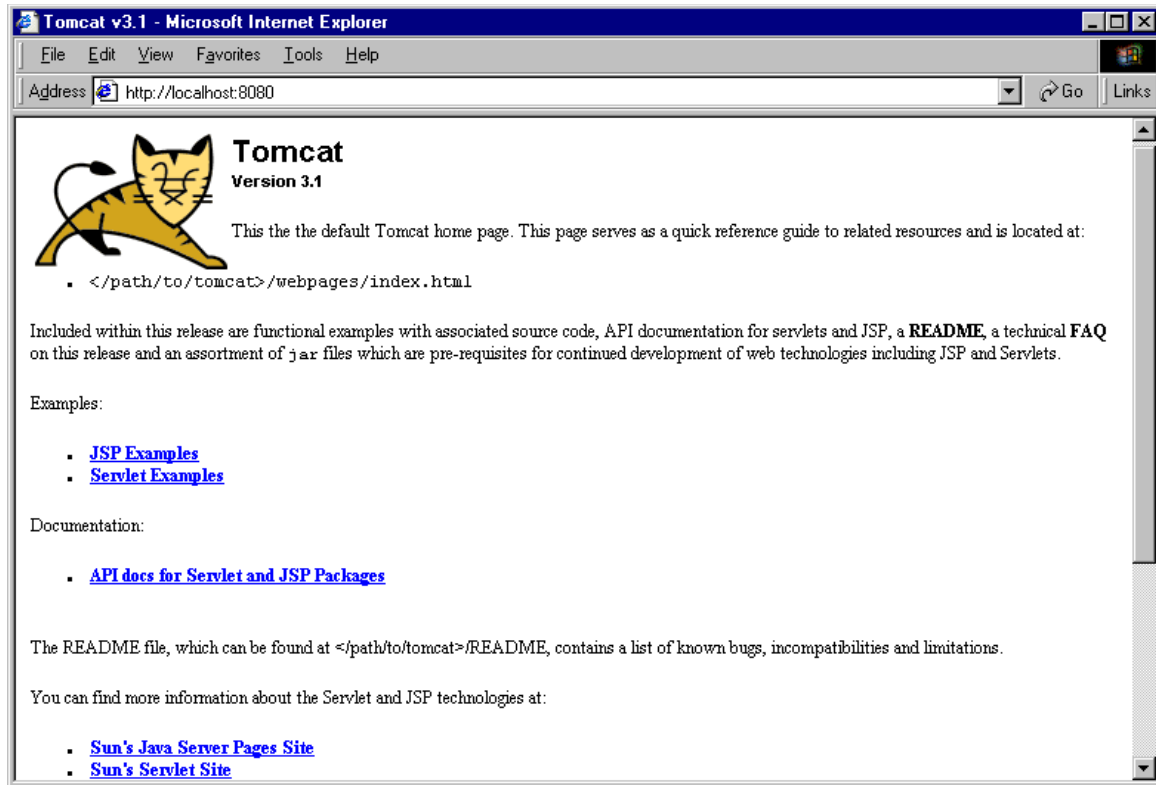
Testing it Out

To use the custom JSP tags created in this chapter, you will need a web server that supports JSP 1.1. For the remainder of this chapter, I will use Tomcat 3.1. It is a great web server and the price is right too!²⁷

Configuring Tomcat 3.1

Configuring Tomcat is easy once you figure out what to do *. In this section, I assume that you have already installed Tomcat and it is working. To verify my assumption, execute the startup batch file from a dos prompt, start your favorite browser, and point it to <http://localhost:8080/>. You should see a welcome page as shown in figure 1.

²⁷ Tomcat is available for free at <http://jakarta.apache.org/tomcat/index.html> under the Apache license and is part of the Jakarta project. The goal of the Jakarta project is to provide commercial quality Java server side solutions.



```

<taglib>
  <!-- after this the default space is
        "http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
  -->

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>jms</shortname>
  <uri></uri>
  <info>
    A tag library for simplifying the use of JMS from JSP
  </info>

  <tag>
    <name>write</name>
    <tagclass>tags.jms.JmsWriteTag</tagclass>
    <info>Send/Publish a message</info>
    <attribute>
      <name>destination</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>message</name>
      <required>>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>ttl</name>
      <required>>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>priority</name>
      <required>>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
      <name>persistent</name>
      <required>>false</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

  <tag>
    <name>read</name>
    <tagclass>tags.jms.JmsReadTag</tagclass>
    <info>Receive a message</info>
    <attribute>
      <name>destination</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>

</taglib>

```

4. Add a new context "jmsbook" to the existing server.xml file in the TOMCAT_HOME\conf directory. This is shown below.

```

<Context path="/jmsbook" docBase="webapps/jmsbook"
  debug="1" reloadable="true" >
</Context>

```

5. Modify the tomcat batch file in the TOMCAT_HOME\bin directory. We need to change the command line used to start the Java VM that hosts the Tomcat web server so that it can

find out JMS protocol handler. I discussed how to configure a VM for this in the previous chapter. Add the following lines to the tomcat batch file just after the section in the batch file where it sets up the classpath it requires (this is very near to the beginning of the batch file).

```
rem ***** Setup for JmsTags *****

rem Set the URL protocol handler to use
rem Note that I am also specifying a properties
rem file because I want to use Fiorano
set TOMCAT_OPTS= -Djava.protocol.handler.pkgs=jmsbook
                -Djmsbook.jms.propertiesFile=jmsUrl.properties

rem The path to my JMS protocol classes.
set CLASSPATH=%CLASSPATH%;E:\dev\test\

rem Modify the classpath to include the
rem classes for Sun's Java Message Queue and
rem FioranoMQ. We actually only need to include
rem only one of these, not both. For example
rem in this case we only need Fiorano.
set CLASSPATH=%CLASSPATH%;E:\Program
Files\JavaMessageQueue1.0\lib\jms.jar;E:\Program
Files\JavaMessageQueue1.0\lib\jmq.jar;E:\Program
Files\JavaMessageQueue1.0\lib\jmqadmin.jar
set CLASSPATH=%CLASSPATH%;E:\Program
Files\Fiorano\FioranoMQ\lib\fmprtl.zip

rem ***** Done *****
```

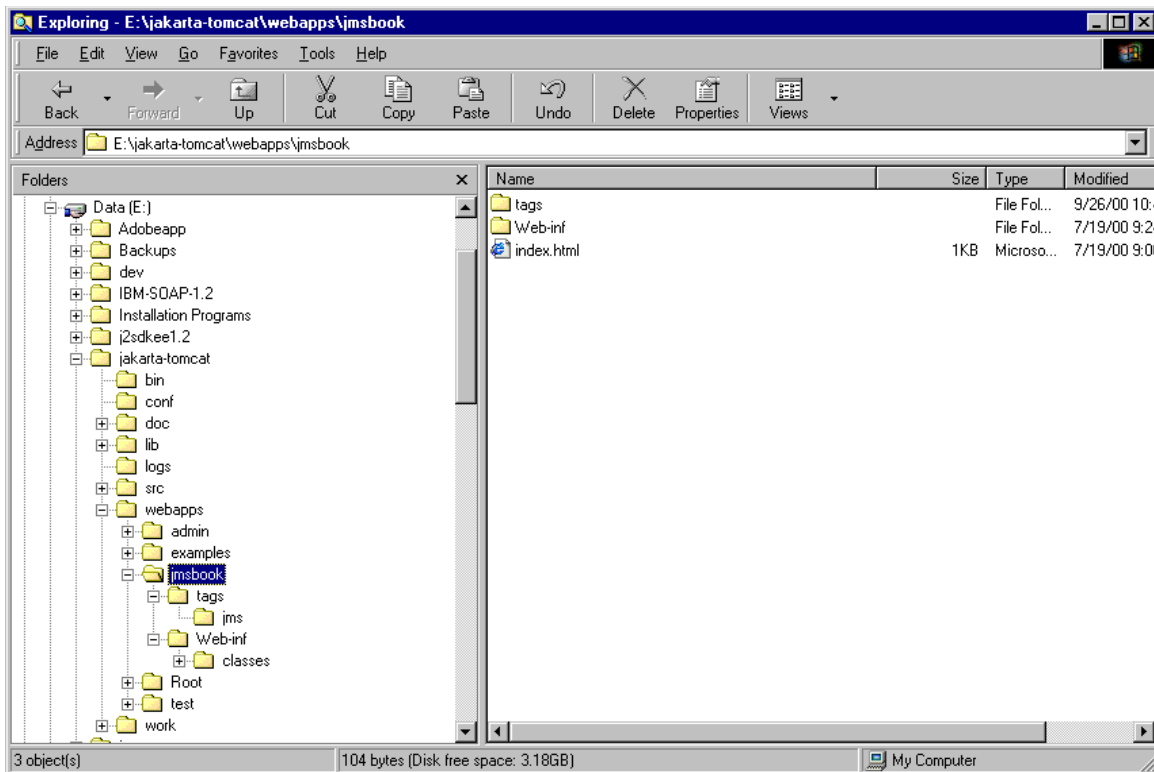


Figure 2: The directory structure

6. Copy the `jmsUrl.properties` file to the `TOMCAT_HOME\bin` directory. This is the same properties file that we used in the previous chapter to configure the JMS protocol handler to work with FioranoMQ rather than the default, which is Sun's Java Message Queue.

Compiling the Tag Handlers

To compile the handler classes execute the following command from a dos prompt.

```
javac -classpath E:\jakarta-tomcat\classes\;E:\jakarta-tomcat\lib\servlet.jar -d E:\jakarta-tomcat\webapps\jmsbook\WEB-INF\classes *.java
```

As I mentioned above, Tomcat is installed in the directory `jakarta-tomcat` on my E drive. The class path must include the `servlet.jar` file that contains all the servlet and jsp required classes. Note that I am directing the compiler to send the compiled classes to the directory `E:\jakarta-tomcat\webapps\jmsbook\WEB-INF\classes`. This is important because this is where Tomcat will look for the tag handler classes when I point my browser to a JSP file that contains custom tags.

The JSP file

Now let's take a look at a sample JSP file (`Test.jsp`) that tests our custom tags. The HTML code including the JSP code is shown below.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Testing JMS Tags</TITLE>

<!-- Important: Reference the tag library definition -->
<%@ taglib uri="JmsTags.tld" prefix="jms" %>

</HEAD>

<BODY>

<!-- Test the write tag with the message attribute -->
<jms:write destination="jms://Queue/ModiQueue" message="Hello World"/>

<!-- Read the message -->
Getting the first message...
<jms:read destination="jms://Queue/ModiQueue"/>

<!--
Test the write tag with the message as the tag body
Also, change the priority of the message from its default
to "3" and the ttl from "0" to "60000". Leave the
persistent status to its default ("true").
-->
<P>
<jms:write destination="jms://Queue/ModiQueue"
                priority="3" ttl="60000">
This is a body test.
</jms:write>

<!-- Read the message -->
Getting the second message...
<jms:read destination="jms://Queue/ModiQueue"/>
</P>

<!-- write with message as the attribute -->
<jms:write destination="jms://Queue/ModiQueue" message="Hello II"/>

<!--
Read the message, but instead of displaying it on the page
store it as the "id" of the <P> tag.
-->
```

To see if this worked, click view source when viewing the page and look at the html code. You should see a tag like:

```
<P id="Hello II" >
-->
Getting the third message...
<P id="<jms:read destination='jms://Queue/ModiQueue'/'>" >
hello
</P>

</BODY>
</HTML>
```

The test HTML/JSP code is very straightforward, especially with the detailed explanations in the comments, so I will skip most of the explanation. However, I would like to point out a key piece of this code: the reference to the tag library definition file, JmsTags.tld. Without this, the web server would have no way of knowing what to do with the custom tags, write and read.

And Finally?

The moment we've been waiting for? actually seeing the custom tags in action. First, start Tomcat. Now start your favorite browser and point it to the following URL

<http://localhost:8080/jmsbook/tags/jms/Test.jsp>. You should see an output similar to figure 3. Also, if you view the source HTML for that (Test.jsp) page in your browser, you should see HTML similar to that in figure 4. Note that the last <P> tag has the id attribute with the value "Hello II". Also look at the output generated in the console window in which Tomcat is running. You'll recognize a lot of this output from the previous chapter. This is because our custom tags build upon the work we did in the previous chapter.



Figure 3: The Output from Test.jsp

