EJB 3RD EDITION - Richard Monson-Haefel

Authors' Note

In the winter of 1997 I was working on a distributed object project using Java RMI. Not surprisingly, the project failed miserably because Java RMI didn't address performance, scalability, fail-over, security, and transactions; qualities of service that are so vital in a production environment. Although that lesson was not new for me—I had seen the same thing happen with CORBA—the timing of the project was especially interesting. It was at that same time Enterprise JavaBeans was first introduced by Sun Microsystems – had Enterprise JavaBeans been available earlier, that same project probably would have succeeded.

At the time I was working on that ill-fated Java RMI project, I was also writing a column for JavaReport Online called the "The Cutting Edge". The column covered what were then, new Java technologies like Java Naming and Directory Interface (JNDI) and the JavaMail API. I was actually looking for a new topic for the 3rd edition of "The Cutting Edge", when I discovered the first public draft of Enterprise JavaBeans, version 0.8. I had first heard about this technology in 1996, but this was the first time anything public has been available. Having worked on CORBA, Java RMI and other distributed object technologies, I knew a good thing when I saw it and immediately began writing an article about this new "Enterprise JavaBeans". Although the article in question has long since been lost in the ether of the Internet, it was at that time the first article ever written on Enterprise JavaBeans.

That seems like eons ago. Since I published that article in March 1998, literally hundreds of articles have been written on Enterprise JavaBeans and several books have come and gone on the subject. Over the past three years this book has kept pace with three versions of the EJB specification and in its 3rd edition is considered by many, to my enormous satisfaction, to be the best book on Enterprise JavaBeans. As the newest version of the specification takes flight and a slew of new books on the subject daybew I can't help but remember the days when the words "Enterprise JavaBeans" drew blank looks from just about everyone. I'm glad those days are over.

What Is Enterprise JavaBeans?

When Java™ was first introduced in the summer of 1995, most of the IT industry focused on its graphical user interface characteristics and the competitive advantage it offered in terms of distribution and platform independence. Those were interesting times. The Applet was king, and only a few of us were attempting to use it on the server side. I reality we spent about half our time coding and the other half trying to convince management that Java was not a fad.

Today, the focus has broadened considerably: Java has been recognized as an excellent platform for creating enterprise solutions, specifically for developing distributed server-side applications. This shift has much to do with Java's emerging role as a universal language for producing implementation-independent abstractions for common enterprise technologies. The JDBC™ API is the first and most familiar example. JDBC provides a vendor-independent Java interface for accessing SQL relational databases. This abstraction has been so successful that it's difficult to find a relational database vendor that doesn't support JDBC. Java abstractions for enterprise technologies have expanded considerably to include JNDI (Java Naming and Directory Interface™) for abstracting directory services, JTA (Java Transaction API) for abstracting access to transaction managers, JMS™ (Java Messaging Service) for abstracting access to different message-oriented middleware products, and so on.

Enterprise JavaBeans™ was first introduced as a draft specification in late 1997 and has since established itself as one of the most important Java enterprise technologies provided by Sun Microsystems. Enterprise JavaBeans (EJB) provides an abstraction for component transaction monitors (CTMs). Component transaction monitors represent the convergence of two technologies: traditional transaction processing monitors, such as CICS, TUXEDO, and Encina, and distributed object services, such as CORBA (Common Object Request Broker Architecture), DCOM, and native Java RMI. Combining the best of both technologies, component transaction monitors provide a robust, component-based environment that simplifies distributed development while automatically managing the most complex aspects of enterprise computing, such as object brokering, transaction management, security, persistence, and concurrency.

Enterprise JavaBeans (EJB) defines a server-side component model that allows business objects to be developed and moved from one brand of EJB container to another. A component (an enterprise bean) presents a simple programming model that allows the developer to focus on its business purpose. An EJB server is responsible for making the component a distributed object and for managing services such as transactions, persistence, concurrency, and security. In addition to defining the bean's business logic, the developer defines the bean's runtime attributes in a way that is similar to choosing the display properties of visual widgets. The transactional, persistence, and security behaviors of a component can be defined by choosing from a list of properties. The end result is

that Enterprise JavaBeans makes developing distributed component systems that are managed in a robust transactional environment much easier. For developers and corporate IT shops that have struggled with the complexities of delivering mission-critical, high-performance distributed systems using CORBA, DCOM, or Java RMI, Enterprise JavaBeans provides a far simpler and more productive platform on which to base development efforts.

When Enterprise JavaBeans 1.0 was finalized in 1998, it quickly become a de facto industry standard. Many vendors announced their support even before the specification was finalized. Since that time Enterprise JavaBeans has been enhanced twice: The specification was first updated in 1999 to version 1.1, which was covered by the 2nd edition. The most recent revision to the specification, version 2.0, is covered by this, the 3rd edition of O'Reilly's EJB book. This 3rd edition also covers EJB 1.1, which is for the most part a subset of functionality offered by EJB 2.0.

Products that conform to the EJB standard have come from every sector of the IT industry, including the TP monitor, CORBA ORB, application server, relational database, object database, and web server industries. Some of these products are based on proprietary models that have been adapted to EJB; many more wouldn't even exist without EJB.

In short, Enterprise JavaBeans 2.0 and 1.1 provides a standard distributed component model that greatly simplifies the development process and allows beans that are developed and deployed on one vendor's EJB server to be easily deployed on a different vendor's EJB server. This book will provide you with the foundation you need to develop vendor-independent EJB solutions.

Who Should Read This Book?

This book explains and demonstrates the fundamentals of the Enterprise JavaBeans 2.0 and 1.1 architecture. Although EJB makes distributed computing much simpler, it is still a complex technology that requires a great deal of time to master. This book provides a straightforward, no-nonsense explanation of the underlying technology, Java classes and interfaces, component model, and runtime behavior of Enterprise JavaBeans. It includes material that is backward compatible with EJB 1.1 and provides special notes and chapters when there are significant differences between 1.1 and 2.0.

Although this book focuses on the fundamentals, it's no "dummies" book. Enterprise JavaBeans embodies an extremely complex and ambitious enterprise technology. While using EJB may be fairly simple, the amount of work required to truly understand and master EJB is significant. Before reading this book, you should be fluent with the Java language and have some practical experience developing business solutions. Experience with distributed object systems is not a must, but you will need some experience with JDBC (or at least an

understanding of the basics) to follow the examples in this book. If you are unfamiliar with the Java language, I recommend that you pick up a copy of *Learning Java™* by Patrick Neimeyer and Jonathan Knudsen, formerly *Exploring Java™*, (O'Reilly). If you are unfamiliar with JDBC, I recommend *Database Programming with JDBC™ and Java™, 2nd Edition* by George Reese (O'Reilly). If you need a stronger background in distributed computing, I recommend *Java™ Distributed Computing* by Jim Farley (O'Reilly).

Organization

Here's how the book is structured. The first three chapters are largely background material, placing Enterprise JavaBeans 2.0 and 1.1 in the context of related technologies, and explaining at the most abstract level how the EJB technology works and what makes up an enterprise bean. Chapters 4 through 13 go into detail about developing enterprise beans of various types. Chapters 14 and 15 could be considered "advanced topics," except that transactions (Chapter 14) are essential to everything that happens in enterprise computing, and design strategies (Chapter 15) help you deal with a number of real-world issues that influence bean design. Chapter 16 describes in detail the XML deployment descriptors used in EJB 2.0 and 1.1. Finally, Chapter 17 is an overview of the Java™ 2, Enterprise Edition (J2EE) includes Servlets, JSP and EJB.

Chapter 1, Introduction
: This chapter defines component transaction monitors and explains how they form the underlying technology of the Enterprise JavaBeans component model.

Chapter 2, Architectural Overview
: This chapter defines the architecture of the Enterprise JavaBeans component model and examines the difference between the three basic types of enterprise beans: entity beans, session beans, and message-driven beans.

Chapter 3, Resource Management and the Primary Services
: This chapter explains how the EJB-compliant server manages an enterprise bean at runtime.

Chapter 4, Developing Your First Enterprise Beans
: This chapter walks the reader through the development of some simple enterprise beans.

Chapter 5, The Client View
: This chapter explains in detail how enterprise beans are accessed and used by a remote client application.

Chapter 6, EJB 2.0 CMP: Basic Persistence
: This chapter provides an explanation of how to develop basic container-managed entity beans in EJB 2.0

Chapter 7, EJB 2.0 CMP: Entity Relationships

This chapter picks up where Chapter 6 left off, expanding your understanding of container-managed persistence to complex bean-to-bean relationships

Chapter 8, EJB 2.0 CMP: EJB QL

This chapter addresses the Enterprise JavaBeans Query Language (EJB QL), which is used to query EJBs and locate specific entity beans in EJB 2.0 container-managed persistence.

Chapter 9, EJB 1.1: Container-Managed Persistence

This chapter covers EJB 1.1 container-managed persistence, which is supported in EJB 2.0 for backward compatibility. Read this chapter only if you need to support legacy EJB applications.

Chapter 10, Bean-Managed Persistence

This chapter covers the development of bean-managed persistence beans including when to store, load, and remove data from the database.

Chapter 11, Entity-Container Contract

This chapter covers the general protocol between an entity bean and its container at runtime and applies to container-managed persistence in EJB 2.0 and 1.1, as well as bean-managed persistence.

Chapter 12, Session Beans

This chapter shows how to develop stateless and stateful session beans.

Chapter 13, Message-Driven Beans

This chapter shows how to develop message-driven beans in EJB 2.0.

Chapter 14, Transactions

This chapter provides an in-depth explanation of transactions and describes the transactional model defined by Enterprise JavaBeans.

Chapter 15, Design Strategies

This chapter provides some basic design strategies that can simplify your EJB development efforts and make your EJB system more efficient.

Chapter 16, XML Deployment Descriptors

This chapter provides an in-depth explanation of the XML deployment descriptors used in EJB 1.1 and 2.0.

Chapter 17, Java 2, Enterprise Edition

This chapter provides an overview of the Java 2, Enterprise Edition 1.3 and explains how 2.0 fits into this new platform.

Appendix A, The Enterprise JavaBeans API

This appendix provides a quick reference to the classes and interfaces defined in the EJB packages.

Appendix B, State and Sequence Diagrams

This appendix provides diagrams that clarify the life cycle of enterprise beans at runtime.

Appendix C, EJB Vendors

This appendix provides information about the vendors of EJB servers.

Software and Versions

This book covers Enterprise JavaBeans version 2.0 and version 1.1, including all optional features. It uses Java language features from the Java 1.2 platform and JDBC. Because the focus of this book is to develop vendor-independent Enterprise JavaBeans components and solutions, I have stayed away from proprietary extensions and vendor- dependent idioms. Any EJB-compliant server can be used with this book; you should be familiar with that server's specific installation, deployment, and runtime management procedures to work with the examples.

This book covers both EJB 2.0 and EJB 1.1. These two versions have a lot in common, but when they differ, chapters, or text with in a chapter, that specific to each version is clearly marked. Feel free to skip version-specific sections that do not concern you. Unless indicated, the source code in this book has been written for both EJB 2.0 and 1.1

Examples developed in this book are available from *ftp://ftp.oreilly.com/pub/examples/java/ejb*. The examples are organized by chapter.

Example Workbooks

Although EJB applications themselves are portable, the manor in which you install and run EJB products vary wildly from one vendor to the next. For this reason its nearly impossible to cover all the EJB products available, so we have chosen a radical but very effective way to address these differences: Workbooks.

To help you deploy the book examples in different EJB products, the author will publish several *free* "workbooks" which are used along with this book to run the examples on specific commercial and non-commercial EJB servers. The workbook for a specific product will address that products most advanced server. So for example, if the vendor supports EJB 2.0, then the examples in the workbook will address EJB 2.0 features. If, on the other hand, the vendor only supports EJB 1.1, then the examples in the workbook will be specific to EJB 1.1.

Although there are plans to publish workbooks for as many different EJB server, at least two workbooks will be made available immediately. These workbooks are *free* on-line in PDF format. The workbooks are all available at *http://www.oreilly.com/catalog/entjbeans3/* or *http://www.monson-haefel.com.*

Conventions

Italic is used for:

- Filenames and pathnames
- Hostnames, domain names, URLs, and email addresses
- New terms where they are defined

`Constant width` is used for:

- Code examples and fragments
- Class, variable, and method names, and Java keywords used within the text
- SQL commands, table names, and column names
- XML elements and tags

`Constant width bold` is used for emphasis in some code examples.

`Constant width italic` is used to indicate text that is replaceable. For example, in ***`BeanName`***`PK`, you would replace ***`BeanName`*** with a specific bean name.

An Enterprise JavaBean consists of many parts; it's not a single object, but a collection of objects and interfaces. To refer to an Enterprise JavaBean as a whole, we use the name of its business name in Roman type followed by the acronym, EJB (Enterprise JavaBean). For example, we will refer to the Customer EJB when we want to talk about the enterprise bean in general. If we put the name in a constant width font, we are referring explicitly to the bean's remote interface. So `CustomerRemote` is the remote interface that defines the business methods of the Customer EJB.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international or local)
(707) 829-0104 (fax)

You can also send us messages electronically. To be put on our mailing list or to request a catalog, send email to:

info@oreilly.com

To ask technical questions or comment on the book, send email to:

bookquestions@oreilly.com

We have a web site for the book, where we'll list errata and any plans for future editions. You can access this page at:

http://www.oreilly.com/catalog/entjbeans2/

For more information about this book and others, see the O'Reilly web site at:

http://www.oreilly.com/

The author maintains a web site for the discussion of EJB and related distributed computing technologies (*http://www.ejbnow.com*). EJBNow.com provides news about this book as well as code tips, articles, and an extensive list of links to EJB resources.

Acknowledgments

While there is only one name on the cover of this book, the credit for its development and delivery is shared by many individuals. Michael Loukides, my editor, was pivotal to the success of every edition of this book. Without his experience, craft, and guidance, this book would not have been possible.

Many expert technical reviewers helped ensure that the material was technically accurate and true to the spirit of Enterprise JavaBeans. Of special note are David Chappell of David Chappell & Associates, Jim Farley, author of *Java™ Distributed Computing* (O'Reilly, 1998), Greg Nyberg of ObjectPartners, Prasad Muppirala and Shannon Pieper of BORN Information Services, ……. They contributed greatly to the technical accuracy of this book and brought a combination of industry and real-world experience to bear, helping to make this one of the best books on Enterprise JavaBeans published today.

Special thanks also go to Sriram Srinivasan of BEA, Anne Thomas of Sun Microsystems, and Ian McCallion of IBM Hursley, Tim Rohaly of jGuru.com, James D. Frentress of ITM Corp., Andrzej Jan Taramina of Accredo Systems, Marc Loy, co-author of *Java™ Swing* (O'Reilly, 1998), Don Weiss of Step 1, Mike Slinn of The Dialog Corporation, and Kevin Dick of Kevin Dick & Associates. The contributions of these technical experts were critical to the technical and conceptual accuracy of earlier editions of this book. Others I would like to thank include Maggie Mezquita, Greg Hartzel, John Klug and Jon Jamsa of BORN Information who all suffered though the first draft of the first edition so long ago to provide valuable feedback.

Thanks also to Vlad Matena and Mark Hapner of Sun Microsystems, the primary architects of Enterprise JavaBeans; Linda DeMichiel, EJB 2.0 specification lead;

and Bonnie Kellett J2EE Program Manager – they were all willing to answer several of my most complex questions. Thanks to all the participants in the EJB-INTEREST mailing list hosted by Sun Microsystems for their interesting and sometimes controversial, but always informative, postings over the past four years.