```
Test[1] - Notepad
File Edit Search Help
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Testing JMS Tags</TITLE>
</HEAD>
<BODY>
Getting the first message...
Hello World
<P>
Getting the second message...
This is a body test.
</P>
Getting the third message...
<P id="Hello II" >
hello
</P>
</BODY>
</HTML>
```

Figure 4: Test.jsp as delivered to the browser

Summary

Using JSP content developers can develop very powerful and dynamic web pages/applications. However, when coupled with JMS, JSP can become even more powerful. Using JMS from JSP does not have to be difficult either, because the architects of JSP have given us the power of extending it with custom tags. As I mentioned at the beginning of this chapter, custom tags allow the encapsulation of extremely complex and/or monotonous tasks into an easy to use tag (or set of tags) syntax. JSP developers love this because it feels very natural to use these custom tags. In this chapter I capitalized on the custom tag architecture of JSP to extend it with JMS capability. In addition, I leveraged the JMS protocol architecture developed in the previous chapter not only to make the development of the custom tags much easier, but also to make them independent of any JMS provider.

Chapter 11

Using JMS with EJB

Both JMS and EJB are key pieces of the J2EE platform, yet very little has been said/done to support using these APIs together²⁸, apart from allowing the access of JMS resources from EJB beans (or, EJBs). In this chapter, I will introduce an architecture for using JMS providers with application servers that support EJB 1.1. This chapter assumes a working knowledge of EJB 1.1.

JMS can be used with EJB in at least two ways: 1) as a resource available to EJBs and 2) to work around the synchronous nature of EJB 1.1. In this chapter, I will discuss both of these topics in detail starting with using JMS as a resource from an EJB.

JMS as a Resource

Since JMS is a part of J2EE, it can be safely accessed from within your EJBs. In other words, using JMS from within an EJB is really no different than using it from other Java applications. However, it is important that you understand the lifecycle of the EJB that you are accessing JMS resources from, since it is this lifecycle that helps determine the optimum way to obtain and release the JMS resources i.e. connections, sessions, etc. For example,

- In an entity bean, you should create all your JMS resources in the `ejbCreate` method and release all your JMS releases in the `ejbRemove` method. No action needs to be taken in the `ejbActivate` and `ejbPassivate` methods (unless of course you want to conserve your JMS resources).
- In a stateful session bean, you should create all your JMS resources in the `ejbCreate` and `ejbActivate` methods. These resources must be released in the `ejbRemove` and `ejbPassivate` methods. Additionally, in the instance variables for the JMS resources should be set to `null` in the `ejbPassivate` method as well. This requirement is imposed by the EJB specification for any nontransient and nonserializable instance fields in a stateful session bean.
- In a stateless session bean, you should create all your JMS resources in the `ejbCreate` method and release all your JMS releases in the `ejbRemove` method. Since a stateless session bean does not get activated and passivated like a stateful session bean does, no action needs to be taken in the `ejbActivate` and `ejbPassivate` methods.

These are actually the same rules that you would follow when using other resources such as JDBC from your EJBs. So, nothing really new here either. More interesting however, is how JMS transactions (refer to chapter 4 for a refresher) interact with EJB transactions. Let's take a look at an example stateless session bean that uses JMS as a resource and also demonstrates the use of JMS transactions from a typical EJB. The bean itself is very simple with just one method – `buy` – that is accessible to clients. I have deployed the bean such that this method requires a transaction and this transaction is managed by the container. Following are the noteworthy points of this example:

1. Since this is a stateless session bean, all the JMS resources required by the bean are initialized in the `ejbCreate` method as shown below.

```
public void ejbCreate() throws java.rmi.RemoteException {  
    // Initialize the JMS resources
```

²⁸ Actually EJB 2.0 takes care of this. Commercial application servers that support EJB 2.0 are not available as the time of this writing.

```

QueueConnectionFactory connectionFactory =
    getQueueConnectionFactory();
if( connectionFactory == null )
    throw new RuntimeException(
        "Can not get the Queue Connection Factory");

try {
    connection =
        connectionFactory.createQueueConnection();
    session = connection.createQueueSession(true,1);
    Queue queue = session.createQueue("inventory");
    inventory = session.createSender(queue);
    queue = session.createQueue("receivables");
    receivables= session.createSender(queue);
    connection.start();
}
.
.
.

```

Notice the call to the private method `getQueueConnectionFactory` to get the queue connection factory. In this case the implementation assumes that Fiorano's FioranoMQ is being used. To make this bean JMS neutral you can either use the strategy pattern discussed in chapter 8 or make the queue connection factory available in JNDI. Also note that I have created a transacted session by passing in a `true` value as the first parameter to the `createQueueSession` method.

2. Also, since this is a stateless session bean all JMS resources are released in the `ejbRemove` method as shown below.

```

public void ejbRemove() {
    try {
        // Release all JMS resources
        session.close();
        connection.close();
    }
    catch( JMSEException e ) {
    }
}

```

3. Finally, let's look at the `buy` method itself. The `buy` method gets two parameters: the buyer information and the order information. It is assumed that a client would call this method in response to a user trying to buy a product/service from an e-commerce site. This method starts by checking the order information by calling the `verifyOrder` method. If the order verification is successful, the `buy` method sends a message to the inventory queue. This step is shown below:

```

// First verify the order information
boolean b = verifyOrder(order);
// if the order is OK, send a message to
// the inventory queue.
if( b ) {
    TextMessage msg = session.createTextMessage();
    msg.setText("This is the message to the inventory " +
        "manager to decrement the inventory count...");
    inventory.send(msg);
}
.
.
.

```

Once this is done, the `buy` method checks the buyer information by calling the `verifyBuyer` method. If the buyer verification is successful, then a message is sent to the receivables queue. This step is shown below:

```
// Now verify the buyer information
b = verifyBuyer(buyer);
// if it's OK then send a message to the receivables queue.
if( b ) {
    TextMessage msg = session.createTextMessage();
    msg.setText("This is the message to the receivables " +
               "manager to adjust the receivables...");
    receivables.send(msg);
}
.
.
```

If both the verifications are successful, then this method also instructs the JMS session to commit the current transaction as follows:

```
// Tell JMS to commit the transaction.
session.commit();
```

This causes the two messages (one to the inventory queue and one to the receivables queue) to actually be sent. In other words, the session holds on to the messages until the transaction is committed. This is because the session may be asked to rollback the transaction in which case the messages must not be sent. In our case, if either one of the verifications fails, the `buy` method throws an exception. This exception is caught by the `buy` method itself, which then rolls back the current transaction on the session and on the bean as well. This is shown below.

```
.
.
.
// got an exception
// there was some problem!
catch( Exception e ) {
    try {
        // Tell JMS to rollback the transaction
        // This means that no messages "sent" within
        // this transactions will be sent.
        session.rollback();
    }
    catch( JMSEException e1 ) {
    }
    // Rollback the EJB transaction
    _context.setRollbackOnly();
.
.
.
```

This brings up an interesting point. There are actually two transactions going on here. One transaction is started by the EJB container and is the transaction within which the `buy` method is being run. The other transaction is automatically started up by the JMS session as soon as the first message is sent. These two transactions are independent of each other. That means committing or rolling back one transaction will not affect the other. Even if I used a bean-managed transaction instead of a container managed transaction, which means that I would manually create a transaction within the bean, I would still end up with two separate and independent transactions. It would be nice if the JMS session accepted a transaction to work within instead of always creating its own. For example, something like:

```
// create a new user transaction
```

```

Transaction tx = ?
// Tell the session to use this transaction
session.useTx(tx);
// send a few messages?

// Commit or roll back tx
// This would affect both the EJB and JMS.

```

But, you cannot do this right now; no harm in wishful thinking though. The complete source for our stateless session bean is shown below.

```

// The Bean implementation.
import javax.jms.*;

public class BuyBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext _context = null;

    // the JMS resources that we will need to
    // hold on to.
    private QueueConnection connection = null;
    private QueueSession session = null;
    private QueueSender inventory = null;
    private QueueSender receivables = null;

    public void setSessionContext(
        javax.ejb.SessionContext sessionContext) {
        _context = sessionContext;
    }

    public void ejbCreate() throws java.rmi.RemoteException {

        // Initialize the JMS resources
        QueueConnectionFactory connectionFactory =
            getQueueConnectionFactory();
        if( connectionFactory == null )
            throw new RuntimeException(
                "Can not get the Queue Connection Factory");

        try {
            connection = connectionFactory.createQueueConnection();
            session = connection.createQueueSession(true,1);
            Queue queue = session.createQueue("inventory");
            inventory = session.createSender(queue);
            queue = session.createQueue("receivables");
            receivables = session.createSender(queue);
            connection.start();
        }
        catch( JMSEException e ) {
            if( e.getLinkedException() != null )
                throw new RuntimeException(
                    e.getLinkedException().getMessage());
            else
                throw new RuntimeException(e.getMessage());
        }
    }

    public void ejbRemove() {
        try {
            // Release all JMS resources
            session.close();
            connection.close();
        }
        catch( JMSEException e ) {
        }
    }

    public void ejbActivate() {

```



```

}

public void ejbPassivate() {
}

public boolean buy(BuyerInfo buyer, OrderInfo order)
throws java.rmi.RemoteException
{
    // First verify the order information
    try {
        boolean b = verifyOrder(order);
        // if the order is OK, send a message to
        // the inventory queue.
        if( b ) {
            TextMessage msg = session.createTextMessage();
            msg.setText("This is the message to the inventory " +
                "manager to decrement the inventory count...");
            inventory.send(msg);
        }
        // otherwise just throw an exception
        // why..., you'll see.
        else {
            throw new Exception();
        }

        // Now verify the buyer information
        b = verifyBuyer(buyer);
        // if it's OK then send a message to the receivables queue.
        if( b ) {
            TextMessage msg = session.createTextMessage();
            msg.setText("This is the message to the receivables " +
                "manager to adjust the receivables...");
            receivables.send(msg);
        }
        // otherwise just throw an exception
        // why..., you'll see.
        else {
            throw new Exception();
        }

        // Tell JMS to commit the transaction.
        session.commit();

        // Buy completed...
        return(true);
    }
    // got an exception
    // there was some problem!
    catch( Exception e ) {
        try {
            // Tell JMS to rollback the transaction
            // This means that no messages "sent" within
            // this transactions will be sent.
            session.rollback();
        }
        catch( JMSEException e1 ) {
        }
        // Rollback the EJB transaction
        _context.setRollbackOnly();
        // Buy failed.
        return(false);
    }
}

// Get the queue connection factory
// This code is specific to Fiorano.
private QueueConnectionFactory getQueueConnectionFactory() {
    try {

```

```

        fiorano.jms.rtl.FioranoInitialContext ic = null;
        ic = new fiorano.jms.rtl.FioranoInitialContext();
        ic.bind();
        QueueConnectionFactory factory =
            (QueueConnectionFactory)ic.lookup("primaryqcf");
        ic.dispose();
        return(factory);
    }
    catch( JMSEException e ) {
        if( e.getLinkedException() != null )
            throw new RuntimeException(
                e.getLinkedException().getMessage());
        else
            throw new RuntimeException(e.getMessage());
    }
}

// verification fails for "John Doe"
// no hard feelings Mr. Doe :)
private boolean verifyBuyer(BuyerInfo buyer) {
    if( buyer.name.equalsIgnoreCase("John Doe") )
        return(false);
    else
        return(true);
}

// verification fails for itemNumbers less than 0
// or greater than 1000 or if the qty is less than 0.
private boolean verifyOrder(OrderInfo order) {
    if( order.itemNumber < 0 || order.itemNumber > 1000
        || order.qty < 0 )
        return(false);
    else
        return(true);
}
}

```

To test this out, assume that a client executes the following.

```

// Assume that "home" is a valid reference
// to the home for this bean
// Get the bean.
Buy buyer = home.create();

// First call
BuyerInfo buyerInfo = new BuyerInfo("Tarak Modi");
// OrderInfo takes an item number and quantity
OrderInfo orderInfo = new OrderInfo(501,10);
boolean b = buyer.buy(buyerInfo, orderInfo);

// Second call
buyerInfo = new BuyerInfo("John Doe");
orderInfo = new OrderInfo(501,10);
b = buyer.buy(buyerInfo, orderInfo);

// Third call
buyerInfo = new BuyerInfo("Tarak Modi");
orderInfo = new OrderInfo(1501,18);
b = buyer.buy(buyerInfo, orderInfo);

```

In the above code snippet, only the first call will succeed i.e. return a `true` value²⁹. As a result both the inventory and receivables queues will only contain one message each. This is because whenever the `buy` method fails, it rolls back the JMS session transaction, which causes the JMS session to discard any messages sent within that transaction instead of actually sending them.

Now, let's take a look at how JMS can be used as more than just a [powerful] resource in conjunction with EJB. More specifically, I would like to present an architecture that allows clients to access EJBs asynchronously.

Asynchronous EJB

Communication with and/or between EJBs is a synchronous process, with the caller blocking till the call is completed. But, there are many situations when such blocking is not desirable. For example, consider an EJB that provides a backup service. The backup may take a long time, possibly several hours. It seems silly that the caller has to block for all this time since the backup could be done in the background (i.e. asynchronously), thus allowing the client to do other things. That is where JMS fits into the picture.

The "Wrong" Architecture

Let's continue with the example of an EJB that provides a backup service: the Backup Server. For now, let's assume this is implemented as a Stateless Session bean. The home and remote interfaces for this bean are defined below:

```
// The home interface.
public interface BackupHome extends javax.ejb.EJBHome {
    Backup create()
        throws java.rmi.RemoteException, javax.ejb.CreateException;
}

// The Bean specific interface.
public interface Backup extends javax.ejb.EJBObject {
    public void backup() throws java.rmi.RemoteException;
}
```

A typical stateless session bean is defined below:

```
// The Bean implementation.
public class BackupBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext _context;

    public void setSessionContext(
        javax.ejb.SessionContext sessionContext) {
        _context = sessionContext;
    }

    public void ejbCreate() throws java.rmi.RemoteException {
    }

    public void ejbRemove() {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void backup() throws java.rmi.RemoteException {
        // Complex Backup logic.
    }
}
```

²⁹ The second call fails because of the buyer's name and the third call fails because of an invalid item number.