Finally, the most sincere gratitude must be extended to my wife, Hollie, for supporting and assisting me through three years of painstaking research and writing which were required to produce three editions of this book. Without her unfailing support and love, this book would not have been completed.

1

Introduction

This book is about Enterprise JavaBeans 1.1 and 2.0 the second and third versions of the Enterprise JavaBeans specification. Just as the Java platform has revolutionized the way we think about software development, Enterprise JavaBeans has revolutionized the way we think about developing mission-critical enterprise software. It combines server-side components with distributed object technologies and asynchronous messaging to greatly simplify the task of application development. It automatically takes into account many of the requirements of business systems: security, resource pooling, persistence, concurrency, and transactional integrity.

This book shows you how to use Enterprise JavaBeans to develop scalable, portable business systems. But before we can start talking about EJB itself, we'll need a brief introduction to the technologies addressed by EJB, such as component models, distributed objects, component transaction monitors (CTMs), and asynchronous messaging. It's particularly important to have a basic understanding of component transaction monitors, the technology that lies beneath EJB. In Chapters 2 and 3, we'll start looking at EJB itself and see how enterprise beans are put together. The rest of this book is devoted to developing enterprise beans for an imaginary business and discussing advanced issues.

It is assumed that you're already familiar with Java; if you're not, *Exploring Java*™ by Patrick Niemeyer and Josh Peck is an excellent introduction. This book also assumes that

you're conversant in the JDBC API, or at least SQL. If you're not familiar with JDBC, see *Database Programming with JDBC™ and Java™, 2nd Edition*, by George Reese.

One of Java's most important features is platform independence. Since it was first released, Java has been marketed as "write once, run anywhere." While the hype has gotten a little heavy-handed at times, code written with Sun's Java programming language is remarkably platform independent. Enterprise JavaBeans isn't just platform independent—it's also implementation independent. If you've worked with JDBC, you know a little about what this means. Not only can the JDBC API run on a Windows machine or on a Unix machine, it can also access relational databases of many different vendors (DB2, Oracle, Sybase, SQLServer, etc.) by using different JDBC drivers. You don't have to code to a particular database implementation; just change JDBC drivers and you change databases. It's the same with Enterprise JavaBeans. Ideally, an Enterprise JavaBeans component, an enterprise bean, can run in any application server that implements the Enterprise JavaBeans (EJB) specification.[1] This means that you can develop and deploy your EJB business system in one server, such as Orion , and later move it to a different EJB server, such as Pramati, BEA's WebLogic, IBM's WebSphere, or open source projects like OpenEJB, JOnAS, and JBoss. Implementation independence means that your business components are not dependent on the brand of server, which means there are more options before you begin development, during development, and after deployment.

Setting the Stage

Before defining Enterprise JavaBeans more precisely, let's set the stage by discussing a number of important concepts: distributed objects, business objects, and component transaction monitors and asynchronous messaging.

Distributed Objects

Distributed computing allows a business system to be more accessible. Distributed systems allow parts of the system to be located on separate computers, possibly in many different locations, where they make the most sense. In other words, distributed computing allows business logic and data to be reached from remote locations. Customers, business partners, and other remote parties can use a business system at any time from almost anywhere. The most recent development in distributed computing is *distributed objects*. Dis-

1 Provided that the bean components and EJB servers comply with the specification and no proprietary functionality is used in development.

tributed object technologies such as Java RMI, CORBA, and Microsoft's .NET allow objects running on one machine to be used by client applications on different computers.

Distributed objects evolved from a legacy form of three-tier architecture, which is used in TP monitor systems such as IBM's CICS or BEA's TUXEDO. These systems separate the presentation, business logic, and database into three distinct tiers (or layers). In the past, these legacy systems were usually composed of a "green screen" or dumb terminals for the presentation tier (first tier), COBOL or PL/1 applications on the middle tier (second tier), and some sort of database, such as DB2, as the backend (third tier). The introduction of distributed objects in recent years has given rise to a new form of three-tier architecture. Distributed object technologies make it possible to replace the procedural COBOL and PL/1 applications on the middle tier with business objects. A three-tier distributed business object architecture might have a sophisticated graphical or web based interface, business objects on the middle tier, and a relational or some other database on the backend. More complex architectures are often used in which there are many tiers: different objects reside on different servers and interact to get the job done. Creating these *n*- tier architectures with Enterprise JavaBeans is relatively easy.

Server-Side Components

Object-oriented languages, such as Java, C++, and Smalltalk, are used to write software that is flexible, extensible, and reusable—the three axioms of object-oriented development. In business systems, object-oriented languages are used to improve development of GUIs, to simplify access to data, and to encapsulate the business logic. The encapsulation of business logic into *business objects* has become is a fairly recent focus in the information technology industry. Business is fluid, which means that a business's products, processes, and objectives evolve over time. If the software that models the business can be encapsulated into business objects, it becomes flexible, extensible, and reusable, and therefore evolves as the business evolves.

A server-side component model may define an architecture for developing *distributed business objects*. They combine the accessibility of distributed object systems with the fluidity of objectified business logic. Server-side component models are used on the middle-tier application servers, which manage the components at runtime and make them available to remote clients. They provide a baseline of functionality that makes it easy to develop distributed business objects and assemble them into business solutions.

Server-side components can also be used to model other aspects of a business system, such as presentation and routing. The Java Servlet for example is a server-side component that is used to generate HTML and XML data for presentation layer of a three-tier architecture. The EJB 2.0 message-driven beans, which are discussed later, are a

server-side components that is used for routing asynchronous messages from one source to another.

Server-side components, like other components, can be bought and sold as independent pieces of executable software. They conform to a standard component model and can be executed without direct modification in a server that supports that component model. Server-side component models often support attribute-based programming, which allows the runtime behavior of the component to be modified when it is deployed, without having to change the programming code in the component. Depending on the component model, the server administrator can declare a server-side component's transactional, security, and even persistence behavior by setting these attributes to specific values.

As an organization's services, products and operating procedures evolve, server-side components can be reassembled, modified, and extended so that the business system reflects those changes. Imagine a business system as a collection of server-side components that model concepts like customers, products, reservations, and warehouses. Each component is like a Lego block that can be combined with other components to build a business solution. Products can be stored in the warehouse or delivered to a customer; a customer can make a reservation or purchase a product. You can assemble components, take them apart, use them in different combinations, and change their definitions. A business system based on *server-side components* is fluid because it is objectified, and it is accessible because the components can be distributed.

Component Transaction Monitors

A new breed of software called *application servers* has recently evolved to manage the complexities associated with developing business systems in today's Internet world. An application server is often made up of some combination of several different technologies, including web servers, ORBs, MOM (message-oriented middleware), databases, and so forth. An application server can also focus on one technology, such as distributed objects. Application servers that are based on distributed objects vary in sophistication. The simplest facilitate connectivity between the client applications and the distributed objects and are called object request brokers (ORBs). ORBs allow client applications to locate and use distributed objects easily. ORBs, however, have frequently proven to be inadequate in high-volume transactional environments. ORBs provide a communication backbone for distributed objects, but they fail to provide the kind of robust infrastructure that is needed to handle larger user populations and mission-critical work. In addition, ORBs provide a fairly crude server-side component model that places the burden of handling transactions, concurrency, persistence, and other system-level considerations on the shoulders of the application developer. These services are not automatically

supported in an ORB. Application developers must explicitly access these services (if they are available) or, in some cases, develop them from scratch.

Early in 1999, Anne Manes[2] coined the term *component transaction monitor* (CTM) to describe the most sophisticated distributed object application servers. CTMs evolved as a hybrid of traditional TP monitors and ORB technologies. They implement robust server-side component models that make it easier for developers to create, use, and deploy business systems. CTMs provide an infrastructure that can automatically manage transactions, object distribution, concurrency, security, persistence, and resource management. They are capable of handling huge user populations and mission-critical work, but also provide value to smaller systems because they are easy to use. CTMs are the ultimate application server. Other terms for these kinds of technology include object transaction monitor (OTM), component transaction server, distributed component server, COMware, and so forth. This book uses the term "component transaction monitor" because it embraces the three key characteristics of this technology: the use of a component model, the focus on transactional management, and the resource and service management typically associated with monitors.

Enterprise JavaBeans: Defined

Sun Microsystems' definition of Enterprise JavaBeans is:

> The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.[3]

Wow! Now that's a mouthful and not atypical of how Sun defines many of its Java technologies—have you ever read the definition of the Java language itself? It's about twice as long. This book offers a shorter definition:

---

[2] At the time that Ms. Manes coined the term she worked for the Patricia Seybold Group under her maiden name, Anne Thomas. Ms. Manes is now the Directory of Business Strategy for Sun Microsystems, Sun Software division.

3 Sun Microsystems' *Enterprise JavaBeans™ Specification, v2.0*, Copyright 2001 by Sun Microsystems, Inc.

Enterprise JavaBeans is a standard server-side component model for component transaction monitors.

We have already set the stage for this definition by briefly defining the terms distributed objects, server-side components, and component transaction monitors. To provide you with a complete and solid foundation for learning about Enterprise JavaBeans, this chapter will now expand on these definitions.

If you already have a clear understanding of distributed objects, transaction monitors, CTMs, and asynchronous messaging feel free to skip the rest of this chapter an move on to chapter 2.

Distributed Object Architectures

EJB is a component model for component transaction monitors, which are based on distributed object technologies. Therefore, to understand EJB you need to understand how distributed objects work. Distributed object systems are the foundation for modern three-tier architectures. In a three-tier architecture, as shown in Figure 1-1, the presentation logic resides on the client (first tier), the business logic on the middle tier (second tier), and other resources, such as the database, reside on the backend (third tier).

[FIGURE]

Figure 1-1: Three-tier architecture

All distributed object protocols are built on the same basic architecture, which is designed to make an object on one computer look like it's residing on a different computer. Distributed object architectures are based on a network communication layer that is really very simple. Essentially, there are three parts to this architecture: the business object, the skeleton, and the stub.

The *business object* is the business object that resides on the middle tier. It's an instance of an object that models the state and business logic of some real-world concept, like person, order, account. Every business object class has matching stub and skeleton classes built specifically for that type of business object. So, for example, a distributed business object called `Person` would have matching `Person_Stub` and `Person_Skeleton` classes. As shown in Figure 1-3, the business object and skeleton reside on the middle tier, and the stub resides on the client.

The *stub* and the *skeleton* are responsible for making the business object, which lives on the middle tier, look as if it is running locally on the client machine. This is accomplished

through some kind of *remote method invocation* (RMI) protocol. An RMI protocol is used to communicate method invocations over a network. CORBA, Java RMI, and Microsoft .NET all use their own RMI protocol.[4] Every instance of the business object on the middle tier is wrapped by an instance of its matching skeleton class. The skeleton is set up on a port and IP address and listens for requests from the stub, which resides on the client machine and is connected via the network to the skeleton. The stub acts as the business object's surrogate on the client and is responsible for communicating requests from the client to the business object through the skeleton. Figure 1-3 illustrates the process of communicating a method invocation from the client to the server object and back. The stub and the skeleton hide the communication specifics of the RMI protocol from the client and the implementation class, respectively.

[FIGURE]

Figure 1-2: RMI loop

The business object implements a public interface that declares its business methods. The stub implements the same interface as the business object, but the stub's methods do not contain business logic. Instead, the business methods on the stub implement whatever networking operations are required to forward the request to the business object and receive the results. When a client invokes a business method on the stub, the request is communicated over the network by streaming the name of the method invoked, and the values passed in as parameters, to the skeleton. When the skeleton receives the incoming stream, it parses the stream to discover which method is requested, and then invokes the corresponding business method on the business object. Any value that is returned from the method invoked on the business object is streamed back to the stub by the skeleton. The stub then returns the value to the client application as if it had processed the business logic locally.

Rolling Your Own Distributed Object

The best way to illustrate how distributed objects work is to show how you can implement a distributed object yourself, with your own distributed object protocol. This will give you some appreciation for what a true distributed object protocol like CORBA does. Actual distributed object systems such as DCOM, CORBA, and Java RMI are, however, much more complex and robust than the simple example we will develop here. The distributed object system we develop in this chapter is only illustrative; it is not a real technology,

4 The acronym RMI isn't specific to Java RMI. This section uses the term RMI to describe distributed object protocols in general. Java RMI is the Java language version of a distributed object protocol.

nor is it part of Enterprise JavaBeans. The purpose is to provide you with some understanding of how a more sophisticated distributed object system works.

Here's a very simple distributed business object called `PersonServer` that implements the `Person` interface. The `Person` interface captures the concept of a person business object. It has two business methods: `getAge()` and `getName()`. In a real application, we would probably define many more behaviors for the `Person` business object, but two methods are enough for this example:

```
public interface Person {
    public int getAge() throws Throwable;
    public String getName() throws Throwable;
}
```

The implementation of this interface, `PersonServer`, doesn't contain anything at all surprising. It defines the business logic and state for a `Person`:

```
public class PersonServer implements Person {
    int age;
    String name;

    public PersonServer(String name, int age){
        this.age = age;
        this.name = name;
    }
    public int getAge(){
        return age;
    }
    public String getName(){
        return name;
    }
}
```

Now we need some way to make the `PersonServer` available to a remote client. That's the job of the `Person_Skeleton` and `Person_Stub`. The `Person` interface describes the concept of a person independent of implementation. Both the `PersonServer` and the `Person_Stub` implement the `Person` interface because they are both expected to support the concept of a person. The `PersonServer` implements the interface to provide the actual business logic and state; the `Person_Stub` implements the interface so that it can look like a `Person` business object on the client and relay requests back to the skeleton, which in turn sends them to the object itself. Here's what the stub looks like:

```
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
```

```
import java.net.Socket;

public class Person_Stub implements Person {
    Socket socket;

    public Person_Stub() throws Throwable {
       /* Create a network connection to the skeleton.
          Use "localhost" or the IP Address of the skeleton
          if it's on a different machine. */
       socket = new Socket("localhost",9000);
    }
    public int getAge() throws Throwable {
       // When this method is invoked, stream the method name to the
       // skeleton.
       ObjectOutputStream outStream =
           new ObjectOutputStream(socket.getOutputStream());
       outStream.writeObject("age");
       outStream.flush();
       ObjectInputStream inStream =
           new ObjectInputStream(socket.getInputStream());
       return inStream.readInt();
    }
    public String getName() throws Throwable {
        // When this method is invoked, stream the method name to the
        // skeleton.
        ObjectOutputStream outStream =
            new ObjectOutputStream(socket.getOutputStream());
        outStream.writeObject("name");
        outStream.flush();
        ObjectInputStream inStream =
            new ObjectInputStream(socket.getInputStream());
        return (String)inStream.readObject();
    }
}
```

When a method is invoked on the `Person_Stub`, a `String` token is created and streamed to the skeleton. The token identifies the method that was invoked on the stub. The skeleton parses the method-identifying token, invokes the corresponding method on the business object, and streams back the result. When the stub reads the reply from the skeleton, it parses the value and returns it to the client. From the client's perspective, the stub processed the request locally. Now let's look at the skeleton:

```
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;
```

```
import java.net.ServerSocket;

public class Person_Skeleton extends Thread {
    PersonServer myServer;

    public Person_Skeleton(PersonServer server){
        // Get a reference to the business object that this skeleton wraps.
        this.myServer = server;
    }
    public void run(){
      try {
        // Create a server socket on port 9000.
        ServerSocket serverSocket = new ServerSocket(9000);
        // Wait for and obtain a socket connection from stub.
        Socket socket = serverSocket.accept();
        while (socket != null){
            // Create an input stream to receive requests from stub.
            ObjectInputStream inStream =
                new ObjectInputStream(socket.getInputStream());
            // Read next method request from stub. Block until request is
            // sent.
            String method = (String)inStream.readObject();
            // Evaluate the type of method requested.
            if (method.equals("age")){
                // Invoke business method on server object.
                int age = myServer.getAge();
                // Create an output stream to send return values back to
                // stub.
                ObjectOutputStream outStream =
                    new ObjectOutputStream(socket.getOutputStream());
                // Send results back to stub.
                outStream.writeInt(age);
                outStream.flush();
            } else if(method.equals("name")){
                // Invoke business method on server object.
                String name = myServer.getName();
                // Create an output stream to send return values back to
                // the stub.
                ObjectOutputStream outStream =
                    new ObjectOutputStream(socket.getOutputStream());
                // Send results back to stub.
                outStream.writeObject(name);
                outStream.flush();
            }
```

```
        }
      } catch(Throwable t) {t.printStackTrace();System.exit(0); }
    }
    public static void main(String args [] ){
        // Obtain a unique instance Person.
        PersonServer person = new PersonServer("Richard", 36);
        Person_Skeleton skel = new Person_Skeleton(person);
        skel.start();
    }
}
```

The `Person_Skeleton` routes requests received from the stub to the business object, `PersonServer`. Essentially, the `Person_Skeleton` spends all its time waiting for the stub to stream it a request. Once a request is received, it is parsed and delegated to the corresponding method on the `PersonServer`. The return value from the business object is then streamed back to the stub, which returns it as if it was processed locally.