```

    } // Takes a very long time.
}

```

Now let's use JMS to make the actual backup operation asynchronous. We can't just spawn a thread from the `backup` method because EJBs are not allowed to do so. That is, the following is illegal:

```

// Illegal Code snippet
public void backup()throws java.rmi.RemoteException {
    new BackupThread().start();
}

// The backup thread
private class BackupThread extends Thread {
    public void run() {
        // Complex Backup logic.
        // Takes a very long time.
    }
}

```

The intuitive way to use JMS from our backup EJB is to package up the backup request into a JMS message and send this message to well-known destination. The stateless session bean would have also registered itself as an asynchronous message listener with that well-known destination and so at some point the JMS provider will invoke the `onMessage` method on the bean. Coding this would be very easy as shown below.

```

// The Bean implementation with JMS - The WRONG way!
import javax.jms.*;
public class BackupBean
    implements javax.ejb.SessionBean, MessageListener {

    private javax.ejb.SessionContext _context;

    public void setSessionContext(
        javax.ejb.SessionContext sessionContext) {
        _context = sessionContext;
    }

    public void ejbCreate() throws java.rmi.RemoteException {
        // Register this class as a message listener
        // for the well-known destination.
    }

    public void ejbRemove() {
        // Unregister this class as a message listener
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void backup()throws java.rmi.RemoteException {
        // Send a message indicating
        // that a backup is required.
        // i.e. schedule a backup
        // No backup is performed here
    }

    // This method is called by the JMS provider.
    // The message received here is the same
    // message that the "backup" method sent
}

```

```

    // to the provider.
    public void onMessage(Message m) {
        // Complex Backup logic.
        // Takes a very long time.
    }
}

```

Simple enough, but please do not do this! We have just violated a major rule of EJB: a bean must never be accessed directly, but only through its remote interface. In the above architecture, JMS bypasses the container and directly invokes the `onMessage` method on the bean. A couple of problems (out of many) that could arise are:

1. The bean could already be in use by the container. Remember, stateless session beans are not thread-safe.
2. The bean could be in a "pooled" state.

So how do you [correctly and legally] use JMS with EJB. That is the topic of the next section.

A "Correct" Alternative Architecture

An accepted solution to this dilemma is to use a delegation model based architecture. Figure 1 shows the proposed architecture. There are several key differences from the previous architecture.

1. The client does not directly interact with the Backup EJB. Instead it sends a message to the well-known destination.
2. A new "daemon" known as the AsyncDelegator has been introduced. This process receives messages from the destination.
3. The AsyncDelegator invokes the appropriate method on the backup bean. It does this via the remote interface via the container. This architecture places a new requirement on EJBs in that their remote interface must contain the following two methods required by the AsyncDelegator. These methods are shown below as part of the new interface for the Backup bean.

```

// The Bean specific interface.
public interface Backup extends javax.ejb.EJBObject {
    // All "Async" beans MUST define these two methods
    public void onMessage(String soapXml)
        throws java.rmi.RemoteException;
    public void invokeService(java.util.Map map)
        throws java.rmi.RemoteException;
}

```

I will be using the Inprise Application Server. There is one line of code in AsyncDelegator that is Inprise specific. I will point this out during the discussion.

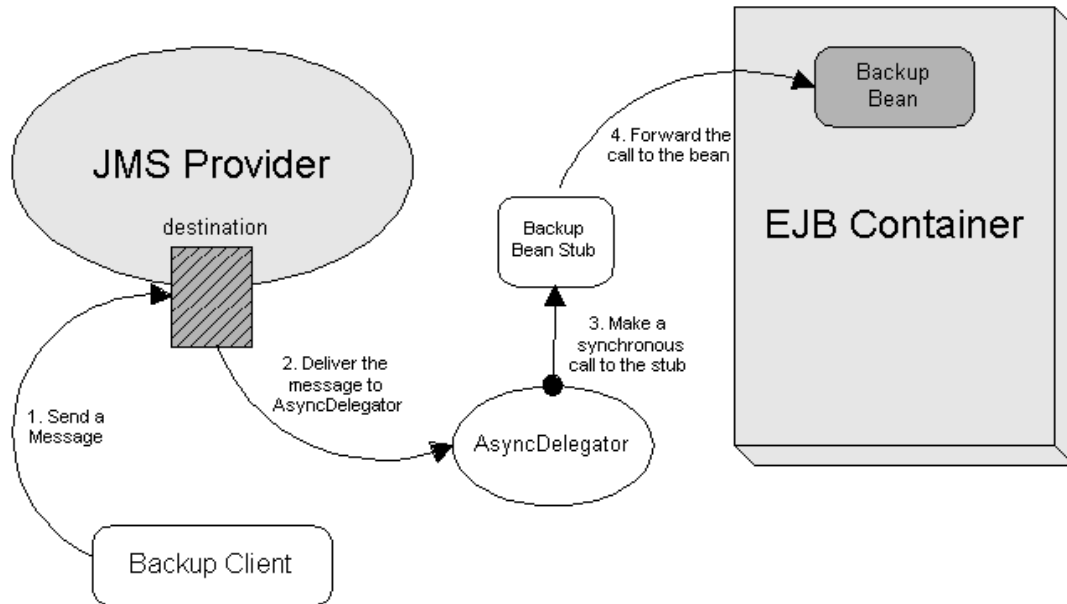


Figure 1: The proposed architecture for using JMS with EJB

The AsyncDelegator

This is a central piece in the proposed architecture, so let's take a detailed look at it.

`AsyncDelegator` has two primary (and related) responsibilities.

1. To receive messages from well-known destinations.
2. To invoke the appropriate method (`onMessage` or `invokeService`) on the appropriate beans in response to receiving a message.

To receive messages from JMS destinations, `AsyncDelegator` must interface with a JMS provider that will be used by the client as well (see figure 1). In my implementation I am using Sun's Java Message Queue for simplicity. However, as I've shown in chapters 8 and 9, it is very easy to break this dependency. I leave this as an exercise for you.

`AsyncDelegator` knows which bean to invoke the method on for each well known destination from a properties file which is a required argument for startup. This properties file specifies a mapping between destinations and EJBs. Each property name is actually the name of a valid JMS queue. The value of each property is a comma delimited string of home name and fully qualified home classname pairs. The home name and home class name are separated by a '|'. Following is the properties file that I am using.

```
# delegator.properties
BackupQueue=backup|BackupHome
```

Here the queue name is "BackupQueue". Also, this properties file informs `AsyncDelegator` that whenever a message arrives on this queue, it should invoke the appropriate method on an EJB whose home name is "backup". The home class returned by the container corresponding to the name "backup" will be of type "BackupHome".

For each destination (i.e. queue) in the properties file, `AsyncDelegator` creates an instance of `AsyncThread`. This thread receives the name of the queue and a list of host name and host classname pairs as its constructor parameters. This is shown below.

```
// For each property, create a thread that receives
```

```

// messages on the queue and forwards each message
// to a bean obtained from each home specified
String destination = (String)enum.nextElement();
String homes = props.getProperty(destination);
if( homes.equals("") )
    continue;
LinkedList homeList = new LinkedList();
StringTokenizer st = new StringTokenizer(homes, ",");
while( st.hasMoreTokens() ) {
    homeList.add(st.nextToken());
}

// Create a new thread?
AsyncThread t = new AsyncThread(destination, homeList);

```

Each instance of `AsyncThread` waits for messages from the specified destination i.e. listens for messages as shown below.

```
Message m = receiver.receive(1000);
```

From now on, I will refer to these threads as listener threads.

Creating the EJB

When a message is received, the listener thread must invoke the appropriate method on each bean specified in the properties file. But before it can invoke the method on a bean, it must create the bean. This is shown below.

```

// For each specified home, get an EJB and
// invoke the "appropriate" method.
for( int i=0; i<homeList.size(); i++ ) {
    String homeItem = (String)homeList.get(i);
    String homeName =
        homeItem.substring(0,homeItem.indexOf("|"));
    String homeClassname =
        homeItem.substring(homeItem.indexOf("|")+1);

    System.out.println("Forwarding message to " +
        homeName + "of class type " + homeClassname);

    // Some AppServers may require additional
    // configuration for the Initial context.
    javax.naming.Context context = new
        javax.naming.InitialContext();

    Object ref = context.lookup(homeName);
    Class homeClass = Class.forName(homeClassname);

    // This is required for Inprise AppServer since
    // it uses RMI-IIOP as its transport protocol.
    Object home =
        javax.rmi.PortableRemoteObject.narrow(ref,homeClass);

    // Get the create method
    Method methodCreate = homeClass.getMethod("create",
        new Class[]{});

    // Finally, we have the bean!
    Object ejb = methodCreate.invoke(home, new Object[]{});
    .
    .
    .

```

In the above code snippet, there is one line of code that is Inprise specific. Once I obtain a stub to the home object, I "narrow" (i.e. cast) it to the correct class as shown below.

```

// This is required for Inprise AppServer since
// it uses RMI-IIOP as its transport protocol.
Object home = javax.rmi.PortableRemoteObject.narrow(ref,homeClass);

```

Actually, this is not really Inprise specific, but is required by any application server that uses RMI-IIOP as the transport protocol for EJB. To break this dependency, we can use a similar strategy as we did for breaking the dependency on any specific JMS provider. Make `AsyncDelegator` configurable so that it relies on an external helper class to get the home object. This external helper class may contain application server specific code.

Once, we have the home object, we can invoke the `create` method on it. Notice that I look for the `create` method with no parameters. The reason behind this is that I assume that stateless session beans will be used to provide the asynchronous services, since they are the most ideal for this purpose. Stateless session beans are dedicated to a client only for the life of a method call and they do not contain/hold any client specific state. Plus they are the most efficiently pooled bean. All these characteristics make stateless session beans ideal for providing asynchronous services.

Invoking the method

Until now, I've been saying "invoke the appropriate method on the bean." So what is the appropriate method? It depends on the type of the message received. If the message is a `TextMessage` then the `onMessage` method is invoked passing the string contents of the message as the parameter to the method as shown below.

```

// TextMessage means get the containing text
// and invoke the "onMessage" method on the EJB.
if( m instanceof TextMessage ) {
    Method methodOnMessage =
        ejb.getClass().getMethod("onMessage",new
            Class[]{String.class});

    // Create a new "invoker" thread to
    // actually invoke the method.
    Thread t = new InvokerThread(methodOnMessage,ejb,
        new Object[] {(TextMessage)m.getText()});
    .
    .
    .

```

If on the other hand the message is a `MapMessage` then the `invokeService` method is called. This method is passed an instance of a `Map` that mirrors the body of the `MapMessage`, with the exception that primitives in the message are "objectified" in the map. For example, an `int` in the message is objectified into an `Integer` in the map.

```

// MapMessage means create a Map that mirrors
// the message and invoke the "invokeService"
// method on the EJB.
else if( m instanceof MapMessage ) {
    MapMessage mapMsg = (MapMessage)m;
    HashMap map = new HashMap();
    Enumeration mapNames = mapMsg.getMapNames();
    while( mapNames.hasMoreElements() ) {
        String name = (String)mapNames.nextElement();
        map.put(name,mapMsg.getObject(name));
    }
    Method methodInvokeService =
        ejb.getClass().getMethod("invokeService",
            new Class[]{Map.class});

    // Create a new "invoker" thread to
    // actually invoke the method.
    Thread t = new InvokerThread(methodInvokeService,ejb,

```



```

new Object[] {map});
.
.
.

```

You will notice that instead of invoking the `[onMessage or invokeService]` method directly from the listener thread, I create another thread that I call the invoker thread, which is an instance of the class `InvokerThread`. The reason behind this is that the method may take a long time to finish and I do not want my listener thread to block for all that time. For example, a backup may take several hours. Instead, the listener thread creates a new thread that actually invokes the method and blocks, while the listener thread is free to listen for other messages. This design pattern is similar to the one used by stream (TCP) socket programmers, where there is a separate socket that listens/waits for client connections and for each client connection received, the listener socket creates a new socket that actually communicates with the client.

Shutting down the AsyncDelegator

`AsyncDelegator` creates an instance of the class `StopThread`. This thread monitors the standard input (i.e. keyboard) for a "q" followed by return key. When this sequence of keystrokes is received, this thread calls the `shutdown` method, which is responsible for ensuring a clean shutdown. To ensure a clean shutdown, the `shutdown` method needs cooperation from the rest of the `AsyncDelegator` code in two ways.

1. Whenever a new thread is created it must be added to the `threads` list. All access to the `threads` list must be synchronized. The code snippet illustrates how the listener thread creates a new invoker thread.