Now that we've created all the machinery, let's look at a simple client that makes use of the `Person`:

```
public class PersonClient {
    public static void main(String [] args){
        try {
            Person person = new Person_Stub();
            int age = person.getAge();
            String name = person.getName();
            System.out.println(name+" is "+age+" years old");
        } catch(Throwable t) {t.printStackTrace();}
    }
}
```

This client application shows how the stub is used on the client. Except for the instantiation of the `Person_Stub` at the beginning, the client is unaware that the `Person` business object is actually a network proxy to the real business object on the middle tier. In Figure 1-5, the RMI loop diagram is changed to represent the RMI process as applied to our code.

[FIGURE]

Figure 1-3: RMI Loop with Person business object

As you examine Figure 1-5, notice how the RMI loop was implemented by our distributed `Person` object. RMI is the basis of distributed object systems and is responsible for making distributed objects *location transparent*. Location transparency means that a server object's actual location—usually on the middle tier—is unknown and unimportant

to the client using it. In this example, the client could be located on the same machine or on a different machine very far away, but the client's interaction with the business object is the same. One of the biggest benefits of distributed object systems is location transparency. Although transparency is beneficial, you cannot treat distributed objects as local objects in your design because of the performance differences. This book will provide you with good distributed object design strategies that take advantage of transparency while maximizing the distributed system's performance.

When this book talks about the stub on the client, we will often refer to it as a *remote reference* to the business object. This allows us to talk more directly about the business object and its representation on the client.

Distributed object protocols such as CORBA, DCOM, and Java RMI provide a lot more infrastructure for distributed objects than the `Person` example. Most implementations of distributed object protocols provide utilities that automatically generate the appropriate stubs and skeletons for business objects. This eliminates custom development of these constructs and allows a lot more functionality to be included in the stub and skeleton.

Even with automatic generation of stubs and skeletons, the `Person` example hardly scratches the surface of a sophisticated distributed object protocol. Real world protocols like Java RMI and CORBA IIOP provide error and exception handling, parameter passing, and other services like the passing of transaction and security context. In addition, distributed object protocols support much more sophisticated mechanisms for connecting the stub to the skeleton; the direct stub-to-skeleton connection in the `Person` example is fairly primitive.

Real distributed object protocols, like CORBA, also provide an Object Request Broker (ORB), which allows clients to locate and communicate with distributed objects across the network. ORBs are the communication backbone, the switchboard, for distributed objects. In addition to handling communications, ORBs generally use a naming system for locating objects and many other features such as reference passing, distributed garbage collection, and resource management. However, ORBs are limited to facilitating communication between clients and distributed business objects. While they may support services like transaction management and security, use of these services is not automatic. With ORBs, most of the responsibility for creating system-level functionality or incorporating services falls on the shoulders of the application developer.

Component Models

The term “component model” has many different interpretations. Enterprise JavaBeans specifies a *server-side* component model. Using a set of classes and interfaces from the `javax.ejb` package, developers can create, assemble, and deploy components that conform to the EJB specification.

The original JavaBeans™, is also a component model, but it’s not a server-side component model like EJB. In fact, other than sharing the name “JavaBeans,” these two component models are completely unrelated. In the past, a lot of the literature had referred to EJB as an extension of the original JavaBeans, but this is a misrepresentation. Other than the shared name, and the fact that they are both Java component models, the two APIs serve very different purposes. EJB does not extend or use the original JavaBeans component model.

JavaBeans is intended to be used for *intra*process purposes, while EJB is designed to be used for *inter*process components. In other words, the original JavaBeans was not intended for distributed components. JavaBeans can be used to solve a variety of problems, but is primarily used to build clients by assembling visual (GUI) and nonvisual widgets. It’s an excellent component model, possibly the best component model for intraprocess development ever devised, but it’s not a server-side component model. EJB is designed to address issues involved with managing distributed business objects in a three-tier architecture.

Given that JavaBeans and Enterprise JavaBeans are completely different, why are they both called component models? In this context, a component model defines a set of contracts between the component developer and the system that hosts the component. The contracts express how a component should be developed and packaged. Once a component is defined, it becomes an independent piece of software that can be distributed and used in other applications. A component is developed for a specific purpose but not a specific application. In the original JavaBeans, a component might be a push button or spreadsheet that can be used in any GUI application according to the rules specified in the original JavaBeans component model. In EJB, a component might be a customer business object that can be deployed in any EJB server and used to develop any business application that needs a customer business object. Other types of Java component models include Servlets, JSPs, and Applets.

Component Transaction Monitors

The CTM industry grew out of both the ORB and the transaction processing monitor (TP monitor) industries. The CTM is really a hybrid of these two technologies that provides a powerful, robust distributed object platform. To better understand what a CTM is, we will examine the strengths and weakness of TP monitors and ORBs.

TP Monitors

Transaction processing monitors have been evolving for about 30 years (CICS was introduced in 1968) and have become powerful, high-speed server platforms for mission-critical applications. Some TP products like CICS and TUXEDO may be familiar to you. TP monitors are operating systems for business applications written in languages like COBOL. It may seem strange to call a TP monitor an "operating system," but because they control an application's entire environment, it's a fitting description. TP monitor systems automatically manage the entire environment that a business application runs in, including transactions, resource management, and fault tolerance. The business applications that run in TP monitors are written in procedural programming languages (e.g. COBOL and C) that are often accessed through network messaging or remote procedure calls (RPC). Messaging allows a client to send a message to a TP monitor requesting that some application be run with certain parameters. It's similar in concept to the Java event model. Messaging can be synchronous or asynchronous, meaning that the sender may or may not be required to wait for a response. RPC, which is the ancestor of RMI, is a distributed mechanism that allows clients to invoke procedures on applications in a TP monitor as if the procedure was executed locally. The primary difference between RPC and RMI is that RPC is used for *procedure*-based applications and RMI is used for distributed *object* systems. With RMI, methods can be invoked on a specific object identity, a specific business entity. In RPC, a client can call procedures on a specific type of application, but there is no concept of object identity. RMI is object oriented; RPC is procedural.

TP monitors have been around for a long time, so the technology behind them is as solid as a rock; that is why they are used in many mission-critical systems today. But TP monitors are not object oriented. Instead, they work with procedural code that can perform complex tasks but has no sense of identity. Accessing a TP monitor through RPC is like executing static methods; there's no such thing as a unique object. In addition, because TP monitors are based on procedural applications, and not objects, the business logic in a TP monitor is not as flexible, extensible, or reusable as business objects in a distributed object system.

Object Request Brokers

Distributed object systems allow unique objects that have state and identity to be accessed across a network. Distributed object technologies like CORBA and Java RMI grew out of RPC with one significant difference: when you invoke a distributed object method, it's on an object instance, not an application procedure. Distributed objects are usually deployed on some kind of ORB, which is responsible for helping client applications find distributed objects easily.

ORBs, however, do not define an "operating system" for distributed objects. They are simply communications backbones that are used to access and interact with unique remote objects. When you develop a distributed object application using an ORB, all the responsibility for concurrency, transactions, resource management, and fault tolerance falls on your shoulders. These services may be supported by an ORB, but the application developer is responsible for incorporating them into the business objects. In an ORB, there is no concept of an "operating system," where system-level functionality is handled automatically. The lack of implicit system-level infrastructure places an enormous burden on the application developer. Developing the infrastructure required to handle concurrency, transactions, security, persistence, and everything else needed to support large user populations is a Herculean task that few corporate development teams are equipped to accomplish.

CTMs: The Hybrid of ORBs and TP Monitors

As the advantages of distributed objects became apparent, the number of systems deployed using ORBs increased very quickly. ORBs support distributed objects by employing a somewhat crude server-side component model that allows distributed objects to be connected to a communication backbone, but don't implicitly support transactions, security, persistence, and resource management. These services must be explicitly accessed through APIs by the distributed object, resulting in more complexity and, frequently, more development problems. In addition, resource management strategies such as instance swapping, resource pooling, and activation may not be supported at all. These types of strategies make it possible for a distributed object system to scale, improving performance and throughput and reducing latency. Without automatic support for resource management, application developers must implement homegrown resource management solutions, which requires a very sophisticated understanding of distributed object systems. ORBs fail to address the complexities of managing a component in a high-volume, mission-critical environment, an area where TP monitors have always excelled.

With three decades of TP monitor experience, it wasn't long before companies like IBM and BEA began developing a hybrid of ORBs and TP monitor systems, which we refer to as component transaction monitors. These types of application servers combine the fluidity and accessibility of distributed object systems based on ORBs with the robust "operating system" of a TP monitor. CTMs provide a comprehensive environment for server- side components by managing concurrency, transactions, object distribution, load balancing, security, and resource management automatically. While application developers still need to be aware of these facilities, they don't have to explicitly implement them when using a CTM.

The basic features of a CTM are distributed objects, an infrastructure that includes transaction management and other services, and a server-side component model. CTMs support these features in varying degrees; choosing the most robust and feature-rich CTM is not always as critical as choosing one that best meets your needs. Very large and robust CTMs can be enormously expensive and may be overkill for smaller projects. CTMs have come out of several different industries, including the relational database industry, the application server industry, the web server industry, the CORBA ORB industry, and the TP monitor industry. Each vendor offers products that reflect their particular area of expertise. However, when you're getting started, choosing a CTM that supports the Enterprise JavaBeans component model may be much more important than any particular feature set. Because Enterprise JavaBeans is implementation independent, choosing an EJB CTM provides the business system with the flexibility to scale to larger CTMs as needed. We will discuss the importance of EJB as a standard component model for CTMs later in this chapter.

Analogies to Relational Databases

This chapter spent a lot of time talking about CTMs because they are essential to the definition of EJB. The discussion of CTMs is not over, but to make things as clear as possible before proceeding, we will use relational databases as an analogy for CTMs.

Relational databases provide a simple development environment for application developers, in combination with a robust infrastructure for data. As an application developer using a relational database, you might design the table layouts, decide which columns are primary keys, and define indexes and stored procedures, but you don't develop the indexing algorithm, the SQL parser, or the cursor management system. These types of system-level functionality are left to the database vendor; you simply choose the product that best fits your needs. Application developers are concerned with how business data is organized, not how the database engine works. It would be waste of resources for an application developer to write a relational database from scratch when vendors like Microsoft, Oracle, and others already provide them.

Distributed business objects, if they are to be effective, require the same system-level management from CTMs as business data requires from relational databases. System-level functionality like concurrency, transaction management, and resource management is necessary if the business system is going to be used for large user populations or mission-critical work. It is unrealistic and wasteful to expect application developers to reinvent this system-level functionality when commercial solutions already exist.

CTMs are to business objects what relational databases are to data. CTMs handle all the system-level functionality, allowing the application developer to focus on the business problems. With a CTM, application developers can focus on the design and development of the business objects without having to waste thousands of hours developing the infrastructure that the business objects operate in.

EJB 2.0: Asynchronous Messaging

An asynchronous messaging system allows two or more applications to exchange information in the form of messages. A message, in this case, is a self-contained package of business data and network routing headers. The business data contained in a message can be anything—depending on the business scenario—and usually contains information about some business transaction. In enterprise messaging systems, messages inform an application of some event or occurrence in another system.

Messages are transmitted from one application to another on a network using message-oriented middleware (MOM). MOM products ensure that messages are properly distributed among applications. In addition, MOMs usually provide fault tolerance, load balancing, scalability, and transactional support for enterprises that need to reliably exchange large quantities of messages.

MOM vendors use different message formats and network protocols for exchanging messages, but the basic semantics are the same. An API is used to create a message, give it a payload (application data), assign it routing information, and then send the message. The same API is used to receive messages produced by other applications.

In all modern enterprise messaging systems, applications exchange messages through virtual channels called *destinations.* When sending a message, it's addressed to a destination, not a specific application. Any application that subscribes or registers an interest in that destination may receive that message. In this way, the applications that receive messages and those that send messages are decoupled. Senders and receivers are not bound to each other in any way and may send and receive messages as they see fit.

Java Message Service

Each MOM vendor implements its own networking protocols, routing, and administration facilities, but the basic semantics of the developer API provided by different MOMs are the same. It's this similarity in APIs that makes the Java Message Service possible.

The Java Message Service (JMS) is a vendor-agnostic Java API that can be used with many different MOM vendors. JMS is very similar to JDBC in that application developer reuses the same API to access many different systems. If a vendor provides a compliant service provider for JMS, then the JMS API can be used to send and receive messages to that vendor. For example, you can use the same JMS API to send messages using Progress' SonicMQ as you do IBM's MQSeries.

Message-Driven Beans

All JMS vendors provide application developers with the same API for sending and receiving messages, and sometimes they provide a component model for developing routers that can receive and send messages. These component models, however, are proprietary and not portable across MOM vendors.

Enterprise JavaBeans 2.0 introduces a new kind of component, called a message-driven bean, which is a kind of standard JMS bean. It can receive and send asynchronous JMS messages, because it's co-located with other kinds of RMI beans (entity and session beans) it can also interact with RMI components.

Message-driven beans in EJB 2.0 act as an integration point for a EJB application, allowing other applications to asynchronous messages which can be captured and processed by an EJB application. This is an extremely important feature that will allow EJB applications to better integrate with legacy and other proprietary systems.

Message-driven beans are also transactional and required all the infrastructure associated with other RMI based transactional server-side components. Like other RMI based components, message-driven beans are considered business objects, which full fill an important role of routing and interpreting requests and coordinating the application of those requests against other RMI based components, namely enterprise beans. Message-driven beans are a good fit for the component transaction manager landscape and are an excellent addition to the Enterprise JavaBeans platform.