```

// Create a new "invoker" thread to
// actually invoke the method.
Thread t = new InvokerThread(methodOnMessage,ejb,
    new Object[] {((TextMessage)m).getText()});
// Add it to the threads list?
synchronized(threads) {
    threads.add(t);
}
// and then start the thread.
t.start();

```

2. Just before a thread terminates itself, it must remove itself from and then call `notify` on the `threads` list. Once again, accessing the `threads` list must be synchronized. Synchronization here is not just for thread safety but also to ensure that the Java runtime allows the `notify` method to be called. Also, a thread is responsible for cleaning up all allocated resources, which it does before removing itself from the `threads` list. For example, the following code snippet illustrates how a listener thread terminates.

```

try {
    // perform cleanup.
    if( receiver != null ) {
        receiver.close();
        session.close();
    }
}
catch( Exception e ) {
    e.printStackTrace();
}

// Remove myself from the threads list
// and call notify.
synchronized(threads) {
    // This thread is done...
    threads.remove(this);
    threads.notify();
}

```

```
}
```

The `shutdown` method also needs each listener thread to periodically call the `running` method and if this method returns a `false` value, the listener thread must terminate itself in the way specified in (2) above.

Now let's look at the `shutdown` method. It proceeds in three steps:

1. Set the `_shutdown` flag to true as shown below.

```
synchronized(_shutdown) {
    _shutdown = new Boolean(true);
}
```

This is the flag that the `running` method uses to determine if the `AsyncDelegator` is shutting down, as shown below.

```
private static boolean running() {
    // If we've been told to shutdown, return a false value.
    synchronized(_shutdown) {
        return(!_shutdown.booleanValue());
    }
}
```

2. Now wait for all the threads to terminate as shown below

```
// Now wait for all the threads to end.
// There are two types of threads
// 1. The "listener" threads to receive messages from queues.
// 2. The "invoker" threads that actually invoke methods
// on EJBs.
synchronized(threads) {
    while( !threads.isEmpty() ) {
        try {
            threads.wait();
        }
        catch( Exception e ) {
        }
    }
}
```

3. Finally, close the JMS connection as shown below.

```
// Close the JMS connection.
try {
    connection.close();
}
catch( Exception e ) {
    e.printStackTrace();
}
```

The complete implementation of the `AsyncDelegator` is shown below.

```
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
import java.util.LinkedList;
import java.util.StringTokenizer;
import java.util.HashMap;
import java.util.Map;
import java.util.Enumeration;
import java.lang.reflect.Method;
```

```

import javax.jms.TextMessage;
import javax.jms.Message;
import javax.jms.MapMessage;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSession;
import javax.ejb.EJBHome;
import javax.rmi.PortableRemoteObject;

public class AsyncDelegator {
    private static QueueConnection connection = null;
    private static Boolean _shutdown = new Boolean(false);
    private static LinkedList threads = new LinkedList();

    public static void main(String[] args) {
        if( args.length != 1 || !new File(args[0]).exists() ) {
            System.err.println("The delegator takes one parameter," +
                " which is a valid properties file name.");
            System.exit(0);
        }

        try {
            // load the properties from the file
            Properties props = new Properties();
            props.load(new FileInputStream(args[0]));

            // Get the queue connection factory.
            // This is the only "Sun Java Message Queue" specific code.
            QueueConnectionFactory connectionFactory =
                new com.sun.messaging.QueueConnectionFactory();

            // Use the factory to create the queue connection.
            connection = connectionFactory.createQueueConnection();
            connection.start();

            // Enumerate the properties.
            // Each property name is the name of a queue to receive
            // messages on. The value of each property is a
            // comma-delimited string of pairs of home names
            // and home classnames.
            // The home name and the classname are separated by a "|".
            // For example, a valid property could be:
            //     BackupQueue=BackupServer1|BackupHome,
            //     BackupServer2|BackupHome,
            //     FastBackupServer|FastBackupHome
            // Using such a property file it is possible to set up a
            // many to many relationship between queues and EJBs.
            java.util.Enumeration enum = props.keys();
            while( enum.hasMoreElements() ) {
                // For each property, create a thread that receives
                // messages on the queue and forwards each message
                // to a bean obtained from each home specified
                String destination = (String)enum.nextElement();
                String homes = props.getProperty(destination);
                if( homes.equals("") )
                    continue;
                LinkedList homeList = new LinkedList();
                StringTokenizer st = new StringTokenizer(homes, ",");
                while( st.hasMoreTokens() ) {
                    homeList.add(st.nextToken());
                }

                // Create a new thread and remember it.
                AsyncThread t = new AsyncThread(destination, homeList);
                // Don't need to synchronize access to "threads" yet.
                // We will need to later.
            }
        }
    }
}

```

```

        threads.add(t);
        t.start();
    }
}
catch( Exception e ) {
    e.printStackTrace();
    shutdown();
}

// This thread will hang around till a user presses
// q followed by a return. At that point this thread
// will tell the delegator to shutdown.
new StopThread().start();
System.out.println("AsyncDelegator is Ready.");
System.out.println("Press q<enter> at any point " +
    "to quit program.");
}

private static boolean running() {
    // If we've been told to shutdown, return a false value.
    synchronized(_shutdown) {
        return(!_shutdown.booleanValue());
    }
}

private static void shutdown() {
    // The delegator must shut down now.

    // Make it so...
    synchronized(_shutdown) {
        _shutdown = new Boolean(true);
    }

    // Now wait for all the threads to end.
    // There are two types of threads
    // 1. The "listener" threads to receive messages from queues.
    // 2. The "invoker" threads that actually invoke methods
    // on EJBs.
    synchronized(threads) {
        while( !threads.isEmpty() ) {
            try {
                threads.wait();
            }
            catch( Exception e ) {
            }
        }
    }

    // Close the JMS connection.
    try {
        connection.close();
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
}

// This is the "listener" thread that receives messages
// from a queue.
private static class AsyncThread extends Thread {
    // which queue?
    private String destination;
    // A list of home name and classname pairs.
    private LinkedList homeList = null;

    public AsyncThread(String destination, LinkedList homeList) {
        super(destination);
        this.destination = destination;
    }
}

```

```

        this.homeList = homeList;
    }

    public void run() {
        System.out.println("Thread " + this.getName() +
            " started.");

        QueueSession session = null;
        QueueReceiver receiver = null;
        try {
            // JMS setup work...
            session = connection.createQueueSession(false,1);
            Queue queue = session.createQueue(destination);
            receiver = session.createReceiver(queue);
            // Continue till asked to stop.
            while( AsyncDelegator.running() ) {
                // Wait for a message...
                Message m = receiver.receive(1000);
                if( m == null )
                    continue;

                System.out.println("Received a Message");

                // For each specified home, get an EJB and
                // invoke the "appropriate" method.
                for( int i=0; i<homeList.size(); i++ ) {
                    String homeItem = (String)homeList.get(i);
                    String homeName =
                        homeItem.substring(0,homeItem.indexOf("|"));
                    String homeClassname =
                        homeItem.substring(homeItem.indexOf("|")+1);

                    System.out.println("Forwarding message to " +
                        homeName + "of class type " +
                        homeClassname);

                    // Some AppServers may require additional
                    // configuration for the Initial context.
                    javax.naming.Context context =
                        new javax.naming.InitialContext();

                    Object ref = context.lookup(homeName);

                    Class homeClass = Class.forName(homeClassname);

                    // This is required for Inprise AppServer since
                    // it uses RMI-IIOP as its transport protocol.
                    Object home =
                        javax.rmi.PortableRemoteObject.narrow(ref,
                            homeClass);

                    Method methodCreate =
                        homeClass.getMethod("create", new Class[]{});
                    Object ejb = methodCreate.invoke(home,
                        new Object[]{});

                    // TextMessage means get the containing text
                    // and invoke the "onMessage" method
                    //on the EJB.
                    if( m instanceof TextMessage ) {
                        Method methodOnMessage =
                            ejb.getClass().getMethod("onMessage",
                                new Class[]{String.class});

                        // Create a new "invoker" thread to
                        // actually invoke the method.
                        Thread t =
                            new InvokerThread(methodOnMessage, ejb,

```

```

        new Object[] { ((TextMessage)m).getText() });
        synchronized(threads) {
            threads.add(t);
        }
        t.start();
    }
    // MapMessage means create a Map that mirrors
    // the message and invoke the "invokeService"
    // method on the EJB.
    else if( m instanceof MapMessage ) {
        MapMessage mapMsg = (MapMessage)m;
        HashMap map = new HashMap();
        Enumeration mapNames =
            mapMsg.getMapNames();
        while( mapNames.hasMoreElements() ) {
            String name =
                (String)mapNames.nextElement();
            map.put( name, mapMsg.getObject( name ) );
        }
        Method methodInvokeService =
            ejb.getClass().getMethod("invokeService",
                new Class[] { Map.class });
        // Create a new "invoker" thread to
        // actually invoke the method.
        Thread t =
            new InvokerThread( methodInvokeService, ejb,
                new Object[] { map } );
        synchronized(threads) {
            threads.add(t);
        }
        t.start();
    }
}
}
}
}
}
}
}
}
}
}

catch( Exception e ) {
    System.out.println("Exception in thread: " +
        Thread.currentThread().getName());
    e.printStackTrace();
}

try {
    // cleanup.
    if( receiver != null ) {
        receiver.close();
        session.close();
    }
}
catch( Exception e ) {
    e.printStackTrace();
}

synchronized(threads) {
    // This thread is done...
    threads.remove(this);
    threads.notify();
}

System.out.println("Thread " + this.getName() + " done.");
}
}

// The "shutdown" thread.
private static class StopThread extends Thread {
    public void run() {
        try {
            // Wait for the user to enter <q> followed by a return.
            while( true ) {

```

```

        int character = System.in.read();
        if( character != -1 && character == 113 ) {
            synchronized(this) {
                System.out.println("Preparing for
                                   shutdown.");
                // shutdown...
                AsyncDelegator.shutdown();
                System.out.println("Shutdown complete.");
                return;
            }
        }
        java.lang.Thread.sleep(1000);
    }
}
catch( Exception e ) {
    e.printStackTrace();
}
}

// The "invoker" thread.
private static class InvokerThread extends Thread {
    private Method method;
    private Object target;
    private Object[] args;

    // setup parameters in constructor
    public InvokerThread(Method method, Object target,
                        Object[] args) {

        this.method = method;
        this.target = target;
        this.args = args;
    }

    // invoke the method
    public void run() {
        try {
            System.out.println("Invoker Thread is invoking method "
                + method.getName());
            method.invoke(target, args);
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
        finally {
            synchronized(threads) {
                // This thread is done.
                threads.remove(this);
                threads.notify();
            }
        }
    }
}
}
}
}
}

```

The Architecture in action

I assume that you have Inprise Application server and Sun's Java Message Queue installed and working. I have installed both of these products in my E:\Program Files\ directory.

The Backup EJB

The Backup EJB is a stateless session bean as before and it provides the implementation for the two methods in the Backup remote interface. The implementation of the `onMessage` method is very straightforward; it simply prints the text string to standard out. In reality this message could be fairly complex, such as a SOAP packet.

The `invokeService` method implementation is more interesting. It gets a parameter of type `java.util.Map`, which contains the following key information:

- The name of the service to invoke. In our case, the only valid service is "backup". The name of the service is completely arbitrary, but documented.
- Any parameters if necessary. The parameters are identified by name and depend on the name of the service. For example, the "backup" service requires a "days" parameter that determines which files get backed up.

The complete implementation of the Backup bean is shown below. Notice that the bean is completely unaware of JMS.

```
// The Bean implementation.
// Notice that it has no knowledge of JMS

public class BackupBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext _context;

    public void setSessionContext(
        javax.ejb.SessionContext sessionContext) {
        _context = sessionContext;
    }

    public void ejbCreate() throws java.rmi.RemoteException {
    }

    public void ejbRemove() {
    }

    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    // One of the mandatory methods that must be implemented.
    public void onMessage(String soapXml)
        throws java.rmi.RemoteException {
        try {
            System.out.println("Got a message " + soapXml);
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
    }

    // The other mandatory method
    public void invokeService(java.util.Map map)
        throws java.rmi.RemoteException {
        try {
            // What is the requested service?
            String service = (String)map.get("service");
            System.out.println("Invoking service: " + service);

            // Does the client want a backup?
            if( service.equals("backup") ) {
                // The parameter
                int days = ((Integer)map.get("days")).intValue();
                // and the call?
                performBackup(days);
            }
            else
                System.err.println("Invalid service name.");
        }
        catch( Exception e ) {
        }
    }
}
```



```

        e.printStackTrace();
    }
}

// This is where the backup logic would reside.
private void performBackup(int howFarBackInDays) {
    try {
        System.out.println("Backing up all files that have" +
            " a timestamp in the last " +
            howFarBackInDays + " days.");
        Thread.sleep(5000);
        System.out.println("Backup Complete.");
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
}
}

```

The Deployment Descriptor

I have included the deployment descriptor that I used to deploy the bean as shown below. Once again, this is a standard descriptor for deploying a stateless session bean and is in way tainted by JMS.