CTMs and Server-Side Component Models

CTMs require that business objects adhere to the server-side component model implemented by the vendor. A good component model is critical to the success of a development project because it defines how easily an application developer can write business objects for the CTM. The component model is a contract that defines the responsibilities of the CTM and the business objects. With a good component model, a developer knows what to expect from the CTM and the CTM understands how to manage the business object. Server-side component models are great at describing the responsibilities of the application developer and CTM vendor.

Server-side component models are based on a specification. As long as the component adheres to the specification, it can be used by the CTM. The relationship between the server-side component and the CTM is like the relationship between a CD-ROM and a CD player. As long as the component (CD-ROM) adheres to the player's specifications, you can play it.

A CTM's relationship with its component model is also similar to the relationship the railway system has with trains. The railway system manages the train's environment, providing alternate routes for load balancing, multiple tracks for concurrency, and a traffic control system for managing resources. The railway provides the infrastructure that trains run on. Similarly, a CTM provides server-side components with the entire infrastructure needed to support concurrency, transactions, load balancing, etc.

Trains on the railway are like server-side components: they all perform different tasks but they do so using the same basic design. The train, like a server-side component, focuses on performing a task, such as moving cars, not managing the environment. For the engineer, the person driving the train, the interface for controlling the train is fairly simple: a brake and throttle. For the application developer, the interface to the server-side component is similarly limited.

Different CTMs may implement different component models, just as different railways have different kinds of trains. The differences between the component models vary, like railway systems having different track widths and different controls, but the fundamental operations of CTMs are the same. They all ensure that business objects are managed so that they can support large populations of users in mission-critical situations. This means that resources, concurrency, transactions, security, persistence, load balancing, and distribution of objects can be handled automatically, limiting the application developer to a simple interface. This allows the application developer to focus on the business logic instead of the enterprise infrastructure.

Microsoft's .NET Framework

Microsoft was the first vendor to ship a CTM. Originally called the Microsoft Transaction Server (MTS), it was later renamed COM+. Microsoft's COM+ is based on the Component Object Model (COM), originally designed for use on the desktop but eventually pressed into service as a server-side component model. For distributed access, COM+ clients use DCOM (Distributed Component Object Model).

When MTS was introduced in 1996, it was exciting because it provided a very comprehensive environment for business objects. With MTS, application developers could write COM components without worrying about system-level concerns. Once a business object was designed to conform to the COM model, MTS (and now COM+) would take care of everything else, including transaction management, concurrency, resource management—everything!

Recently, COM+ has become part of Microsoft's new .NET Framework. The core functionality provided by COM+ services remains essentially the same in .NET, but the way it's appears to a developer changes significantly. Rather than writing components as COM objects, applications written for the .NET Framework are built as *managed objects*. All managed objects, and in fact all code written for the .NET Framework, depends on a Common Language Runtime (CLR). For Java-oriented developers, the CLR is much like a Java VM, and a managed object is very analogous to an instance of a Java class, i.e., to a Java object.

Although .NET Framework provides many interesting features, as an open standard, it falls short. The COM+ services in the .NET Framework are Microsoft's proprietary CTM, which means that using this technology binds you to the Microsoft platform. This may not be so bad, because .NET promises to work well, and the Microsoft platform is pervasive. In addition, the .NET Framework's support for SOAP (Simple Object Access Protocol) will enable business objects in the .NET world to communicate with objects on any other platform written in any language. This can potentially make business objects in .NET universally accessible, a feature that is not easily dismissed.

If, however, your company is expected to deploy server-side components on a non-Microsoft platform, .NET is not a viable solution. In addition, the COM+ services in the .NET Framework are focused on stateless components; there's no built-in support for persistent transactional objects. Although stateless components can offer higher performance, business systems need the kind of flexibility offered by CTMs that include stateful and persistent components.

EJB and CORBA CTMs

Until the fall of 1997, non-Microsoft CTMs were pretty much nonexistent. Promising products from IBM, BEA, and Hitachi were on the drawing board, while MTS was already on the market. Although the non-MTS designs were only designs, they all had one thing in common: they all used CORBA as a distributed object service.

Most non-Microsoft CTMs were focused on, what was at the time, the more open standard of CORBA so that they could be deployed on non-Microsoft platforms and support non-Microsoft clients. CORBA is both language and platform independent, so CORBA CTM vendors could provide their customers with more implementation options[5]. The problem with CORBA CTM designs was that they all had different server-side component models. In other words, if you developed a component for one vendor's CTM, you couldn't turn around and use that same component in another vendor's CTM. The component models were too different.

With Microsoft's MTS far in the lead by 1997 (it had already been around a year), CORBA-based CTM vendors needed a competitive advantage. One problem CTMs faced was a fragmented CORBA market where each vendor's product was different from the next. A fragmented market wouldn't benefit anyone, so the CORBA CTM vendors needed a standard to rally around. Besides the CORBA protocol, the most obvious standard needed was a component model, which would allow clients and third-party vendors to develop their business objects to one specification that would work in any CORBA CTM. Microsoft was, of course, pushing their component model as a standard—which was attractive because MTS was an actual working product—but Microsoft didn't support CORBA. The OMG (Object Management Group), the same people who developed the CORBA standard, were defining a server-side component model. This held promise because it was sure to be tailored to CORBA, but the OMG was slow in developing a standard—at least too slow for the evolving CTM market[6].

---

[5] Recently, the introduction of SOAP (Simple Object Access Protocol) brings into question the future of the CORBA IIOP protocol (Internet-InterOperability Protocol). It's obvious that these two protocols are competing to become the standard language-independent protocol for distributed computing. IIOP has been around for several years and is therefore far more mature, but as a late entry SOAP may quickly catch up by leveraging lessons learned in the development of IIOP.

[6] Eventually, CORBA's CTM component model was released and called CCM, for CORBA Component Model. It has seen lackluster acceptance is general, and was forced to adopt Enterprise JavaBeans as part of its component model just to be viable and interesting.

In 1997, Sun Microsystems was developing the most promising standard for server-side components called Enterprise JavaBeans. Sun offered some key advantages. First, Sun was respected and was known for working with vendors to define Java-based and vendor-agnostic APIs for common services. Sun had a habit of adopting the best ideas in the industry and then making the Java implementation an open standard—usually successfully. The Java database connectivity API, called JDBC, was a perfect example. Based largely on Microsoft's own ODBC, JDBC offered vendors a more flexible model for plugging in their own database access drivers. In addition, developers found the JDBC API much easier to work with. Sun was doing the same thing in its newer technologies like the JavaMail™ API and the Java Naming and Directory Interface (JNDI). These technologies were still being defined, but the collaboration among vendors was encouraging and the openness of the APIs was attractive.

Although CORBA offered an open standard, it attempted to standardize very low-level facilities like security and transactions. Vendors could not justify rewriting existing products such as TUXEDO and CICS to the CORBA standards. EJB got around that problem by saying it doesn't matter how you implement the low-level services; all that matters is all the facilities be applied to the components according to the specification—a much more palatable solution for existing and prospective CTM vendors. In addition, the Java language offered some pretty enticing advantages, not all of them purely technical. First, Java was a hot and sexy technology and simply making your product Java-compatible seemed to boost your exposure in the market. Java also offered some very attractive technical benefits. Java was more or less platform independent. A component model defined in the Java language would have definite marketing and technical benefits.

As it turned out, Sun had not been idle after it announced Enterprise JavaBeans. Sun's engineers had been working with several leading vendors to define a flexible and open standard to which vendors could easily adapt their existing products. This was a tall order because vendors had different kinds of servers including web servers, database servers, relational database servers, application servers, and early CTMs. It's likely that no one wanted to sacrifice their architecture for the common good, but eventually the vendors agreed on a model that was flexible enough to accommodate different implementations yet solid enough to support real mission-critical development. In December of 1997, Sun Microsystems released the first draft specification of Enterprise JavaBeans, EJB 1.0, and vendors have been flocking to the server-side component model ever since.

Benefits of a Standard Server-Side Component Model

So what does it mean to be a standard server-side component model? Quite simply, it means that you can develop business objects using the Enterprise JavaBeans (EJB) component model and expect them to work in any CTM that supports the complete EJB specification. This is a pretty powerful statement because it largely eliminates the biggest problem faced by potential customers of CORBA-based CTM products: fear of vendor "lock-in." With a standard server-side component model, customers can commit to using an EJB-compliant CTM with the knowledge that they can migrate to a better CTM if one becomes available. Obviously, care must be taken when using proprietary extensions developed by vendors, but this is nothing new. Even in relational database industry— which has been using the SQL standard for a couple of decades— optional proprietary extensions abound.

Having a standard server-side component model has benefits beyond implementation independence. A standard component model provides a vehicle for growth in the third- party products. If numerous vendors support EJB, then creating add-on products and component libraries is more attractive to software vendors. The IT industry has seen this type of cottage industry grow up around other standards like SQL, where hundreds of add-on products can be purchased to enhance business systems whose data is stored in SQL-compliant relational databases. Report generating tools and data warehouse products are typical examples. The GUI component industry has seen the growth of its own third-party products. A healthy market for component libraries already exists for GUI component models like Microsoft's ActiveX and Sun's original JavaBeans component models.

There are many examples of third-party product for Enterprise JavaBeans today Add-on products that provide services to EJB-compliant systems like credit card processing, legacy database access, and other business services have been introduced. These types of products make development of EJB systems simpler and faster than the alternatives, making the EJB component model attractive to corporate IS and server vendors alike. The industry has market grow for prepackaged EJB components in several domains including sales, finance, education, web content management, collaboration and other areas.

Titan Cruises: An Imaginary Business

To make things a little easier, and more fun, we will attempt to discuss all the concepts in this book in the context of one imaginary business, a cruise line called Titan. A cruise line makes a particularly interesting example because it incorporates several different businesses: a cruise has cabins that are similar to

hotel rooms, serves meals like a restaurant, offers various recreational opportunities, and needs to interact with other travel businesses.

This type of business is a good candidate for a distributed object system because many of the system's users are geographically dispersed. Commercial travel agents, for example, who need to book passage on Titan ships, will need to access the reservation system. Supporting many—possibly hundreds—of travel agents requires a robust transactional system to ensure that agents have access and reservations are completed properly.

Throughout this book we will build a fairly simple slice of Titan's EJB system that focuses on the process of making a reservation for a cruise. This will give us an opportunity to develop enterprise beans like Ship, Cabin, TravelAgent, ProcessPayment, and so forth. In the process, you will need to create relational database tables for persisting data used in the example. It is assumed that you are familiar with relational database management systems and that you can create tables according to the SQL statements provided. EJB can be used with any kind of database or legacy application, but relational databases seem to be the most commonly understood database so we have chosen this as the persistence layer.

What's Next?

In order to develop business objects using EJB, you have to understand the life cycle and architecture of EJB components. This means understanding conceptually how EJB's components are managed and made available as distributed objects. Developing an understanding of the EJB architecture is the focus of the next two chapters.

2

Architectural Overview

As you learned in Chapter 1, Enterprise JavaBeans is a component model for component transaction monitors, the most advanced type of business application server available today. To effectively use Enterprise JavaBeans, you need to understand the EJB architecture, so this book includes two chapters on the subject. This chapter explores the core of EJB: how enterprise beans are distributed as business objects. Chapter 3 explores the services and resource management techniques supported by EJB.

To be truly versatile, the EJB component design had to be smart. For application developers, assembling enterprise beans is simple, requiring little or no expertise in the complex system-level issues that often plague three-tier development efforts. While EJB makes it easy for application developers, it also provides system developers (the people who write EJB servers) with a great deal of flexibility in how they support the EJB specification.

The similarities among different component transaction monitors (CTMs) allow the EJB abstraction to be a standard component model for all of them. Each vendor's CTM is implemented differently, but they all support the same primary services and similar resource management techniques. The primary services and resource management techniques are covered in more detail in Chapter 3, but some of the infrastructure for supporting them is addressed in this chapter.

The Enterprise Bean Component

Enterprise JavaBeans server-side components come in three fundamentally different types: *entity*, *session*, and *message-driven beans*. Both session and entity beans are RMI based server-side components that are accessed using distributed object protocols. The message-driven bean, which is new to EJB 2.0, is an asynchronous server-side component that responds to JMS asyncrhonous messages.

A good rule of thumb is that entity beans model business concepts that can be expressed as nouns. For example, an entity bean might represent a customer, a piece of equipment, an item in inventory, or even a place. In other words, entity beans model real-world objects; these objects are usually persistent records in some kind of database. Our hypothetical cruise line will need entity beans that represent cabins, customers, ships, etc.

Session beans are an extension of the client application and are responsible for managing processes or tasks. A Ship bean provides methods for doing things directly to a ship but doesn't say anything about the context under which those actions are taken. Booking passengers on the ship requires that we use a Ship bean, but also requires a lot of things that have nothing to do with the Ship itself: we'll need to know about passengers, ticket rates, schedules, and so on. A session bean is responsible for this kind of coordination. Session beans tend to manage particular kinds of activities, for example, the act of making a reservation. They have a lot to do with the relationships between different enterprise beans. A TravelAgent session bean, for example, might make use of a Cruise, a Cabin, and a Customer—all entity beans—to make a reservation.

Similarly, the message-driven beans in EJB 2.0 are responsible for coordinating tasks involving other session and entity beans. The major difference between a message-driven bean and a session bean is how they are accessed. While a session bean provides a remote interface that defines which methods can be invoked, a message-driven bean does not. Instead, the message driven bean subscribes or listens for specific asynchronous messages to which it responds by processing the message and managing the activities of other beans in response to those messages. For example, a TravelAgent message-driven bean would receive to a asynchronous messages—perhaps from a legacy reservation system—from which it would coordinate the interactions of the Cruise, Cabin, and Customer beans to make a reservation.

The activity that a session or message-driven bean represents is fundamentally transient: you start making a reservation, you do a bunch of work, and then it's finished. The session and message-driven beans do not represent things in the database. Obviously, session and message-driven beans have lots of side effects on the database: in the process of making a reservation, you might create a new Reservation by assigning a Customer to a particular Cabin on a particular Ship. All of these changes would be reflected in the database by actions on the

respective entity beans. Session and message-driven beans like TravelAgent, which are responsible for making a reservation on a cruise, can even access a database directly and perform reads, updates, and deletes to data. But there's no TravelAgent record in the database—once the bean has made reservation is, it waits to process another.

What makes this distinction difficult is that it's extremely flexible. The relevant distinction for Enterprise JavaBeans is that an entity bean has persistent state; the session and message-driven beans model interactions but do not have persistent state.

Classes and Interfaces

A good way to understand the design of enterprise beans is to look at how you'd go about implementing one. To implement entity and session enterprise beans, you need to define the component interfaces, a bean class, and a primary key:

There are basically two kinds of component interfaces, remote and local. The remote interfaces are supported by both EJB 2.0 and 1.1 while the local component interfaces are new in EJB 2.0 and are not supported by EJB 1.1.

Remote interface
: The remote interface for an enterprise bean defines the bean's business methods that can be accessed from applications outside the EJB container: the business methods a bean presents to the outside world to do its work. It enforces conventions and idioms that are well suited for distributed object protocols. The remote interface extends `javax.ejb.EJBObject`, which in turn extends `java.rmi.Remote`. The remote interface is one of the bean's component interfaces and is used by session and entity beans in conjunction with the remote home interface.

Remote Home interface
: The home interface defines the bean's life cycle methods that can be accessed from applications outside the EJB container: the life-cycle methods for creating new beans, removing beans, and finding beans. It enforces conventions and idioms that are well suited for distributed object protocols. The home interface extends `javax.ejb.EJBHome`, which in turn extends `java.rmi.Remote`. The remote home interface is one of the bean's component interfaces and is used by session and entity beans in conjunction with the remote interface.