```

<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
  <description>
    This is an example for a Async Stateless session bean
  </description>
  <enterprise-beans>
    <session>
      <description>
        This is an example for a Async Stateless session bean
      </description>
      <ejb-name>backup</ejb-name>
      <home>BackupHome</home>
      <remote>Backup</remote>
      <ejb-class>BackupBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>backup</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>

```

An example Client

Now let's look at an example client program that uses this architecture. The complete client implementation is shown below.

```

// A test client.
// Notice that it has no knowledge of EJB
import javax.jms.*;

```

```

public class BackupClient {
    public static void main(String[] args) throws Exception {
        try {
            // JMS setup work.

            // Get the queue connection factory.
            // This is the only "Sun Java Message Queue"
            //specific code.
            QueueConnectionFactory connectionFactory =
                new com.sun.messaging.QueueConnectionFactory();

            QueueConnection connection =
                connectionFactory.createQueueConnection();
            QueueSession session =
                connection.createQueueSession(false,1);
            Queue queue = session.createQueue("BackupQueue");
            QueueSender sender = session.createSender(queue);
            connection.start();

            // Create a test message with a complex
            // SOAP packet
            TextMessage m = session.createTextMessage();
            m.setText("This is a complex SOAP packet.");
            sender.send(m);

            // Create a message to invoke a remote
            // service called "backup"
            MapMessage m2 = session.createMapMessage();
            m2.setString("service", "backup");
            m2.setInt("days", 25);
            sender.send(m2);

            // Create a message to invoke remote service
            // called "backup" (again)
            m2 = session.createMapMessage();
            m2.setString("service", "backup");
            m2.setInt("days", 15);
            sender.send(m2);

            // done.
            session.close();
            connection.close();
        }
        catch( Exception e ) {
            e.printStackTrace();
        }
    }
}

```

Note that the client has no EJB code in it at all. It looks like any other JMS client, simply sending out messages to JMS destinations. A client in this architecture can send two types of messages: a `TextMessage` and a `MapMessage`. This client sends out both types as examples. The text content of the `TextMessage` can be anything that the receiving end (i.e the EJB) understands. For example, here it is assumed to be a SOAP packet. The `MapMessage` format is more interesting. It always contains a well-known key "service" that is set to the name of the service desired, which in this case is "backup". Depending on the service, the message may also contain "parameter" keys. For example, the "backup" service requires one parameter "days". So the message contains a "days" key that is set equal to the parameter value (25 in the first message, 15 in the second). Notice that the client sends all three messages without blocking and then ends. The requests/services indicated by these messages are performed concurrently as you'll see for yourself in the next section.

Compiling and Running the pieces

Setting up the environment

Copy the following into a batch file called setenv.bat. Be sure to change the directory paths.

```
REM Path includes the bin directory for the Inprise AppServer
set PATH="E:\Program Files\Inprise\AppServer\bin";C:\Program
Files\jdk1.2.2\bin;C:\WINNT\system32;C:\WINNT;C:\THINKPAD

REM Setup the classpath for the ORB and EJB
REM This is required by the Delegator and the EJB
set CLASSPATH=E:\Program
Files\Inprise\AppServer\lib\vbjorb.jar;E:\Program
Files\Inprise\AppServer\lib\vbobj.jar;E:\Program
Files\Inprise\AppServer\lib\vbjdev;E:\Program
Files\Inprise\AppServer\lib\vbdev;

REM Setup the classpath for Java Message Queue
REM This is required by the client and the Delegator
set JMQ_HOME=E:\Program Files\JavaMessageQueue1.0
set
CLASSPATH=%CLASSPATH%;%JMQ_HOME%\lib\jms.jar;%JMQ_HOME%\lib\jmq.jar;%JMQ
_HOME%\lib\jmqadmin.jar
```

Compiling the pieces

Copy the following into a batch file called make_all.bat.

```
vbjc BackupHome.java
java2iioop -compile BackupHome
vbjc *.java
jar cMf backup_beans.jar META-INF *.class
vbj com.inprise.ejb.util.Verify backup_beans.jar
```

Now from a dos prompt in the directory that contains all the code (AsyncDelegator.java, BackupHome.java, Backup.java, BackupBean.java, and BackupClient.java), run the setenv batch file followed by make_all batch file as follows:

```
setenv
make_all
```

Start the OSAGENT

In the same dos box start the OSAGENT.

Start the Java Message Queue Router

From another dos box in the bin directory of the Java Message Queue installation, start the router as follows:

```
set JAVA_HOME=C:\Program Files\jdk1.2.2
set JMQ_HOME=E:\Program Files\JavaMessageQueue1.0
irouter
```

Start the AsyncDelegator

From a dos prompt in the same directory that you compiled the code in, start the delegator as follows:

```
setenv
vbj %FLAGS% AsyncDelegator delegator.properties
```

Here delegator.properties is a properties file in the same directory that contains the following.

```
# delegator.properties
BackupQueue=backup|BackupHome
```

Start an EJB container for the Backup bean

From a dos prompt in the same directory that you compiled the code in, start a container for the backup EJB as follows:

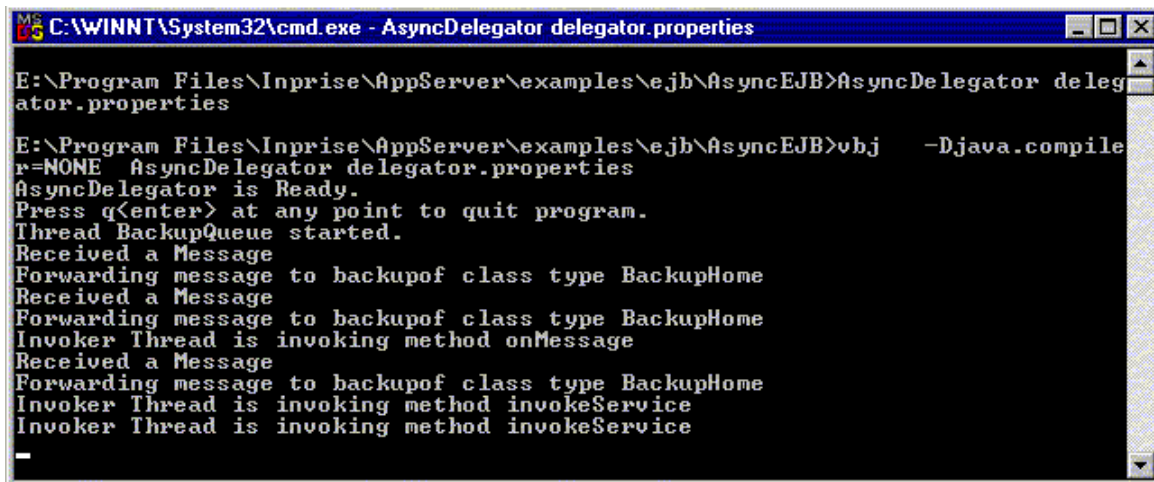
```
setenv
vbj com.inprise.ejb.Container ejbcontainer backup_beans.jar -jts -jns
```

Run the Backup client

Finally, from a dos prompt in the same directory that you compiled the code in, start the client program as follows:

```
setenv
vbj BackupClient
```

Figures 2 and 3 show the dos boxes for the AsyncDelegator and the EJB container at this point on my machine. You may have one, two, or three instances of the backup bean in the pool. I have two as seen in figure 3, which is the most common. Also, note how the two backups started by the client are occurring concurrently.



```
C:\WINNT\System32\cmd.exe - AsyncDelegator delegator.properties
E:\Program Files\Inprise\AppServer\examples\ejb\AsyncEJB>AsyncDelegator delegator.properties
E:\Program Files\Inprise\AppServer\examples\ejb\AsyncEJB>vbj -Djava.compile
r=NONE AsyncDelegator delegator.properties
AsyncDelegator is Ready.
Press q<enter> at any point to quit program.
Thread BackupQueue started.
Received a Message
Forwarding message to backupof class type BackupHome
Received a Message
Forwarding message to backupof class type BackupHome
Invoker Thread is invoking method onMessage
Received a Message
Forwarding message to backupof class type BackupHome
Invoker Thread is invoking method invokeService
Invoker Thread is invoking method invokeService
```

Figure 2: The AsyncDelegator after the backup client has been executed

```

MS-DOS Batch File
C:\WINNT\System32\cmd.exe - BackupServer
=====
Initializing ORB..... done
Initializing JMS..... done
Initializing JTS.... done
Initializing EJBs..... done
Container [ejbcontainer1] is ready
EJB Container Statistics
=====
Time                Sun May 07 10:12:25 EDT 2000
Memory (used)       1456 Kb (max 1456 Kb)
Memory (total)      2047 Kb (max 2047 Kb)
Memory (free)       28.0%
-----
Home                backup
Total in memory     0
Total in use        0
=====
Got a message This is a complex SOAP packet.
Invoking service: backup
Backing up all files that have a timestamp in the last 25 days.
Invoking service: backup
Backing up all files that have a timestamp in the last 15 days.
EJB Container Statistics
=====
Time                Sun May 07 10:12:31 EDT 2000
Memory (used)       1894 Kb (max 1894 Kb)
Memory (total)      2115 Kb (max 2115 Kb)
Memory (free)       10.0%
-----
Home                backup
READY              2
Total in memory     2
Total in use        2
=====
Backup Complete.
Backup Complete.

```

Figure 3: The EJB container after the backup client has been executed

Summary

In this chapter, I presented an architecture that allows the use of JMS with EJBs. This architecture is based on a delegation model. One interesting aspect of this architecture is that clients need not know that they are in fact dealing with EJBs. Similarly, EJBs are unaware that their clients are using JMS. JMS is accessible from EJBs, so this was not critical for the success of the architecture. This implies that I could have designed the `onMessage` and `invokeService` methods to accept JMS messages. However, JMS messages are not required to be serializable, so they cannot be passed by value as RMI (or RMI-IIOP) parameters. A few application servers come integrated with JMS providers and hence passing messages in these cases would not be a problem. However, my goal is always to avoid vendor specific capabilities. As an aside, it would be nice if JMS made the `Message` interface extend the `Serializable` interface, which would not require the reliance on vendor specific features/implementations to pass messages as parameters in remote calls. Another advantage of this architecture is that it can be used to access any resource/API that cannot normally be used directly with EJB, since the container or EJBs are never aware of that resource being used in this architecture.

In this chapter, I also showed you the wrong way to use JMS and EJB together to prove that there is definitely some thought involved in coming up with a workable and legal solution. The most intuitive and/or simplest solution is not always the right solution. In the next chapter, I will introduce you to the new Message-driven beans in EJB 2.0 that provide a standard way to integrate JMS with EJBs.

Chapter 12

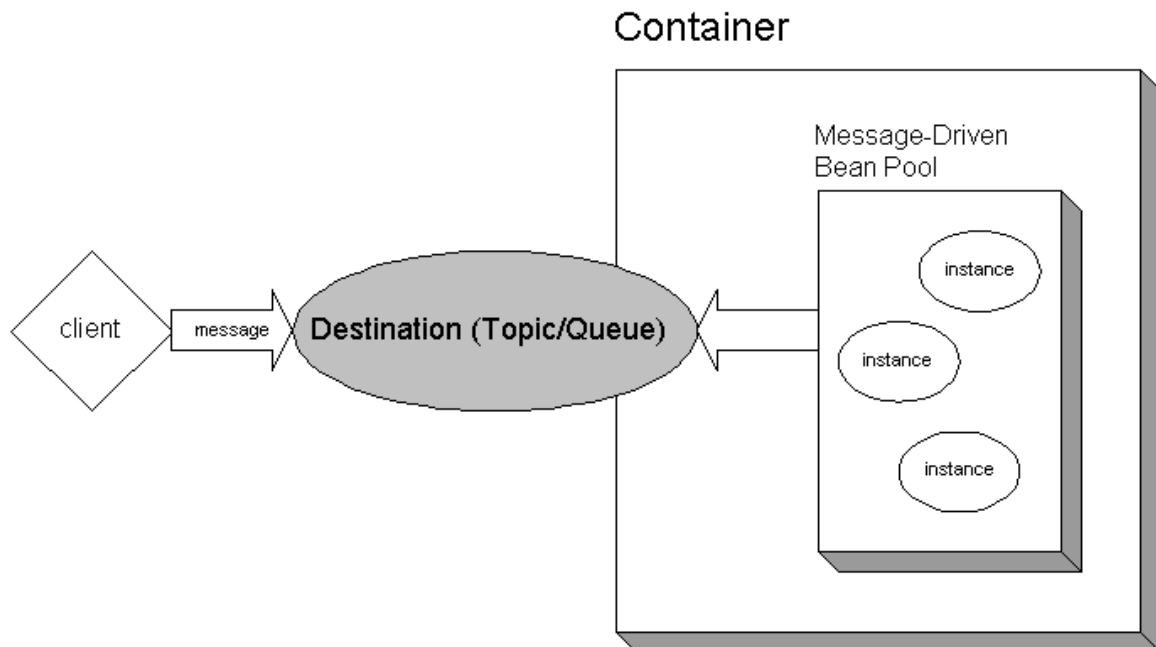
An Introduction to Message-Driven Beans in EJB 2.0³⁰

The Basics

Until the introduction of EJB 2.0, only synchronous method calls on enterprise beans (Entity and Session) were possible. JMS providers were still available as a resource to these beans and bean developers could obtain asynchronous behavior using JMS, but as we saw in the previous chapter this required significant work by the bean developer. Recognizing this, EJB 2.0 has introduced a new type of enterprise bean, the `MessageDrivenBean`, that is an asynchronous consumer of JMS messages. This chapter examines the message-driven bean in detail.

As mentioned above and in chapter 11, EJB 1.1 had support for two types of enterprise beans – entity beans and session beans. EJB 2.0 continues this support and adds a new message-driven bean. There are two fundamental differences between the message-driven bean and the other two types of enterprise beans. These differences are listed below.

1. Message-driven beans do not have a home or a remote interface. This is because a message-driven bean is not an RPC component and does not have business methods that are invoked by a client.
2. A client interacts with a message-driven bean in the same way it interacts with any other JMS application/server. The client simply sends a message to the appropriate destination (queue or topic) and [optionally] receives messages on the appropriate (or same) destination. In other words, from the perspective of a client, the existence of a Message-driven bean is completely hidden behind the JMS destination for which it is the message listener. This is shown in figure 1. As obvious from the figure, this is radically different than the way clients interact with entity and session beans.



³⁰ A thorough understanding of the EJB 1.1 specification is required for this chapter.

Figure 1: The client view of a message-driven bean

But there are some similarities as well. For example, as far as maintaining state goes, a message-driven bean is similar to a stateless session bean. It does not maintain any state for any specific client, not even conversational state. And just as in the case of a stateless session bean, the instance variables of the message-driven bean instance can also contain state across the handling of client messages. Examples of such state include an open database connection or an object reference to an EJB object.

Creating a Message-driven Bean

All message-driven beans must implement the `MessageDrivenBean` interface. All message-driven beans must also implement the `javax.jms.MessageListener` interface. This is because a message-driven bean is an asynchronous consumer of JMS messages (refer back to chapters 2 and 3 for a refresher if needed). The container calls the `onMessage` method when a message has arrived for the bean to service. Typically, the `onMessage` method contains the business logic that handles the processing of the message.

EJB Note

Only message-driven beans can asynchronously receive messages i.e. implement the `MessageListener` interface. Session and entity beans are not permitted to be JMS message listeners. So if you use JMS from session or entity beans you must use the blocking `receive` method to listen for messages.

Luckily, and very conveniently, the `MessageDrivenBean` interface already extends the `MessageListener` interface, so you don't have to remember to implement the JMS specific `MessageListener` interface. The definition of the `MessageDrivenBean` interface is shown below.