EJB 2.0: Local interface
: The local interface for an enterprise bean defines the bean's business methods that can be used by other beans co-located in the same EJB container: the business methods a bean presents other beans in the same address space. It allows beans to interact without the overhead of a distributed object protocol, which makes them more performant. The local

interface extends `javax.ejb.EJBLocalObject`. The local interface is one of the bean's component interfaces and is used by session and entity beans in conjunction with the local home interface.

EJB 2.0: Local Home interface
: The home interface defines the bean's life cycle methods that can be used by other beans co-located in the same EJB container: that is, the life-cycle methods a bean presents to other beans in the same address space. It allows beans to interact without the overhead of a distributed object protocol, which improves their performance. The local home interface extends `javax.ejb.EJBLocalHome`. The local home interface is one of the bean's component interfaces and is used by session and entity beans in conjunction with the local interface.

Bean class
: The session and entity bean classes actually implement the bean's business and life-cycle methods. Note, however, that the bean class for session and entity beans usually does not implement any of the bean's component interfaces directly. However, it must have methods matching the signatures of the methods defined in the remote and local interfaces and must have methods corresponding to some of the methods in the both the remote and local home interfaces. If this sounds perfectly confusing, it is. The book will clarify this as we go along. An entity bean must implement `javax.ejb.EntityBean`; a session bean must implement `javax.ejb.SessionBean`. The `EntityBean` and `SessionBean` extend `javax.ejb.EnterpriseBean`.

  The message-driven bean in EJB 2.0 does not use any of the component interfaces, because it is never accessed by method calls from other applications or beans. Instead, the message-driven bean contains a single method, `onMessage()`, which is called by the container when a new message arrives. So the message-driven bean does not have a component interface as does the session and entity beans, it only needs the bean class to operate. The message-driven bean class implements the `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener` interfaces. The JMS `MessageListener` interface is what makes a message-driven bean specific to JMS, instead of some other protocol. EJB 2.0 requires the use of JMS, but future versions may allow other messaging systems. The `MessageDrivenBean`, like the `EntityBean` and `SessionBean`, extends the `javax.ejb.EnterpriseBean` interface.

Primary key
: The primary key is a very simple class that provides a pointer into the database. Only entity beans need a primary key; the only requirement for this class is that it implements java.io.Serializable.

EJB 2.0 adds the crucial distinction between remote and local interfaces. Local interfaces provide a way for beans in the same container to interact efficiently; calls to methods in the local interface don't involve RMI; the methods in the

local interfaces don't need to declare that they throw `RemoteException`, and so on. An enterprise bean isn't required to provide a local interface, if you know when you're developing the enterprise bean that it will only interact with remote clients. Likewise, an enterprise bean doesn't need to provide a remote interface if it knows it will only be called by enterprise beans in the same container. You can provide local or remote component interface or both.

The complexity—particularly all the confusion about classes implementing the methods of an interface but not implementing the interface itself—comes about because enterprise beans exist in the middle between some kind of client software and some kind of database. The client never interacts with a bean class directly; it always uses the methods of the entity or session bean's component interfaces to do its work, interacting with stubs that are generated automatically. (For that matter, a bean that needs the services of another bean is just another client: it uses the same stubs, rather than interacting with the bean class directly.)

Although the local component interfaces (local and local home) in EJB 2.0 represent session and entity beans in the same address space and do not use distributed object protocols, they still represent a stub or proxy to the bean class. While there is no network between co-located beans, the stubs allow the container to monitor the interactions between co-located beans and apply security and transactions as appropriate.

Its important to note, that EJB 2.0's message-driven bean doesn't have any component interfaces, but it may become the client of other session or entity beans and interact with those beans through their component interfaces. The entity and session beans with which the message-driven bean interacts may be co-located, in which case it uses their local component interfaces, or they may be located in a different address space and EJB container, in which case the remote component interfaces are used.

There are also lots of interactions between an enterprise bean and its server. These interactions are managed by a "container," which is responsible for presenting a uniform interface between the bean and the server. (Many people use the terms "container" and "server" interchangeably, which is understandable because the difference between them isn't clearly defined.) The container is responsible for creating new instances of beans, making sure that they are stored properly by the server, and so on. Tools provided by the container's vendor do a tremendous amount of work behind the scenes. At least one tool will take care of creating the mapping between entity beans and records in your database. Other tools generate a lot of code based on the component interfaces and the bean class itself. The code generated does things like create the bean, store it in the database, and so on. This code (in addition to the stubs) is what actually implements the component interfaces, and is the reason your bean class doesn't have to.

Before going on, let's first establish some conventions. When we speak about an enterprise bean as a whole, its component interfaces, bean class, and so forth, we will call it by its common business name, followed by the word "bean." For example, an enterprise bean that is developed to model a cabin on a ship will be called the "Cabin EJB." Notice that we didn't use a constant width font for "Cabin." We do this because we are referring to all the parts of the bean (the component interfaces, bean class, etc.) as a whole, not just one particular part like the remote interface or bean class. The term *enterprise bean* denotes any kind of bean including entity, session, or message-driven beans. Similarly, *entity bean* denotes a entity type enterprise bean; *session bean* a session type enterprise bean; and *message-driven bean* a message-driven type enterprise bean. It's popular to use the acronym EJB for enterprise bean, a style adopted in this book to distinguish an enterprise bean as a whole from its component parts.

We will also use suffixes to distinguish between local component interfaces and remote component interfaces. When we are talking about the remote interface of the Cabin EJB we will use combine the common business name with the word *Remote*. For example, the remote interface for the Cabin EJB is called the `CabinRemote` interface. In EJB 2.0, the local component interface of the Cabin EJB would be the `CabinLocal` interface. The home interfaces follow the convention by adding the word *Home* to the mix. The remote and local home interfaces for the Cabin EJB would be `CabinHomeRemote` and `CabinHomeLocal` respectively. The bean class is always the common business name followed by the word *Bean*. For example, the Cabin EJB's bean class would be named `CabinBean`.

The remote interface

Having introduced the machinery, let's look at how to build an entity or stateful enterprise bean with remote component interfaces. In this section, we will examine the Cabin EJB, an entity bean that models a cabin on a cruise ship. Let's start with its remote interface.

We'll define the remote interface for a Cabin bean using the interface called `CabinRemote`, which defines business methods for working with cabins. All remote-interface types extend the `javax.ejb.EJBObject` interface.

```
import java.rmi.RemoteException;

public interface CabinRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String str) throws RemoteException;
    public int getDeckLevel() throws RemoteException;
    public void setDeckLevel(int level) throws RemoteException;
}
```

These are methods for naming the cabin and methods for setting the cabin's deck level; you can probably imagine lots of other methods that you'd need, but this is enough to get started. All of these methods declare that they throw

`RemoteException`, which is required of all methods on remote component interfaces, but not EJB 2.0's local component interfaces. EJB requires the use of Java RMI-IIOP conventions with remote component interfaces, although the underlying protocol can be CORBA IIOP, Java Remote Method Protocol (JRMP), or some other protocol. Java RMI-IIOP will be dis cussed in more detail in the next chapter.

The remote home interface

The remote home interface defines life-cycle methods used by clients of entity and session bean for locating enterprise beans. The remote home interface extends `javax.ejb.EJBHome`. We'll call the home interface for the Cabin bean `CabinHomeRemote` and define it like this:

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CabinHomeRemote extends javax.ejb.EJBHome {
    public Cabin create(Integer id)
        throws CreateException, RemoteException;
    public Cabin findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

The `create()` method will be responsible for initializing an instance of our bean. If your application needs it, you can provide other `create()` methods, with different arguments.

In addition to the `findByPrimaryKey()`, you are free to define other methods that provide convenient ways to look up Cabin beans—for example, you might want to define a method called `findByShip()` that returns all the cabins on a particular ship. Find methods like these are only used in entity beans and are not used in session beans -- and obviously not message-driven beans.

EJB 2.0: The bean class

EJB 2.0: The bean class

Now let's look at an actual entity bean. Here's the code for the `CabinBean`; it's a sparse implementation, but it will show you how the pieces fit together:

```
import javax.ejb.EntityContext;

public abstract class CabinBean implements javax.ejb.EntityBean {


    // EJB 1.0: return void
    public CabinPK ejbCreate(Integer id){
        setId(id);
```



```
        return null;
    }
    public void ejbPostCreate(int id){
        // do nothing
    }

    public abstract String getName();
    public abstract void setName(String str);

    public abstract int getDeckLevel();
    public abstract void setDeckLevel(int level);

    public abstract Integer getId( );
    public abstract void setId(Integer id);

    public void setEntityContext(EntityContext ctx){
        // not implemented
    }
    public void unsetEntityContext(){
        // not implemented
    }
    public void ejbActivate(){
        // not implemented
    }
    public void ejbPassivate(){
        // not implemented
    }
    public void ejbLoad(){
        // not implemented
    }
    public void ejbStore(){
        // not implemented
    }
    public void ejbRemove(){
        // not implemented
    }
}
```

You will have noticed that the `CabinBean` class is declared as abstract, as are several of its methods that access or update the EJB's persistent state. Also notices that there are no instance fields that hold the state information that these methods access. This is because we are working with a container-managed entity bean, which has its abstract methods implemented by the container system automatically—this will be explained in detail later in the book. EJB 2.0 container-managed entity beans are the only beans that are declared as abstract with abstract accessor methods. You won't see abstract classes and methods with other types of entity beans, session beans, or message-driven beans.

EJB 1.1: The bean class

Here's the code for the `CabinBean` in EBJ 1.1:

```
import javax.ejb.EntityContext;

public class CabinBean implements javax.ejb.EntityBean {

    public Integer id;
    public String name;
    public int deckLevel;

    // EJB 1.0: return void
    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
        // do nothing
    }

    public String getName(){
        return name;
    }    public void setName(String str){
        name = str;
    }

    public int getDeckLevel(){
        return deckLevel;
    }
    public void setDeckLevel(int level){
        deckLevel = level;
    }
    public Integer getId( ){
         return id;
    }
    public void setId(Integer id){
         this.id = id;
    }
    public void setEntityContext(EntityContext ctx){
        // not implemented
    }
    public void unsetEntityContext(){
        // not implemented
    }
    public void ejbActivate(){
        // not implemented
    }
    public void ejbPassivate(){
        // not implemented
    }
```



```
    public void ejbLoad(){
        // not implemented
    }
    public void ejbStore(){
        // not implemented
    }
    public void ejbRemove(){
        // not implemented
    }
}
```

EJB 2.0 and 1.1: The bean class

The set and get methods for the cabin's name and deck level are the `CabinBean`'s business methods; they match the business methods defined by the EJB's remote interface, `CabinRemote`. The `CabinBean` class has state and business behavior that models the concept of a cabin. The business methods are the only methods that are visible to the client application; the other methods are visible only to the EJB container or the bean class itself. For example, the `setId()`/`getId()` methods are defined in the bean class but not the remote interface, which means they can not be called by the entity bean's client. The other methods are required by the EJB component model and are not really part of the bean class's public business definition.

The `ejbCreate()` and `ejbPostCreate()` methods initialize the instance of the bean class when a new cabin record is to be added to the database. The last seven methods in the `CabinBean` are defined in the `javax.ejb.EntityBean` interface. These methods are state management callback methods. The EJB container invokes these callback methods on the bean class when important state management events occur. The `ejbRemove()` method, for example, notifies an entity bean that its data is about to be deleted from the database. The `ejbLoad()` and `ejbStore()` methods notify the bean instance that its state is being read or written to the database. The `ejbActivate()` and `ejbPassivate()` methods notify the bean instance that it is about to be activated or deactivated, a process that conserves memory and other resources. `setEntityContext()` provides the bean with an interface to the EJB container that allows the bean class to get information about itself and its surroundings. `unsetEntityContext()` is called by the EJB container to notify the bean instance that it is about to be dereference for garbage collection.

All these callback methods provide the bean class with *notifications* of when an action is about to be taken, or was just taken, on the bean class's behalf by the EJB server. These notifications simply inform the bean of an event, the bean doesn't have to do anything about it. The callback notifications tell the bean where it is during its life cycle, when it is about to be loaded, removed, deactivated, and so on. Most of the callback methods pertain to persistence, which can be done automatically for the bean class by the EJB container.

Because the callback methods are defined in the `javax.ejb.EntityBean` interface, the entity bean class must implement them, but it isn't required to do anything meaningful with the methods if it doesn't need to. Our bean, the `CabinBean`, won't need to do anything when these callback methods are invoked, so these methods are empty implementations. Details about these callback methods, when they are called and how a bean should react, are covered in Chapter 116.

The primary key

The primary key is a pointer that helps locate data that describes a unique record or entity in the database; it is used in the `findByPrimaryKey()` method of the home interface to locate a specific entity. Primary keys are defined by the bean developer and must be some type of serializable object. The Cabin EJB uses a simple `java.lang.Integer` type as its primary key. Its also possible to define custom primary keys, called compound primary keys, which represent complex primary keys consisting of several different fields. Primary keys are covered in detail in Chapter 110.

What about session beans?

`CabinBean` is an entity bean, but a session bean wouldn't be all that different. It would extend `SessionBean` instead of `EntityBean`; it would have an `ejbCreate()` method that would initialize the bean's state, but no `ejbPostCreate()`. Session beans don't have an `ejbLoad()` or `ejbStore()` because session beans are not persistent. While session beans have a `setSessionContext()` method, they don't have an `unsetSessionContext()` method. Finally, a session bean would provide an `ejbRemove()` method, which would be called to notify the bean that the client no longer needs it. However, this method wouldn't tell the bean that its data was about to be removed from the database, because a session bean doesn't represent data in the database.

Session beans don't have a primary key. That's because session beans are not persistent themselves, so there is no need for key that maps to the database. Session beans are covered in detail in Chapter 12.

EJB 2.0: What about message-driven beans?

Message-driven beans do not have component interfaces so there would not be a remote, local, or home interface defined for a message-driven bean. Instead the message-driven bean would define only a few callback methods, and not business methods. The callback methods include the `ejbCreate()` method which is called when the bean class is first created, the `ejbRemove()` method when the bean instance is about to be discarded from the system—usally when the container doesn't need it any longer—the `setMessageDrivenBeanContext()` and the `onMessage()` method.

The `onMessage()` method is called every time a new asynchronous message is delivered to the message-driven bean. The message-driven bean doesn't define `ejbPasivate()`/`ejbActivate()` or `ejbLoad()`/`ejbStore()` methods because it doesn't need them.

Message-driven beans don't have a primary key, for the same reason that session beans don't. They are not persistent, so there is no need for a key to the database. Message-driven beans are covered in detail in Chapter 13.

Deployment Descriptors and JAR Files

Much of the information about how beans are managed at runtime is not addressed in the interfaces and classes discussed previously. You may have noticed, for example, that we didn't talk about how beans interact with security, transactions, naming, and other services common to distributed object systems. As you know from prior discussions, these types of primary services are handled automatically by the EJB CTM server, but the EJB container still needs to know how to apply the primary services to each bean class at runtime. To do this, we use *deployment descriptors*.