```
package javax.ejb;
import javax.jms.Message;
import javax.jms.MessageListener;
public interface MessageDrivenBean extends MessageListener {
    public void onMessage(Message message);
    public void ejbCreate();
    ..public void ejbRemove();
    ..public void setMessageDrivenContext(MessageDrivenContext mdc);
}
```

An example of a simple, yet complete (and legal) message-driven bean is as follows

```
public class HelloWorldBean implements MessageDrivenBean {
    public void ejbCreate() { }
    public void ejbRemove() { }
    public void setMessageDrivenContext(MessageDrivenContext mdc)
    { }
    public void onMessage(Message msg) {
        System.out.println("Hello World.");
    }
}
```

The container calls the `setMessageDrivenContext` method during the creation of the bean with a message-driven context object that implements the `MessageDrivenContext` interface. The `MessageDrivenContext` interface has the following methods:

- The `setRollbackOnly` method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback. Only instances of a message-driven bean with container-managed transaction demarcation can use this method.
- The `getRollbackOnly` method allows the instance to test if the current transaction has been marked for rollback. Only instances of a message-driven bean with container-managed transaction demarcation can use this method.
- The `getUserTransaction` method returns the `javax.transaction.UserTransaction` interface that the instance can use to demarcate transactions, and to obtain transaction status. Only instances of a message-driven bean with bean-managed transaction demarcation can use this method.
- The `getCallerPrincipal` method is inherited from the `javax.ejb.EJBContext` interface. Message-driven bean instances must not call this method.
- The `isCallerInRole` method is inherited from the `EJBContext` interface. Message-driven bean instances must not call this method.
- The `getEJBHome` method is inherited from the `EJBContext` interface. Message-driven bean instances must not call this method.

I will discuss the remaining two methods, `ejbCreate` and `ejbRemove`, of the `MessageDrivenBean` interface when I discuss the lifecycle of a message-driven bean later in this chapter.

The Container Contract

The container is responsible for making sure that all message-driven beans specified in the deployment descriptor are started up when the container starts up. Most containers will create a pool of each message-driven bean, just as they would for stateless session beans. Since all message-driven bean instances are equal, the container may deliver a message to any of the available message-driven bean instances in the pool for that destination. For example, if a container maintains a pool of 50 message-driven beans for each destination, any of these 50 instances may be used to process a message delivered at that destination. The container manages the life cycle of each message-driven bean instance. I will discuss the lifecycle of a message-driven bean in the next section. The container also provides security, concurrency, transactions, and other container specific services to the beans.

The Deployment Descriptor

The deployment descriptor allows the bean developer to concentrate on the business aspects of the bean without worrying about deployment issues such as security. At run-time, the container reads the deployment descriptor and deploys the beans as indicated by the descriptor. Thus the deployment descriptor plays a major role in defining the contract between the container and the bean. Deployment descriptors have played a key role in EJB since the very beginning and have gone through several revisions, with each revision adding more power and flexibility. It is no surprise that message-driven beans are also deployed using deployment descriptors. In this section, I will examine those pieces of the deployment descriptor that deal with message-driven beans.

A message-driven bean is deployed using the `message-driven` element, for example

```
<ejb-jar>
  <enterprise-beans>
    <message-driven>
      .
      .
      .
```

The `message-driven` element consists of the following

- The message-driven bean's implementation class.

- The message-driven bean's transaction management type
- An optional description.
- An optional display name.
- An optional small icon file name.
- An optional large icon file name.
- An optional name assigned to the enterprise bean.
- An optional declaration of the message-driven bean's message selector.
- An optional declaration of the acknowledgment mode for the message-driven bean if bean-managed transaction demarcation is used.
- An optional declaration of the message-driven bean's intended destination type.
- An optional declaration of the bean's environment entries.
- An optional declaration of the bean's EJB references
- An optional declaration of the security identity to be used for the execution of the bean's methods
- An optional declaration of the bean's resource factory references.
- And an optional declaration of the bean's resource environment references.

The declaration for this element is shown below

```
<!ELEMENT message-driven (
    description?, display-name?, small-icon?, large-icon?,
    ejb-name?, ejb-class, transaction-type,
    message-selector?, acknowledge-mode?,
    message-driven-destination?, env-entry*, ejb-ref*,
    security-identity?, resource-ref*, resource-env-ref*
)>
```

Of these, only the following are message-driven bean specific

- The optional declaration of the message-driven bean's intended destination type.
- The optional declaration of the message-driven bean's message selector.
- The optional declaration of the acknowledgment mode for the message-driven bean if bean-managed transaction demarcation is used.

And the following, although not message-driven bean specific, have special meaning/implications when with message-driven beans.

- The message-driven bean's transaction management type
- The optional declaration of the bean's resource environment references.

Now, let's look at each of the above five children elements of the `message-driven` element in detail.

1. The optional declaration of the message-driven bean's intended destination type.

The bean provider may provide the deployer with information about the destination to which a message-driven bean should be assigned. This is done using the `message-driven-destination` element. The declaration of this element is shown below.

```
<!ELEMENT message-driven-destination (
    destination-type, subscription-durability?
)>
```

As seen from the declaration, this element has one mandatory and one optional child element. The mandatory child is the `destination-type` element, which is used to advise the deployer as to the actual destination type to which a message-driven bean should be assigned. The type is specified by the Java interface expected to be implemented by the destination. The `destination-type` element must be one of the two following:

```
<destination-type>javax.jms.Queue</destination-type>
<destination-type>javax.jms.Topic</destination-type>
```

If the destination is a JMS topic then the `subscription-type` element may be used to specify whether the subscriptions are durable. The `subscription-durability` element must be one of the two following:

```
<subscription-durability>durable</subscription-durability>
<subscription-durability>nondurable</subscription-durability>
```

By default, topic subscriptions are non-durable. Also, as mentioned previously, the `message-driven-destination` element is optional and is for informational purposes only. If it is not specified, no default value for this tag is assumed.

2. The optional declaration of the message-driven bean's message selector.
The Bean Provider may declare the JMS message selector to be used in determining which messages the Message-driven bean is to receive. This is done using the `message-selector` element. For example,

```
<message-selector>
    JMSType = 'car' AND color = 'blue' AND weight &gt;31 2500
</message-selector>
```

Obviously, if the `message-selector` element is not specified then no message selector is used and no message filtering occurs.

3. The message-driven bean's transaction management type
This is a mandatory child element of the `message-driven` element. If the enterprise bean is a Session or a Message-driven bean, the bean provider must use the `transaction-type` element to declare whether transaction demarcation is performed by the enterprise bean or by the container. The `transaction-type` element must be one of the two following:

```
<transaction-type>Bean</transaction-type>
<transaction-type>Container</transaction-type>
```

The `onMessage` method is invoked in the scope of a transaction determined by the transaction attribute specified in the deployment descriptor. If the bean is specified as using container-managed transaction demarcation, either the `Required` or the `NotSupported` transaction attribute must be used. When a message-driven bean using bean-managed transaction demarcation uses the `javax.transaction.UserTransaction` interface to demarcate transactions, the message receipt that causes the bean to be invoked is not part of the transaction. If the message receipt is to be part of the transaction, container-managed transaction with the `Required` transaction attribute must be used. Also note that a message-driven bean's `newInstance`, `setMessageDrivenContext`, `ejbCreate`, and `ejbRemove` methods are called with an unspecified transaction context.

4. The optional declaration of the acknowledgment mode
The bean provider may declare the JMS acknowledgment mode option that should be used for a message-driven bean with bean managed transaction demarcation. The `acknowledge-mode` element specifies whether `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE` message acknowledgment semantics (on the JMS session) should be used when a message is delivered to the `onMessage` method of a message-driven bean that uses bean managed transaction demarcation. The `acknowledge-mode` element must be one of the following.

³¹ No, this is not an error. `>` is an entity definition for `">"`. Since `">"` is a special character i.e. a tag delimiter in XML, it must be replaced by this entity definition if it used for any other purpose, such as in this example

```
<acknowledge-mode>auto-acknowledge</acknowledge-mode>
<acknowledge-mode>dups-ok-acknowledge</acknowledge-mode>
```

Note that if container managed transactions are used then the acknowledgement mode is always `AUTO_ACKNOWLEDGE`. This is also the default if this element is not specified. Also, message-driven beans **must not** use the JMS API for message acknowledgment.

5. The optional declaration of the bean's resource environment references. The `resource-env-ref` element contains a declaration of an enterprise bean's reference to an administered object associated with a resource in the enterprise bean's environment. It consists of an optional description, the resource environment reference name, and an indication of the resource environment reference type expected by the enterprise bean code. In the case of the message-driven bean this element may be used to indicate the exact destination to use (as opposed to just the destination type).

For example,