Deployment descriptors serve a function very similar to property files. They allow us to customize behavior of software (enterprise beans) at runtime without having to change the software itself. Property files are often used with applications, but deployment descriptors are specific to a class of enterprise bean. Deployment descriptors are also similar in purpose to property sheets used in Visual Basic and PowerBuilder. Where property sheets allow us to describe the runtime attributes of visual widgets (background color, font size, etc.), deployment descriptors allow us to describe runtime attributes of server-side components (security, transactional context, etc.). Deployment descriptors allow certain runtime behaviors of beans to be customized, without altering the bean class or its interfaces.

When a bean class and its interfaces have been defined, a deployment descriptor for the bean is created and populated with data about the bean. Frequently, IDEs (integrated development environments) that support development of Enterprise JavaBeans will allow developers to graphically set up the deployment descriptors using visual utilities like property sheets. After the developer has set all the properties for a bean, the deployment descriptor is saved to a file. Once the deployment descriptor is complete and saved to a file, the bean can be packaged in a JAR file for deployment.

JAR (*J*ava *ar*chive) files are ZIP files that are used specifically for packaging Java classes (and other resources such as images) that are ready to be used in some type of application. JARs are used for packaging applets, Java applications, JavaBeans, Web applications (Servlets & JSPs), and Enterprise JavaBeans. A JAR file containing one or more enterprise beans includes the bean classes,

component interfaces, and supporting classes for each bean. It also contains one deployment descriptor, which is used for all the beans in the JAR files. When a bean is deployed, the JAR's path is given to the container's deployment tools, which read the JAR file. The container uses the deployment descriptor to learn about the beans contained in the JAR file.

When the JAR file is read at deployment time, the container tools read the deployment descriptor to learn about the bean and how it should be managed at runtime. The deployment descriptor tells the deployment tools what kind of beans are in the JAR file (`SessionBean` or `EntityBean`), how they should be managed in transactions, who has access to the beans at runtime, and other runtime attributes of the beans. The person who is deploying the bean can alter some of these settings, like transactional and security access attributes, to customize the bean for a particular application. Many container tools provide property sheets for graphically reading and altering the deployment descriptor when the bean is deployed. These graphical property sheets are similar to those used by bean developers.

The deployment descriptors help the deployment tools to add beans to the EJB container. Once the bean is deployed, the properties described in the deployment descriptors will continue to be used to tell the EJB container how to manage the bean at runtime.

When Enterprise JavaBeans 1.0 was released serializable classes were used for the deployment descriptor. Starting with Enterprise JavaBeans 1.1, the serializable deployment descriptor classes used in EJB 1.0 were dropped in favor of a more flexible file format based on XML (*E*xtensible *M*arkup *L*anguage). The XML deployment descriptors are text files structured according to a standard EJB DTD (Document Type Definition) that can be extended so the type of deployment information stored can evolve as the specification evolves. Chapter 16 provides a detailed description of EJB 2.0 deployment descriptors. This section provides a brief overview of XML deployment descriptors.

EJB 2.0: Deployment Descriptor

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
EnterpriseJavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-
jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <home>CabinHomeRemote</home>
            <remote>CabinRemote</remote>
            <local-home>CabinHomeLocal</local-home>
            <local>CabinLocal</local>
```



```
            <ejb-class>java.lang.Integer</prim-key-class>
            <persistence-type>Container</persistence-type>
            <reentrant>False</reentrant>
        </entity>
    </enterprise-beans>
</ejb-jar>
```

EJB 1.1: Deployment Descriptor

The following deployment descriptor might be used to describe the Cabin bean:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD
EnterpriseJavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-
jar_1_1.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CabinEJB</ejb-name>
            <home>CabinHomeRemote</home>
            <remote>CabinRemote</remote>
            <ejb-class>CabinBean</ejb-class>            <prim-key-
class>java.lang.Integer</prim-key-class>
            <persistence-type>Container</persistence-type>
            <reentrant>False</reentrant>
        </entity>
    </enterprise-beans>
</ejb-jar>
```

EJB 2.0 and 1.1: Elements of the XML Deployment Descriptor

The deployment descriptor for a real bean would have a lot more information; this example simply illustrates the type of information that you'll find in an XML deployment descriptor.

The second element in any XML document is `!DOCTYPE`. This element describes the organization that defined the DTD for the XML document, the DTD's version, and a URL location of the DTD. The DTD describes how a particular XML document is structured.

All the other elements in the XML document are specific to EJB. They do not represent all the elements used in deployment descriptors, but they illustrate the types of elements that are used. Here's what the elements mean:

`ejb-jar`

> The root of the XML deployment descriptor. All other elements must be nested below this one. It must contain one `enterprise-beans` element as well as other optional elements.

`enterprise-beans`

Contains declarations for all the enterprise beans described by this XML document. It may contain `entity`, `session` or `message-driven` (EJB 2.0) elements, which describe entity, session and message-driven entrprise beans respectively.

`entity`

Describes an entity bean and its deployment information. There must be one of these elements for every entity bean described by the XML deployment descriptor. The `session` element is used in the same way to describe a session bean. The `message-driven` element is different as it does *not* define any component interfaces.

`ejb-name`

The descriptive name of the enterprise bean. It's the name we use for the enterprise bean in conversation, when talking about the bean component as a whole.

`home`

The fully qualified class name of the remote home interface. This is the interface that defines the life-cycle behaviors (create, find, remove) of the enterprise bean to its clients outside the container system.

`remote`

The fully qualified class name of the remote interface. This is the interface that defines the enterprise bean's business methods to its clients outside the container system.

EJB 2.0: `local-home`

The fully qualified class name of the local home interface. This is the interface that defines the life-cycle behaviors (create, find, remove) of the enterprise bean to other co-located enterprise beans.

EJB 2.0: `local`

The fully qualified class name of the local interface. This is the interface that defines the enterprise bean's business methods to other co-located enterprise beans.

`ejb-class`

The fully qualified class name of the bean class. This is the class that implements the business methods of the bean.

`prim-key-class`

The fully qualified class name of the enterprise bean's primary key. The primary key is used to find the bean data in the database.

The last two elements in the deployment descriptor, the persistence-type and reentrant elements, express the persistence strategy and concurrency policies of the entity bean. These elements are explained in more detail later in the book.

As you progress through this book, you will be introduced to the elements that describe concepts we have not covered yet, so don't worry about knowing all of the things you might find in a deployment descriptor.

EJB objects and EJB home

The entity and session beans both declare the component interfaces that their clients will use to access them. Clients outside the container system, like Servlets or Java applications, will always use the enterprise bean's remote component interfaces, while clients that are other enterprise beans in the same container system will usually use local component interfaces to interact. This section explains in logical terms how the component interfaces are connected to instances of the bean class at runtime.

While this discussion helps you understand entity and session beans, it doesn't apply to EJB 2.0's message-driven beans at all, because they do not declare component interfaces. Message-driven beans are a very different kind of animal and a full description of message-driven beans is left to Chapter 13.

Now that you have a basic understanding of some of the enterprise beans parts (component interfaces, bean class, and deployment descriptor) it's time to talk a little more precisely about how these parts come together inside an EJB container system. Unfortunately, we can't talk as precisely as we'd like. There are a number of ways for an EJB container to implement these relationships; we'll show some of the possibilities. Specifically, we'll talk about how the container implements the component interface of entity and session beans, so that clients, applications outside the container or other co-located enterprise beans, can interact with and invoke methods on the bean class.

The two missing pieces are the EJB object itself and the EJB home. You will probably never see the EJB home and EJB object classes because their class definitions are proprietary to the vendor's EJB implementation and are generally not made public. This is good because it represents a separation of responsibilities along areas of expertise. As an application developer, you are intimately familiar with how your business environment works and needs to be modeled, so you will focus on creating the applications and beans that describe your business. System-level developers, the people who write EJB servers, don't understand your business, but they do understand how to develop CTMs and support distributed objects. It makes sense for system-level developers to apply their skills to mechanics of managing distributed objects but leave the business logic to you, the application developer. Let's talk briefly about the EJB object and the EJB home so you understand the missing pieces in the big picture.

The EJB object

This chapter has said a lot about a bean's remote and local interfaces, which extends the `EJBObject` and, for EJB 2.0, the `EJBLocalObject` interfaces

respectively. Who implements these interfaces? Clearly, the stub: we understand that much. But what about the server side?

On the server side, an EJB object is an object that implements the remote and/or local interfaces of the enterprise bean. Local interface are only available to EJB 2.0 container systems. It wraps the enterprise bean instance—that is, the enterprise bean class you've created (in our example, the `CabinBean`)—on the server and expands its functionality to include `javax.ejb.EJBObject` and/or `javax.ejb.EJBLocalObject` behavior.

You will have noticed that "and/or" is used a lot when talking about which interface the EJB object implements. That's because enterprise beans in EJB 2.0 can declare either the local interface, remote interface, or both! Local interfaces dot apply to EJB 1.1, so if you are working with that version, ignore references to them; they are only relevant to EJB 2.0 container systems.

In EJB 2.0, regardless of which interfaces the bean implements, we can think of the EJB object as implementing both. In reality there may be a special EJB object for the remote interface and another special EJB object for the local interface of each enterprise bean; that depends on the how the vendor choose to implement it. For our purposes the term EJB object will be used to talk about the implementation of either local or remote interfaces or both. The functionality of these interfaces is so similar from the EJB object's perspective that discussing separate EJB object implementations wouldn't be beneficial.

The EJB object is generated by the utilities provided by the vendor of your EJB container and is based on the bean classes and the information provided by the deployment descriptor. The EJB object wraps the bean instance and works with the container to apply transactions, security, and other system- level operations to the bean at runtime. Chapter 3 talks more about the EJB object's role with regard to system-level operations.

There are a number of strategies that a vendor can use to implement the EJB object; Figure 2-1 illustrates three possibilities using the `CabinRemote` interface. The same implementation strategies apply to the `CabinLocal` and `javax.ejb.EJBLocalObject` interfaces.

[FIGURE]

Figure 2-1: Three ways to implement the EJB object

In Figure 2-1(a), the EJB object is a classic wrapper because it holds a reference to the bean class and delegates the requests to the bean. Figure 2-1(b) shows that the EJB object class actually extends the bean class, adding functionality specific to the EJB container. In Figure 2-1(c), the bean class is no longer included in the model. In this case, the EJB object has both a proprietary implementation required by the EJB container and bean class method implementations that were copied from the bean class's definition.

The EJB object design that is shown in Figure 2-1(a) is perhaps the most common. Throughout this book, particularly in the next chapter, we will explain how EJB works with the assumption that the EJB object wraps the bean class instance as depicted in Figure 2-1(a). But the other implementations are used; it shouldn't make a difference which one your vendor has chosen. The bottom line is that you never really know much about the EJB object: its implementation is up to the vendor. Knowing that it exists and knowing that its existence answers a lot of questions about how enterprise beans are structured, should be sufficient. Everything that any client (including other enterprise beans) really needs to know about any bean is described by the remote and home interfaces.

The EJB home

The EJB home is a lot like the EJB object. It's another class that's generated automatically when you install an enterprise bean in a container. It implements all the methods defined by the home interfaces (local and remote) and is responsible for helping the container in managing the bean's life cycle. Working closely with the EJB container, the EJB home is responsible for locating, creating, and removing enterprise beans. This may involve working with the EJB server's resource managers, instance pooling, and persistence mechanisms, the details of which are hidden from the developer.

For example, when a create method is invoked on a home interface, the EJB home creates an instance of the EJB object which references a bean instance of the appropriate type. Once the bean instance is associated with the EJB object, the instance's matching `ejbCreate()` method is called. In the case of an entity bean, a new record is inserted into the database. With session beans the instance is simply initialized. Once the `ejbCreate()` method has completed, the EJB home returns a remote or local reference (i.e., a stub) for the EJB object to the client. The client can then begin to work with the EJB object by invoking business methods using the stub. The stub relays the methods to the EJB object; in turn, the EJB object delegates those method calls to the bean instance.

In EJB 2.0, how does the EJB home know which type of EJB object reference (local or remote) to return? It depends on which home interface is being used. If the client invokes a `create()` method on the remote home interface, the EJB home will return a remote interface reference. If the client is working with a local home interface, the EJB home will return a reference implementing the local interface. EJB 2.0 requires that the return type of remote home interface methods be remote interfaces, and that the return type of the local home interface methods be local interfaces.

```
// The Cabin EJB's remote home interface
public interface CabinHomeRemote extends javax.ejb.EJBHome {
    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;
    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
```



```
}


// The Cabin EJB's local home interface
public interface CabinHomeLocal extends javax.ejb.EJBHome {
    public CabinLocal create(Integer id)
        throws CreateException, RemoteException;
    public CabinLocalfindByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

Figure 2-3 illustrates the architecture of EJB with the EJB home and EJB object implementing the home interface and remote or local interface respectively. The bean class is also shown as being wrapped by the EJB object.

[FIGURE]

Figure 2-1: EJB architecture

Deploying a bean

The EJB objects and EJB homes are generated during the deployment process. After the files that define the bean (the component interfaces, and the bean classes) have been packaged into a JAR file, the bean is ready to be deployed: that is, added to an EJB container so that it can be accessed as a distributed component. During the deployment process, tools provided by the EJB container vendor generate the EJB object and EJB home classes by examining the deployment descriptor and the other interfaces and classes in the JAR file.

EJB 2.0: Local vs. Remote Support

Throughout this book we will consider the EJB object and EJB home as constructs that support both the remote and local component interfaces. In reality, we have no idea how the vendor chose to implement the EJB object and EJB home since they are only logical constructs and may not have equivalent software counterparts. It's important to remember that EJB object and EJB home are simply terms to describe the EJB container's responsibilities for supporting the component interfaces. We have chosen to give them a more concrete description in this book purely for instructional purposes, the EJB object and EJB home implementations discussed throughout this book are to be considered illustrative and a true representation of how these terms may be implemented.

Using Enterprise Beans

Let's look at how a client would work with an enterprise bean to do something useful. We'll start with the Cabin EJB that was defined earlier. A cabin is a thing or place whose description is stored in a database. To make the example a little

bit more real, imagine that there are other entity beans, including a Ship, Cruise, Ticket, Customer, Employee, and so on.

Getting Information from an Entity Bean

Imagine that a GUI client needs to display information about a particular cruise, including the cruise name, the ship name, and a list of cabins. Using the cruise ID obtained from a text field, we can use some of our beans to populate the GUI with data about the requested cruise. Here's what the code would look like:

```
CruiseHomeRemote cruiseHome = ... use JNDI to get the home
// Get the cruise id from a text field.
String cruiseID = textFields1.getText();
// Create an EJB primary key from the cruise id.
Integer pk = new java.lang.Integer.parseInt(cruiseID);
// Use the primary key to find the cruise.
CruiseRemote cruise = cruiseHome.findByPrimaryKey(pk);
// Set text field 2 to show the cruise name.
textField2.setText(cruise.getName());
// Get a remote reference to the ship that will be used
// for the cruise from the cruise bean.
ShipRemote ship = cruise.getShip();
// Set text field 3 to show the ship's name.
textField3.setText(ship.getName());

// Get a list of all the cabins on the ship as remote references
// to the cabin beans.
Collection cabins = ship.getCabins();
Iterator cabinItr = cabins.iterator();

// Iterate through the enumeration, adding the name of each cabin
// to a list box.
while( cabinItr.hasNext())
    CabinRemote cabin = (CabinRemote)cabinItr.next();
    listBox1.addItem(cabin.getName());
}
```

Let's start by getting a remote reference to the EJB home for an entity bean that represents a cruise. We are using a remote reference instead of a local one, because the client is a GUI Java application located outside the EJB container. In EJB 1.1, we don't have a choice because only remote component interfaces are supported anyway. It's not shown in the example, but references to the EJB home are obtained using JNDI. Java Naming and Directory Interface (JNDI) is a powerful API for locating resources, such as remote objects, on networks. It's a little too complicated to talk about here, but rest assured that it will be covered in subsequent chapters.

We read a cruise ID from a text field, use it to create a primary key, and use that primary key together with the EJB home to get a `CruiseRemote` reference, the

object that implements the business methods of our bean. Once we have the appropriate Cruise EJB, we can ask the Cruise EJB to give us a remote reference to a Ship EJB that will be used for the cruise. We can then get a `Collection` of remote Cabin EJB references from the Ship EJB and display the names of the Cabin EJBs in the client.

Entity beans model data and behavior. They provide a system with a reusable and consistent interface to data in the database. The behavior used in entity beans is usually focused on applying business rules that pertain directly to changing data. In addition, entity beans can model relationships with other entities. A ship, for example, has many cabins. We can get a list of cabins owned by the ship by invoking the `ship.getCabins()` method.

Entity beans are shared by many clients. An example is the Ship EJB. The behavior and data associated with a Ship EJB will be used concurrently by many clients on the system. There are only three ships in Titan's fleet, so it's easy to imagine that several clients will need to access these entities at the same time. Entity beans are designed to service multiple clients, providing fast, reliable access to data and behavior while protecting the integrity of data changes. Because entity beans are shared, we can rest assured that everyone is using the same entity and seeing the same data as it changes. In other words, we don't have duplicate entities with different representations of the same data.[1]

Modeling Workflow with Session Beans

Entity beans are useful for objectifying data and describing business concepts that can be expressed as nouns, but they're not very good at representing a process or a task. A Ship bean provides methods and behavior for doing things directly to a ship, but it does not define the context under which these actions are taken. The previous example retrieved data about cruises and ships; we could also have modified this data. And if we had gone to enough effort, we could have figured out how to book a passenger—perhaps by adding a Customer bean to a Cruise bean or adding a customer to a list of passengers maintained by the ship. We could try to shove methods for accepting payment and other tasks related to booking into our GUI client application, or even into the Ship or Cabin beans, but that's a contrived and inappropriate solution. We don't want business logic in the client application—that's why we went to a multitier architecture in the first place. Similarly, we don't want this kind of logic in our entity beans that represent ships and cabins. Booking passengers on a ship or scheduling a ship for a cruise are the types of activities or functions of the business, not the Ship or the Cabin bean, and are therefore expressed in terms of a process or task.

---

1 This is dependent on the isolation level set on the bean's data, which is discussed in more detail in Chapter 8.

Session beans act as agents for the client managing business processes or tasks; they're the appropriate place for business logic. A session bean is not persistent like an entity bean; nothing in a session bean maps directly into a database or is stored between sessions. Session beans work with entity beans, data, and other resources to control *workflow*. Workflow is the essence of any business system because it expresses how entities interact to model the actual business. Session beans control tasks and resources but do not themselves represent data.

The following code demonstrates how a session bean, designed to make cruise line reservations, might control the workflow of other entity and session beans to accomplish this task. Imagine that a piece of client software, in this case a user interface, obtains a remote reference to a TravelAgent session bean. Using the information entered into text fields by the user, the client application books a passenger on a cruise:

```
// Get the credit card number from the text field.
String creditCard = textField1.getText();
int cabinID = Integer.parseInt(textField2.getText());
int cruiseID = Integer.parseInt(textField3.getText());

// Create a new Reservation session passing in a reference to a
// customer entity bean.
TravelAgent travelAgent = TravelAgentHome.create(customer);

// Set cabin and cruise IDs.
travelAgent.setCabinID(cabinID);
travelAgent.setCruiseID(cruiseID);

// Using the card number and price, book passage.
// This method returns a Ticket object.
Ticket ticket = travelAgent.bookPassage(creditCard, price);
```

This is a fairly *coarse-grained* abstraction of the process of booking a passenger on a cruise. Coarse-grained means that most of the details of the booking process are hidden from the client. Hiding the *fine-grained* details of workflow is important because it provides us with more flexibility in how the system evolves and how clients are allowed to interact with the EJB system.

The following listing shows some of the code included in the `TravelAgentBean`. The `bookPassage()` method actually works with three entity beans, the Customer, Cabin, and Cruise beans, and another session bean, the ProcessPayment bean. The ProcessPayment bean provides several different methods for making a payment including check, cash, and credit card. In this case, we are using the ProcessPayment session to make a credit card purchase of a cruise ticket. Once payment has been made, a serializable `Ticket` object is created and returned to the client application.

```
public class TravelAgentBean implements javax.ejb.SessionBean {

    public Customer customer;
```



```
    public Cruise cruise;
    public Cabin cabin;

    public void ejbCreate(Customer cust) {
        customer = cust;
    }
    public Ticket bookPassage(CreditCard card, double price)
        throws IncompleteConversationalState {

        if (customer == null || cruise == null || cabin == null) {
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeRemote resHome = (ReservationHome)
                getHome("ReservationHome",ReservationHome.class);
            ReservationRemote reservation =
                resHome.create(customer, cruise, cabin, price);
            ProcessPaymentHomeRemote ppHome =
                (ProcessPaymentHome)getHome("ProcessPaymentHome",
                    ProcessPaymentHome.class);
            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            Ticket ticket =
                   new Ticket(customer,cruise,cabin,price);
            return ticket;
        } catch(Exception e){
            throw new EJBException(e);
        }
    }

// More business methods and EJB state management methods follow.
}
```

This example leaves out some details, but it demonstrates the difference in purpose between a session bean and an entity bean. Entity beans represent the behavior and data of a business object, while session beans model the workflow of beans. The client application uses the TravelAgent EJB to perform a task using other beans. For example, the TravelAgent EJB uses a ProcessPayment EJB and a Reservation EJB in the process of booking a passage. The ProcessPayment EJB processes a credit card and the Reservation EJB records the actual reservation in the system. Session beans can also be used to read, update, and delete data that can't be adequately captured in an entity bean. Session beans don't represent records or data in the database like entity beans but can access data in the database.

All the work performed by TravelAgent session bean could have been coded in the client application. Having the client interact directly with entity beans is a common but troublesome design approach because it ties the client directly to the details of the business tasks. This is troublesome for two reasons: any

change in the entity beans and their interaction require changes to the client, and it's very difficult to reuse the code that models the workflow.

Session beans are coarse-grained components that allow clients to perform tasks without being concerned with the details that make up the task. This allows developers to update the session bean, possibly changing the workflow, without impacting the client code. In addition, if the session bean is properly defined, other clients that perform the same tasks can reuse it. The ProcessPayment session bean, for example, can be reused in many other areas besides reservations, including retail and wholesale sales. For example, the ship's gift shop could use the ProcessPayment EJB to process purchases. As a client of the ProcessPayment EJB, the TravelAgent EJB doesn't care how ProcessPayment works; it's only interested in the ProcessPayment EJB's coarse-grained interface, which validates and records charges.

Moving workflow logic into a session bean also helps to thin down the client applications and reduce network traffic and connections. Excessive network traffic is actually one of the biggest problems in distributed object systems. Excessive traffic can overwhelm the server and clog the network, hurting response times and performance. Session beans, if used properly, can substantially reduce network traffic by limiting the number of requests needed to perform a task. In distributed objects, every method invocation produces network traffic. Distributed objects communicate requests using an RMI loop. This requires that data be streamed between the stub and skeleton with every method invocation. With session beans, the interaction of beans in a workflow is kept on the server. One method invocation on the client application results in many method invocations on the server, but the network only sees the traffic produced by one method call on the session bean. In the TravelAgent EJB, the client invokes `bookPassage()`, but on the server, the `bookPassage()` method produces several method invocations on the component interfaces of other enterprise beans. For the network cost of one method invocation, the client gets several method invocations. In EJB 2.0 we would have used the local component interfaces because they are much more efficient.

In addition, session beans reduce the number of network connections needed by the client. The cost of maintaining many network connections can be very high, so reducing the number of connections that each client needs is important in improving the performance of the system as a whole. When session beans are used to manage workflow, the number of connections that each client has to the server is substantially reduced, which improves the EJB server's performance. Figure 2-5 compares the network traffic and connections used by a client that only uses entity beans to that used by a client that uses session beans.

[FIGURE]

Figure 2-3: Session beans reduce network traffic and thin down clients

Session beans also limit the number of stubs used on the client, which saves the client memory and processing cycles. This may not seem like a big deal, but without the use of session beans, a client might be expected to manage hundreds or even thousands of remote references at one time. In the TravelAgent EJB, for example, the `bookPassage()` method works with several remote references, but the client is only exposed to the remote reference of the TravelAgent EJB.

Stateless and stateful session beans

Session beans can be either stateful or stateless. Stateful session beans maintain *conversational state* when used by a client. Conversational state is not written to a database; it's state that is kept in memory while a client uses a session. Maintaining conversational state allows a client to carry on a conversation with an enterprise bean. As each method on the enterprise bean is invoked, the state of the session bean may change, and that change can affect subsequent method calls. The TravelAgent session bean, for example, may have many more methods than the `bookPassage()` method. The methods that set the cabin and cruise IDs are examples. These set methods are responsible for modifying conversational state. They convert the IDs into remote references to Cabin and Cruise EJBs that are later used in the `bookPassage()` method. Conversational state is only kept for as long as the client application is actively using the bean. Once the client shuts down or releases the TravelAgent EJB, the conversational state is lost forever. Stateful session beans are not shared among clients; they are dedicated to the same client for the life of the enterprise bean.

Stateless session beans do not maintain any conversational state. Each method is completely independent and uses only data passed in its parameters. The ProcessPayment EJB is a perfect example of a stateless session bean. The ProcessPayment EJB doesn't need to maintain any conversational state from one method invocation to the next. All the information needed to make a payment is passed into the `byCreditCard()` method. Stateless session beans provide the highest performance in terms of throughput and resource consumption compared to entity and stateful session beans because only a few stateless session bean instances are needed to serve hundreds, possibly thousands of clients. Chapter 12 talks more about the use of stateless session beans.

EJB 2.0: Accessing EJB with Message-Driven Beans

Message-driven beans are integration points for other applications interested in working with EJB applications. Java applications or legacy systems that need to access an EJB application can send messages via JMS to message-driven beans. The message-driven beans can then process those messages and perform tasks using other entity and session beans.

In many ways, message-driven beans fulfill the same role as session beans by managing the workflow of entity and session beans to complete a given task.

The task to be completed is initiated by an asynchronous message, which has been sent by an application using JMS. Unlike session beans, which respond to business methods invoked on their component interfaces, a message-driven bean responds to asynchronous messages, which are delivered to the message-driven bean through its `onMessage()` method. The fact that the messages are asynchronous means the client that send message doesn't expect and is not waiting for a reply. The messaging client simply sends the message and forgets about it.

As an example, we can recast the TravelAgent EJB developed earlier as a message-driven bean:

```
public class TravelAgentMDBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener {

    CustomerHomeLocal customerHome;
    CruiseHomeLocal cruiseHome;
    CabinHomeLocal cabinHome;
    ReservationHomeLocal reservationHome;
    ProcessPaymentHomeLocal paymentHome;

    public void onMessage(Message msg) {

      try {

        MapMessage message = (MapMessage)msg;
        Integer customerID =
                (Integer) message.getObject("customer_id");
        Integer cruiseID =
                (Integer) message.getObject("cruise_id");
        Integer cabinID =
                (Integer)message.getObject("cabin_id");
        double price = message.getDouble("price");

        CustomerLocal customer =
                     customerHome.findByPrimaryKey(customerID);
        CruiseLocal cruise=
                    cruiseHome.findByPrimaryKey(cruiseID);
        CabinLocal cabin = cabinHome.findByPrimaryKey(cabin_id);

        ReservationLocal reservation =
        reservationHome.create(customer, cruise, cabin, price);

        ProcessPaymentLocal process = paymentHome.create();
        process.byCredit(customer, card, price);