```
<resource-env-ref>
  <resource-env-ref-name>
    jms/StockQueue
  </resource-env-ref-name>
  <resource-env-ref-type>
    javax.jms.Queue
  </resource-env-ref-type>
</resource-env-ref>
```

In this case the name `jms/StockQueue` can be bound to the actual JMS queue that the container can associate with the message-driven bean instance(s).

The Lifecycle of a Message-Driven Bean

Just as entity and session beans have well-defined lifecycles, so does the message-driven bean. The message-driven bean's lifecycle has two states: the *does not exist* state and the *method-ready pool* state. Once again, note the similarity with the stateless session bean's lifecycle. The lifecycle is shown in figure 2.

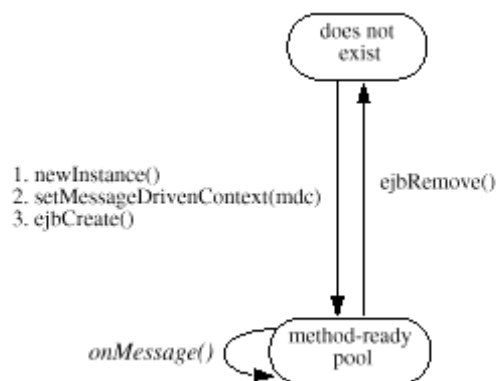


Figure 2: Lifecycle of a Message-Driven Bean

When the container starts up all message-driven beans are in the *does not exist* state. Typically, the container will create a number of bean instances and enter them into the *method-ready pool*

state. If at any time the number of message-driven bean instances are insufficient, more instances may be created. Let's examine the lifecycle in three stages

1. Transitioning from the *does not exist* state to the *method-ready pool* state

This involves four steps

- a. The container creates a new message-driven bean context object for the new message-driven bean that is about to be created.
- b. The container instantiates a bean instance by invoking the `Class.newInstance` method on the message-driven bean class. Thus all message-driven beans must have a default constructor. In fact, an enterprise bean must never define a constructor at all, but take care of all initialization in the `ejbCreate` method instead.
- c. The container then calls the `setMessageDrivenContext` method on the newly created bean instance passing in the context object created in (a).
- d. The container calls the `ejbCreate` method on the bean instance. A message-driven bean has only one `ejbCreate` method, similar to the stateless session bean and this method is called only once in the lifecycle of the message-driven bean.

These four steps are shown in figure 3.

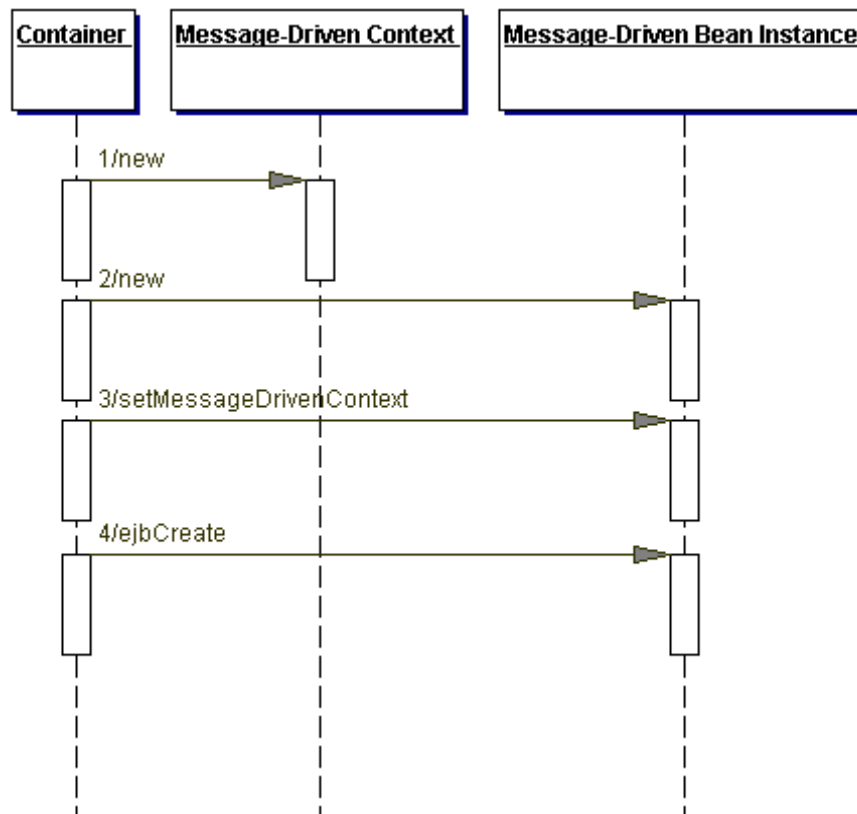


Figure 3: The creation of a message-driven bean

Since message-driven beans do not have any conversational state associated with them they are not subject to activation and passivation. Once again this is similar to stateless session beans. As a result message-driven beans can maintain open connections to resources for their entire lifecycle. This is shown in the code fragment below.

```
// A message-driven bean that maintains an
// open database connection through its life
```

```

public class HelloWorldBean implements MessageDrivenBean {
    private Connection conn = null;

    public void ejbCreate() {
        // Create and keep a reference
        //to a database connection
        conn = ?
    }

    public void ejbRemove() {
        // Clean up
        conn.close();
        conn = null;
    }

    public void setMessageDrivenContext(
        MessageDrivenContext mdc) { }

    public void onMessage(Message msg) {
        // Use the connection to store the message
        // contents into the database.
    }
}

```

2. Life in the *method-ready pool* state

When a message arrives at a destination, the container picks the next available message-driven bean for that destination from the pool of such beans to process the message by calling the `onMessage` method on the selected bean as shown in figure 4.

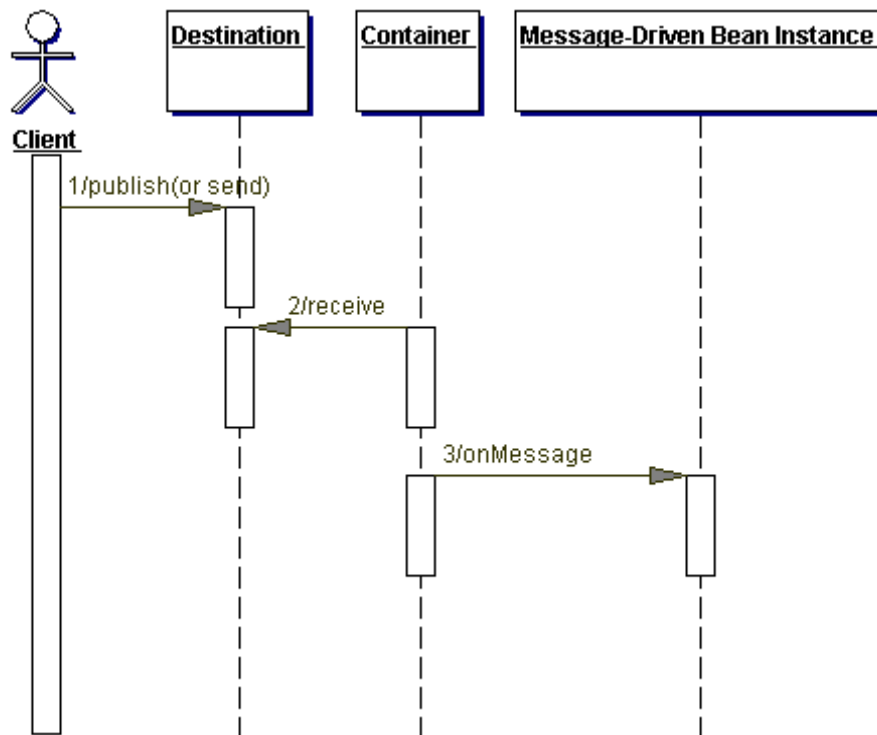


Figure 4: How a message flows from the client to the message-driven bean

Remember, JMS mandates that all calls to the `onMessage` method be serialized. I discussed this in detail in chapter 3. Therefore, when a bean is in use by the container it is unavailable to process any other messages. So a message-driven bean does not have to be coded to be reentrant or thread-safe.

3. Transitioning from the *method-ready pool* state to the *does not exist* state
When the container no longer needs a message-driven bean, it will invoke the `ejbRemove` method signaling the death of this bean. This is shown in figure 5.

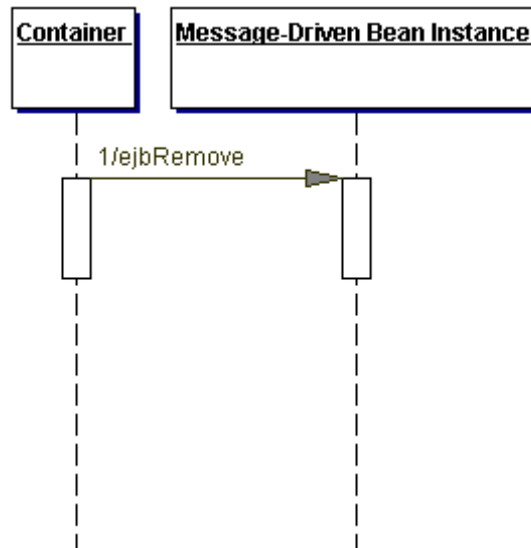


Figure 5: The end of a message-driven bean's life

If the bean maintains any open connections to resources, this is the time to close them and clean up. When this method is called, the message-driven context is still valid and may be used if necessary. According to the EJB specification, the bean provider cannot assume that the container will always invoke the `ejbRemove` method on a message-driven bean instance.

The following scenarios result in `ejbRemove` not being called on an instance:

- A crash of the EJB Container.
- A system exception thrown from the instance's method to the Container.

If the message-driven bean instance allocates resources in the `ejbCreate` method and/or in the `onMessage` method, and releases normally the resources in the `ejbRemove` method, these resources will not be automatically released in the above scenarios. The application using the message-driven bean should provide some clean up mechanism to periodically clean up the unreleased resources.

Summary

The introduction of the message-driven bean has taken the EJB development model into a new era of asynchronous bean processing and has solidified the position of JMS within the J2EE platform. Furthermore, the design of the message-driven bean has many parallels with the stateless session bean, which reduces the learning curve for existing EJB developers.

Appendix A

The JMS Exception Family

It has been argued that of all professionals, software professionals have the biggest egos. This is probably why most software professionals – sometimes including myself – have a hard time admitting that software written by us seldom works the way it's supposed to, the first time around. There are many reasons behind this, time pressures being just one of them. Whatever the reasons are, though, almost all will agree that catching and processing the right exceptions can go a long way in easing the pain associated with making the software accomplish its "real" task.

In this appendix, I will provide an overview of standard exceptions defined in the JMS.

Understanding the JMS Exceptions

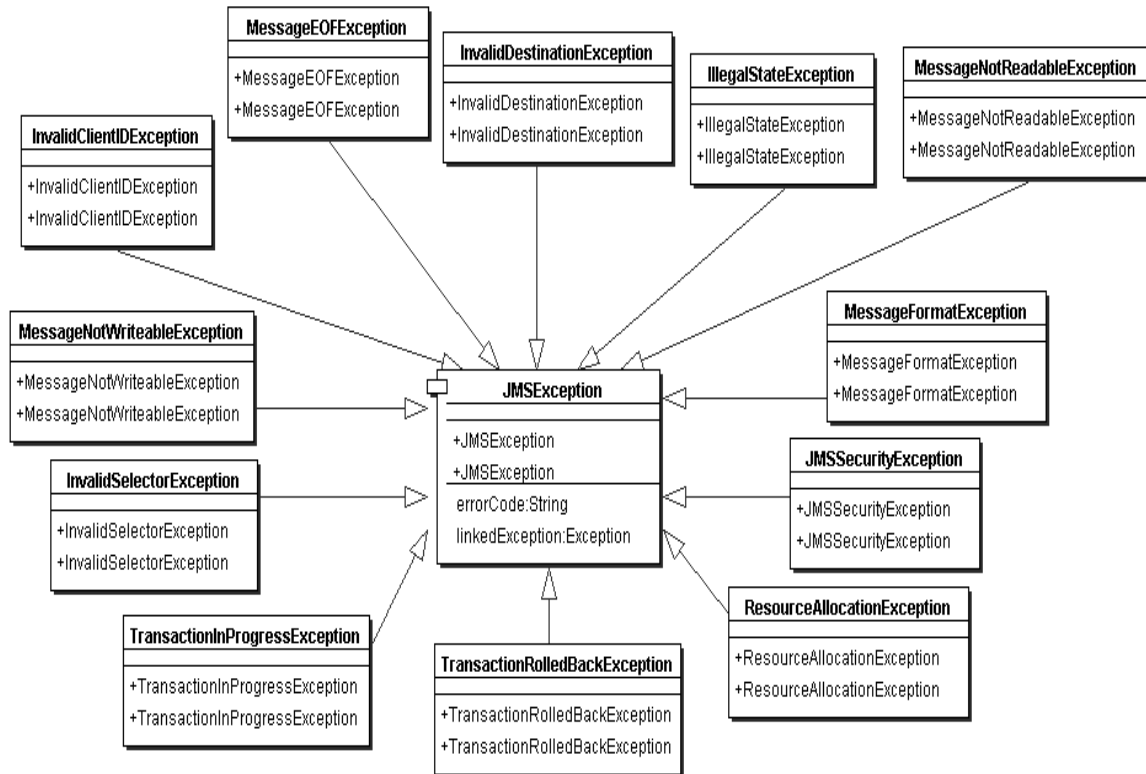


Figure 1: The JMS Exception classes

As shown in figure 1, JMS defines quite a few exceptions for reporting basic error conditions to the client of a JMS compliant messaging product.

While there are many cases where JMS mandates that a specific JMS exception must be thrown, there are also quite a few cases in which JMS only [strongly] suggests that providers use one of the standard exceptions where possible. JMS providers may also derive provider specific exceptions from these if needed. I will clearly point out those situations in which JMS mandates the use of a particular exception by using the words **"must be"** in the exception description.

JMSException

JMSException as the root interface for exceptions thrown by JMS methods. JMSException is a checked exception. This means that the caller of any method that throws this exception must either catch it, or declare it (or one of its superclasses) as being thrown in the `throws` part of the method declaration. The definition of JMSException is shown below:

```
package javax.jms;

public class JMSException extends Exception {
    private String errorCode;
    private Exception linkedException;

    public JMSException(String reason, String errorCode) {
        super(reason);
        this.errorCode = errorCode;
        linkedException = null;
    }

    public JMSException(String reason) {
        super(reason);
        this.errorCode = null;
        linkedException = null;
    }

    public String getErrorCode() {
        return this.errorCode;
    }

    public Exception getLinkedException() {
        return (linkedException);
    }

    public synchronized void setLinkedException(Exception ex) {
        linkedException = ex;
    }
}
```

JMSException provides the following information:

- A provider-specific string describing the error, which is the standard java exception message and is available via `getMessage()`.
- A provider-specific error code of type `String` and available through the method `getErrorCode` on JMSException.
- A reference to another exception. This is often a JMS exception that is the result of a lower level problem. If such an exception has been linked then it is available through the method `getLinkedException` on JMSException.

The first two pieces of information are specified via the appropriate constructor on the JMSException class. The linked exception is set via the `setLinkedException` method.

Now, let's take a look at each of the exceptions that are *derived* from JMSException. For each exception, I'll describe the condition(s) under which the exception may/must be thrown and follow it with definition of the exception. As you'll see the subclasses are all very similar in definition and do not add any new data to the base JMSException.

IllegalStateException

This exception is thrown when a method is invoked at an illegal or inappropriate time or if the provider is not in an appropriate state for the requested operation. For example, this exception

should be thrown if `Session.commit()` is called on a non-transacted session, or calling the `setClientID` method on a `Connection` object whose client identifier has already been set.

```
package javax.jms;
public class IllegalStateException extends JMSEException {
    public IllegalStateException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public IllegalStateException(String reason) {
        super(reason);
    }
}
```

JMSSecurityException

This exception **must be** thrown when a provider rejects a user name/password submitted by a client. It may also be thrown for any case where a security restriction prevents a method from completing.

```
package javax.jms;
public class JMSSecurityException extends JMSEException {
    public JMSSecurityException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public JMSSecurityException(String reason) {
        super(reason);
    }
}
```

InvalidClientIDException

This exception **must be** thrown when a client attempts to set a connection's client id to a value that is rejected by a provider.

```
package javax.jms;
public class InvalidClientIDException extends JMSEException {
    public InvalidClientIDException(String reason) {
        super(reason);
    }
    public InvalidClientIDException(String reason, String
        errorCode) {
        super(reason, errorCode);
    }
}
```

InvalidDestinationException

This exception **must be** thrown when a destination is either not understood by a provider or is no longer valid.

```
package javax.jms;
public class InvalidDestinationException extends JMSEException {
    public InvalidDestinationException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public InvalidDestinationException(String reason) {
        super(reason);
    }
}
```

InvalidSelectorException

This exception **must be** thrown when a JMS client attempts to give a provider a message selector with invalid syntax. I will discuss the message selector syntax in detail in the next chapter.

```

package javax.jms;
public class InvalidSelectorException extends JMSEException {
    public InvalidSelectorException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public InvalidSelectorException(String reason) {
        super(reason);
    }
}

```

MessageEOFException

This exception **must be** thrown when an unexpected end of stream has been reached when a `StreamMessage` or `BytesMessage` is being read. I'll discuss the different types of messages and how to read them in the next chapter as well.

```

package javax.jms;
public class MessageEOFException extends JMSEException {
    public MessageEOFException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public MessageEOFException(String reason) {
        super(reason);
    }
}

```

MessageFormatException

This exception **must be** thrown under the following circumstances:

- A JMS client attempts to use a data type not supported by a message.
- A JMS client attempts to read data in a message as the wrong type.
- A JMS client makes an equivalent type error with message property values. For example, trying to read a boolean value in the message as a long type.
- A JMS client gives a provider a type of message the provider cannot accept.

```

package javax.jms;
public class MessageFormatException extends JMSEException {
    public MessageFormatException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public MessageFormatException(String reason) {
        super(reason);
    }
}

```

MessageNotReadableException

This exception **must be** thrown when a JMS client attempts to read a write-only message.

```

package javax.jms;
public class MessageNotReadableException extends JMSEException {
    public MessageNotReadableException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public
    MessageNotReadableException(String reason) {
        super(reason);
    }
}

```

MessageNotWriteableException

This exception **must be** thrown when a JMS client attempts to write to a read-only message. For example, when a client receives a message, the message body is read only. If an attempt is made to change the contents of the body a `MessageNotWriteableException` exception must/will be thrown.

```
package javax.jms;
public class MessageNotWriteableException extends JMSEException {
    public MessageNotWriteableException(String reason) {
        super(reason);
    }
    public MessageNotWriteableException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
}
```

ResourceAllocationException

This exception is thrown when a provider is unable to allocate the resources required by a method. The JMS specification does not specify what types of resources this includes and hence it depends on the provider.

```
package javax.jms;
public class ResourceAllocationException extends JMSEException {
    public ResourceAllocationException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public ResourceAllocationException(String reason) {
        super(reason);
    }
}
```

TransactionInProgressException

This exception is thrown when an operation is invalid because a transaction is in progress. Attempting to call `Session.commit()` when a session is part of a distributed transaction is an example of such a situation.

```
package javax.jms;
public class TransactionInProgressException extends JMSEException {
    public TransactionInProgressException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public TransactionInProgressException(String reason) {
        super(reason);
    }
}
```

TransactionRolledBackException

This exception **must be** thrown when an attempt to *commit* the current transaction results in a rollback of the transaction.

```
package javax.jms;
public class TransactionRolledBackException extends JMSEException {
    public TransactionRolledBackException(String reason,
        String errorCode) {
        super(reason, errorCode);
    }
    public TransactionRolledBackException(String reason) {
        super(reason);
    }
}
```

Summary

JMS provides a rich exception hierarchy to aid clients in designing and debugging their message-based applications. Understanding this exception hierarchy can come in very handy and can also help understand the workings of JMS.

Appendix B

JMS Compliant Vendors³²

Commercial Implementations

Allaire Corporation (<http://www.allaire.com/products/jrun>)

JRun 3.0 provides a JMS 1.0 compliant Java-based messaging server built from the ground up to specification. To boost flexibility and performance, the JRun messaging server is built using EJBs and runs in the same process as the EJB and transaction server.

BEA Systems (<http://www.beasys.com/>)

WebLogic includes a full-featured messaging system, which can be configured by setting properties in the `weblogic.properties` file, from the WebLogic Console, or programmatically, using the JMS interfaces.

Fiorano Software (<http://www.fiorano.com/>)

Not only does Fiorano provide a JMS 1.0 compliant messaging server, but it also provides a comprehensive test suite for testing the conformance of a JMS implementation to the API standards made by Sun Microsystems. Version 2.0 of the JMS++ Suite, includes positive conformance tests for each line of the JMS 1.01 API. JMS++ is currently being used by Oracle Corporation, BEA/Web-Logic, Progress Software and others for JMS conformance testing. The JMS++ Test Suite is available free of charge to corporations for the purpose of evaluating various JMS implementations.

Gemstone (<http://www.gemstone.com/>)

GemStone/J supports vendors who comply with the JMS 1.0 specification. JMS is an optional feature in all editions of GemStone/J. All sales, service and support are contracted directly through the JMS vendor.

IBM (<http://www-4.ibm.com/software/ts/mqseries/api/mqjava.html>)

One of the ways in which MQSeries supports building message-based applications in Java is by providing a set of classes that implement the JMS 1.0 specification. A JMS application can use the classes to send MQSeries messages to either existing MQSeries or new JMS applications. Use of the MQSeries classes for Java Message Service offers benefits associated with using an open standard to write MQSeries applications, such as the protection of investment both in skills and application code. In addition the JMS classes provide some additional features not present in the MQSeries classes for Java. These extra features include:

- Explicit support for publish and subscribe
- Asynchronous Message Delivery
- Message Selectors
- Structured message classes

PCB Systems (<http://www.pcbsys.com/nirvana/njms.html>)

Nirvana is a 100% Java Message Oriented Middleware (MOM) package. PCB Systems also offers nJMS, which is an implementation of the JMS 1.0 specification on top of Nirvana.

Oracle (<http://www.oracle.com/>)

The Oracle application server provides support for JMS 1.0

³² In alphabetical order; not an exhaustive list

Progress Software (<http://www.progress.com/sonicmq/>)

SonicMQ is a fully JMS 1.0 compliant messaging server.

SAGA Software (<http://www.sagasoftware.com/>)

Sagavista is a JMS compliant enterprise application integration (EAI) product.

SoftWired (<http://www.softwired-inc.com/>)

There are three core products all of which are JMS 1.0 compliant. The most interesting product is iBus//Mobile, a JMS implementation for mobile devices.

SpiritSoft (<http://www.spirit-soft.com/>)

SpiritWave is a JMS 1.0 compliant implementation.

Sun Microsystems (<http://www.sun.com/workshop/jmq/index.html>)

Java Message Queue is a fully JMS 1.0 compliant messaging server. Java Message Queue has been certified by iPlanet for operation with its iPlanet Application Server 6.0 release. Java Message Queue software is comprised of two primary components: a client library and a router. The client side APIs are written entirely in the Java language, but have not been run through any of the tests necessary to certify them as 100% Pure Java. The router element of Java Message Queue is available only as a C language implementation, though plans are underway to develop a Java version in the future.

Venue Software (<http://www.venuesoftware.com/>)

A JMS 1.0 compliant messaging product available.

Open Source Implementations

ObjectCube (<http://www.objectcube.com/products.htm>)

ObjectEvents is a JMS (Java Messaging Service) compliant message oriented middle ware. ObjectEvents is entirely written in Java, but non-Java clients can access the middle ware since the protocol is open and based on TCP/IP and XML. ObjectEvents is distributed as open source software as part of the Enhydra J2EE application server. ObjectEvents Pro is the commercial version of the ObjectEvents product. In addition to JMS compliance, functions such as Load Balancing, Transactions, Fault Tolerance, Delivery Notification, Security and Message Repository will be added based on customer demand.

OpenJMS (<http://openjms.exolab.org/>)

OpenJMS is an open source implementation of JMS 1.0, sponsored by Intalio, Inc. and is part of the much larger open source Intalio platform. The initial development effort was undertaken by Jim Alateras and Jim Mourikis based in Melbourne, Australia.

ObjectWeb (<http://www.objectweb.org/joram/joramHomePage.htm>)

JORAM (Java Open Reliable Asynchronous Messaging) incorporates a JMS compliant messaging server.

Appendix C

Java Naming and Directory Service (JNDI)

JNDI is an API specified in Java that provides naming and directory functionality to applications written in Java. Naming and directory services play a vital role in intranets and the Internet by providing network-wide sharing of a variety of information about users, machines, networks, services, and applications. Examples of popular naming services include the RMI registry (used with RMI), the CORBA Naming Service specified by OMG, and the CORBA Trader Service (also specified by OMG). Recognizing the existence of so many industry-adopted naming and directory services, Javasoft decided not to create yet another naming and directory service, but an API that standardizes access to other naming and directory services from Java.

JNDI is designed especially for Java by using Java's object model. Using JNDI, Java applications can store and retrieve named Java objects of any type. Also as mentioned above, JNDI is defined independent of any specific naming or directory service implementation. It enables Java applications to access different, possibly multiple, naming and directory services using a common API. Different naming and directory service providers can be plugged in seamlessly behind this common API. This allows Java applications to take advantage of information in a variety of existing naming and directory services, such as LDAP, Novell Netware NDS, DNS, etc., and allows Java applications to coexist with legacy applications and systems.

For a tutorial of how to use JNDI refer to the following series of articles by Todd Sundsted in Java World

1. JNDI overview, Part 1: An introduction to naming services
<http://www.javaworld.com/javaworld/jw-01-2000/jw-01-howto.html>
2. JNDI overview, Part 2: An introduction to directory services
<http://www.javaworld.com/javaworld/jw-02-2000/jw-02-howto.html>
3. JNDI overview, Part 3: Advanced JNDI
<http://www.javaworld.com/javaworld/jw-03-2000/jw-03-howto.html>
4. JNDI overview, Part 4: the Doc-u-Matic, a JNDI application
<http://www.javaworld.com/javaworld/jw-03-2000/jw-0331-howto.html>