     } catch(Exception e){
           throw new EJBException(e);
     }
  }
```



```
// More business methods and EJB state management methods follow.
}
```

Notice that all the information about the reservation is obtained from the message delivered to the message-driven bean. In JMS messages can take many forms, one of which is the `javax.jms.MapMessage` used in this example, which carries name-value pairs. Once the information is obtained from the message and the enterprise bean references are obtained, the reservation is processed the same as it was in the session bean. The only difference is that a `Ticket` is not returned to the caller, because message-driven beans don't have to respond to the caller, the process is asynchronous.

Message-driven beans, like stateless session beans, do not maintain any conversational state. The processing of each new message is independent from the previous for subsequent messages.

As was mentioned before the message-driven bean is very different in many respects from entity and session beans, so it's a bit unclear don't worry it will be explained in detail in Chapter 13, *Message-Driven Beans.*

The Bean-Container Contract

The environment that surrounds the beans on the EJB server is often referred to as the *container*. The container is more a concept than a physical construct. Conceptually, the container acts as an intermediary between the bean class and the EJB server. The container manifests and manages the EJB objects and EJB homes for a particular type of bean and helps these constructs to manage bean resources and apply primary services like transactions, security, concurrency, naming, and so forth at runtime. Conceptually, an EJB server may have many containers, each of which may contain one or more types of enterprise beans. As you will discover a little later, the container and the server are not clearly different constructs, but the EJB specification defines the component model in terms of the container's responsibilities, so we will follow that convention here.

Enterprise beans components interact with the EJB container through a well-defined component model. The `EntityBean`, `SessionBean`, and `MessageDrivenBean` (EJB 2.0) interfaces are the bases of this component model. As we learned earlier, these interfaces provide callback methods that notify the bean class of state management events in its life cycle. At runtime, the container invokes the callback methods on the bean instance when appropriate state management events occur. When the container is about to write an entity bean instance's state to the database, for example, it first calls the bean instance's `ejbStore()` method. This provides the bean instance with an opportunity to do some clean up on its state just before it's written to the database. The `ejbLoad()` method is called just after the bean's state is

populated from the database, providing the bean developer with an opportunity to manage the bean's state before the first business method is called.[2] Other callback methods can be used by the bean class in a similar fashion. EJB defines when these various callback methods are invoked and what can be done within their context. This provides the bean developer with a predictable runtime component model.

While all the callback methods are declared in bean interfaces, a meaningful implementation of the methods is not mandatory. In other words, the method body of any or all of the callback methods can be left empty in the bean class. Beans that implement one or more callback methods are usually more sophisticated and access resources that are not managed by the EJB system. Enterprise beans that wrap legacy systems often fall into this category. The only exception to this is the `onMessage()` method, which is defined in the `MessageDrivenBean` interface. This method must be implemented if the message-driven bean is to do anything useful.

`javax.ejb.EJBContext` is an interface that is implemented by the container and is also a part of the bean-container contract. Entity beans use a subclass of `javax.ejb.EJBContext` called `javax.ejb.EntityContext`. Session beans use a subclass called the `javax.ejb.SessionContext`. Message-driven beans use the subclass `javax.ejb.MessageDrivenContext`. These `EJBContext` types provide the bean class with information about its container, the client using the enterprise bean, and the bean itself. They also provide other functionality that is described in more detail in Chapters 11, 12 and 13. The important thing about the `EJBContext` types is that they provide the enterprise bean with information about the world around it, which the enterprise bean can use while processing requests from both clients and callback methods from the container.

In addition to the `EJBContext`, EJB 1.1 and 2.0 have expanded the enterprise bean's interface with the container to include a JNDI name space, called the environment context, which provides the bean with a more flexible and extensible bean-container interface. The JNDI environment context is discussed in detail later in this book.

The Container-Server Contract

The container-server contract is not defined by the Enterprise JavaBeans specification. This was done to facilitate maximum flexibility for vendors defining their EJB server technologies. Other than isolating the beans from the server, the

---

2 The `ejbLoad()` and `ejbStore()` behavior illustrated here is for container-managed persistence. With bean-managed persistence the behavior is slightly different. This is examined in detail in Chapter 9.

container's responsibility in the EJB system is a little vague. The EJB specification only defines a bean-container contract and does not define the container-server contract. It is difficult to determine, for example, exactly where the container ends and the server begins when it comes to resource management and other services.

In the first few generations of EJB servers this ambiguity has not been a problem because most EJB server vendors also provide EJB containers. Since the vendor provides both the container and the server, the interface between the two can remain proprietary. In future generations of the EJB specification, however, some work may be done to define the container-server interface and delimit the responsibilities of the container.

One advantage of defining a container-server interface is that it allows third-party vendors to produce containers that can plug into any EJB server. If the responsibilities of the container and server are clearly defined, then vendors who specialize in the technologies that support these different responsibilities can focus on developing the container or server as best matches their core competency. The disadvantage of a clearly defined container-server interface is that the plug-and-play approach could impact performance. The high level of abstraction that would be required to clearly separate the container interface from the server, would naturally lead to looser binding between these large components, which could result in lower performance. The following rule of thumb best describes the advantages and disadvantages associated with a container-server interface: the tighter the integration, the better the performance; the higher the abstraction, the greater the flexibility. The biggest deterrent to defining a container-server interface is that it would require the definition of low-level facilities, which was one of the problems that established CTM vendors had with CORBA. Allowing vendors to implement low-level facilities like transactions and security as they see fit is one of EJB's biggest attractions for vendors[3].

Many EJB-compliant servers actually support several different kinds of middleware technologies. It's quite common, for example, for an EJB server to support the vendor's proprietary CTM model as well as EJB, Servlets, web server functionality, JMS provider, and other server technologies. Defining an EJB container concept is useful for clearly distinguishing that part of the server that supports EJB from all the other services it provides.

This said, we could define the responsibilities of containers and servers based on current implementations of the EJB specification. In other words, we could examine how current vendors are defining the container in their servers and use

---

[3] Of all the commercial and open source EJB servers available today only one has experimented with defining a container-server interface, OpenEJB. OpenEJB is an open source EJB container system developed by Richard Monson-Haefel, the author of this book.

this as a guide. Unfortunately, the responsibilities of the container in each EJB server largely depend on the core competency of the vendor in question. Database vendors, for example, implement containers differently from TP monitor vendors. The strategies for assigning responsibilities to the container and server are so varied that it would provide little value in understanding the overall architecture to discuss the container and server separately. Instead, this book addresses the architecture of the EJB system as if the container and server were one component.

The remainder of this book treats the EJB server and the container as the same thing and refers to them collectively as the EJB server, container, system, or environment.

Summary

This chapter covered a lot of ground describing the basic architecture of an EJB system. At this point you should understand that beans are business object components. The home interfaces define life-cycle methods for creating, finding, and destroying beans and the remote and local interfaces define the public business methods of the bean. Message-driven beans do not have component interfaces. The bean class is where the state and behavior of the bean are implemented.

There are three basic kinds of beans: entity, session, and message-driven. Entity beans are persistent and represent a person, place, or thing. Session beans are extensions of the client and embody a process or a workflow that defines how other beans interact. Session beans are not persistent, receiving their state from the client, and they live only as long as the client needs them. Message-driven beans in EJB 2.0 are integration points that allow other applications to interact with EJB applications using JMS asynchronous messaging. Message-driven beans, like stateless session beans, are not persistent and do not maintain conversational state.

The EJB object and EJB home are conceptual constructs that delegate method invocations to session and entity beans from the client and help the container to manage the enterprise bean at runtime. The clients of entity and session beans do not interact with the instances of the bean class directly. Instead, the client software interacts with `EJBObject` and `EJBHome` stubs, which are connected to the EJB object and EJB homes respectively. The EJB object implements the remote interface and expands the bean class's functionality. The EJB home implements the home interface and works closely with the container to create, locate, and remove beans.

Beans interact with their container through the well-defined bean-container contract. This contract provides callback methods, the `EJBContext`, and the JNDI environment context. The callback methods notify the bean class that it is

involved in state management event. The `EJBContext` and JNDI environment context provides the bean instance with information about its environment. The container-server contract is not well defined and remains proprietary at this time. Future versions of EJB may specify the container-server contract.

3

Resource Management and the Primary Services

Chapter 2 discussed the basic architecture of Enterprise JavaBeans (EJB), including the relationship between the bean class, component interfaces, the EJB object and EJB home, and the EJB container. These architectural components define a common model for distributed server-side components in component transaction monitors (CTMs).

One of the reasons CTMs are such great distributed object platforms is that they do more than just distribute objects: they manage the resources used by distributed objects. CTMs are designed to manage thousands, even millions, of distributed objects simultaneously. To be this robust, CTMs must be very smart resource managers, managing how distributed objects use memory, threads, database connection, processing power, etc. EJB recognizes that some of the resource management techniques employed by CTMs are very common, and it defines interfaces that help developers create beans that can take advantage of these common practices.

EJB CTMs are also great distributed object brokers. Not only do they help clients locate the distributed objects they need, they also provide many services that make it much easier for a client to use the objects correctly. CTMs commonly support six primary services: concurrency, transaction management, persistence, object distribution, naming, and security. These services provide the kind of infrastructure that is necessary for a successful three-tier system.

With the introduction of message-driven beans in EJB 2.0, Enterprise JavaBeans goes beyond most CTMs by expanding the platforms responsibility to include

managing asynchronous messaging components. CTMs have historically been responsible only for managing RMI-based distributed objects. While the method of access is different for message-driven beans, EJB is still responsible for managing the primary services for message-driven beans just as it does for session and entity beans.

This chapter discusses both the resource management facilities and the primary services that are available to Enterprise JavaBeans.

Resource Management

One of the fundamental benefits of using EJB servers is that they are able to handle heavy workloads while maintaining a high level of performance. A large business system with many users can easily require thousands of objects—even millions of objects—to be in use simultaneously. As the number of interactions among these objects increase, concurrency and transactional concerns can degrade the system's response time and frustrate users. EJB servers increase performance by synchronizing object interactions and sharing resources.

There is a relationship between the number of clients and the number of distributed objects that are required to service them. As client populations increase, the number of distributed objects and resources required increases. At some point, the increase in the number of clients will impact performance and diminish throughput. EJB explicitly supports two mechanisms that make it easier to manage large numbers of beans at runtime: instance pooling and activation.

Instance Pooling

The concept of pooling resources is nothing new. A commonly used technique is to pool database connections so that the business objects in the system can share database access. This trick reduces the number of database connections needed, which reduces resource consumption and increases throughput. Pooling and reusing database connections is less expensive than creating and destroying connections as needed. Some CTMs also apply resource pooling to server-side components; this technique is called *instance pooling*. Instance pooling reduces the number of component instances, and therefore resources, needed to service client requests. In addition, it is less expensive to reuse pooled instances than to frequently create and destroy instances.

As you already know, EJB clients of session and entity beans interact with these types of enterprise beans through the remote, and for EJB 2.0, the local interfaces that are implemented by EJB objects. Client applications never have direct access to the actual session or entity bean. Instead, they interact with EJB objects, which wrap bean instances. Similarly, JMS clients in EJB 2.0 never interact with message-driven beans directly. They send messages which are routed to the EJB

container system. The EJB container then delivers these messages to the proper message-driven bean.

Instance pooling leverages indirect access to enterprise beans to provide better performance. In other words, since clients never access beans directly, there's no fundamental reason to keep a separate copy of each enterprise bean for each client. The server can keep a much smaller number of enterprise beans around to do the work, reusing enterprise bean instance to service different requests. Although this sounds like a resource drain, when done correctly, it greatly reduces the resources actually required to services all the client requests.

The entity bean life cycle

To understand how instance pooling works for RMI components (session and entity beans), let's examine the life cycle of an entity bean. EJB defines the life cycle of an entity bean in terms of its relationship to the instance pool. An entity bean exists in one of three states:

No state

> When a bean instance is in this state, it has not been instantiated yet. We identify this state to provide a beginning and an end for the life cycle of a bean instance.

Pooled state

> When an instance is in the pooled state, it has been instantiated by the container but has not yet been associated with an EJB object.

Ready State

> A bean instance in this state has been associated with an EJB object and is ready to respond to business method invocations.

Overview of state transitions

Each EJB vendor implements instance pooling for entity beans differently, but all instance pooling strategies attempt to manage collections of bean instances so that they are quickly accessible at runtime. To create an instance pool, the EJB container creates several instances of a bean class and then holds onto them until they are needed. As clients make business method requests, bean instances from the pool are assigned to the EJB objects associated with the clients. When the EJB object doesn't need the instance any more, it's returned to the instance pool. An EJB server maintains instance pools for every type of bean deployed. Every instance in an instance pool is *equivalent*; they are all treated equally. Instances are selected arbitrarily from the instance pool and assigned to EJB objects as needed.

Soon after the bean instance is instantiated and placed in the pool, it's given a reference to a `javax.ejb.EJBContext` provided by the container. The `EJBContext` provides an interface that the bean can use to communicate with

the EJB environment. This `EJBContext` becomes more useful when the bean instance moves to the Ready State. Enterprise beans also have a JNDI context called the environment naming context. The function of the environment naming context is not critical to this discussion and will be addressed in more detail later in the chapter.

When a client uses an EJB home to obtain a remote or local interface to a bean, the container responds by creating an EJB object. Once created, the EJB object is assigned a bean instance from the instance pool. When a bean instance is assigned to an EJB object, it officially enters the Ready State. From the Ready State, a bean instance can receive requests from the client and callbacks from the container. Figure 3-1 shows the sequence of events that result in an EJB object wrapping a bean instance and servicing a client.

[FIGURE]

Figure 3-1: A bean moves from the instance pool to the Ready State

When a bean instance moves into the Ready State, the `EntityContext` takes on new meaning. The `EntityContext` provides information about the client that is using the bean. It also provides the instance with access to its own EJB home and EJB object, which is useful when the bean needs to pass references to itself to other instances, or when it needs to create, locate, or remove beans of its own class. So the `EntityContext` is not a static class; it is an interface to the container and its state changes as the instance is assigned to different EJB objects.

When the client is finished with a bean's remote reference, either the remote reference passes out of scope or one of the bean's remove methods is called.[1] Once a bean has been removed or is no longer in scope, the bean instance is disassociated from the EJB object and returned to the instance pool. Bean instances can also be returned to the instance pool during lulls between client requests. If a client request is received and no bean instance is associated with the EJB object, an instance is retrieved from the pool and assigned to the EJB object. This is called *instance swapping*.

After the bean instance returns to the instance pool, it is again available to service a new client request. Figure 3-3 illustrates the life cycle of a bean instance.

[FIGURE]

Figure 3-2: Life cycle of a bean instance

---

1 The `EJBHome`, `EJBLocalHome`, `EJBObject`, and `EJBLocalObject` interfaces all define methods that can be used to remove a bean.

The number of instances in the pool fluctuates as instances are assigned to EJB objects and returned to the pool. The container can also manage the number of instances in the pool, increasing the count when client activity increases and lowering the count during less active periods.

Instance swapping

Stateless session beans offer a particularly powerful opportunity to leverage instance pooling. A stateless session bean does not maintain any state between method invocations. Every method invocation on a stateless session bean operates independently, performing its task without relying on instance variables. This means that any stateless session instance can service requests for any EJB object of the proper type, allowing the container to swap bean instances in and out between method invocations made by the client.

Figure 3-5 illustrates this type of instance swapping between method invocations. In Figure 3-5(a), instance A is servicing a business method invocation delegated by EJB object 1. Once instance A has serviced the request, it moves back to the instance pool (Figure 3-5(b)). When a business method invocation on EJB object 2 is received, instance A is associated with that EJB object for the duration of the operation (Figure 3-5(c)). While instance A is servicing EJB object 2, another method invocation is received by EJB object 1 from the client, which is serviced by instance B (Figure 3-5(d)).

[FIGURE]

Figure 3-3: Stateless session beans in a swapping strategy

Using this swapping strategy allows a few stateless session bean instances to serve hundreds of clients. This is possible because the amount of time it takes to perform most method invocations is substantially shorter than the pauses between method invocations. The periods in a bean instance's life when it is not actively servicing the EJB object are unproductive; instance pooling minimizes these inactive periods. When a bean instance is finished servicing a request for an EJB object, it is immediately made available to any other EJB object that needs it. This allows fewer stateless session instances to service more requests, which decreases resource consumption and improves performance.

Stateless session beans are declared stateless in the deployment descriptor. Nothing in the class definition of a session bean is specific to being stateless. Once a bean class is deployed as stateless, the container assumes that no conversational state is maintained between method invocations. So a stateless bean can have instance variables, but because bean instances can be servicing several different EJB objects, they should not be used to maintain conversational state.

Implementations of instance pooling vary, depending on the vendor. One way that instance pooling implementations often differ is in how instances are

selected from the pool. Two of the common strategies are FIFO and LIFO. The FIFO (first in, first out) strategy places instances in a queue, where they wait in line to service EJB objects. The LIFO (last in, first out) uses more of stack strategy, where the last bean that was added to the stack is the first bean assigned to the next EJB object. Figure 3-5 uses a LIFO strategy.

EJB 2.0: Message-Driven Beans and Instance Pooling

Message-driven beans, like stateless session beans, do not maintain state specific to a client request, which makes them an excellent component for instance pooling.

In most EJB containers a pool of each type of message-driven bean is used to service incoming messages; each type of message-driven bean has its own instance pool. Message-driven beans subscribe or listen to a specific message destination, which is a kind of address used when sending messages. When a JMS client sends an asynchronous message to a specific destination, it is delivered to EJB container. The EJB container will first determine which message-driven bean subscribes to that destination, and then it will choose an instance of that type from the instance pool to process the message. Once the message-driven bean instance has finished processing the message (when the `onMessage( )` method returns) the EJB container will return the instance to its instance pool. An EJB container can process hundreds, possibly thousands, of messages concurrently by leveraging instance pools. Figure 3-X illustrates how client requests are processed by an EJB container.

FIGURE 3-X

Figure 3-x: Message-Driven bean instance pooling

In Figure 3-X A the top JMS client delivers a message to Destination A and the bottom JMS client delivers a message to Destination B. The EJB container chooses an instance of MessageDrivenBean_1 to process the message intended to Destination A, and an instance of MessageDrivenBean_2 to process the message intended for Destination B. The bean instances are removed from the pool and assigned and used to process the messages.

A moment later the middle JMS client sends a message to Destination B, at this point the first two messages have already been processed and the container is returning the instances to their respective pools. As the new message comes in the container choose a new instance of MessageDrivenBean_2 to process the message.

Message driven beans are always deployed to process messages from a specific destination. In the above example, instances of MessageDrivenBean_1 only process messages for Destination A, while instances of MessageDrivenBean_2 only processes messages for Destination B. Several messages for the same destination can be processed at the same time. If, for example, a hundred

messages for Destination A all arrived at the same time from a hundred different JMS clients, the EJB container would simply choose a hundred instances of MessageDrivenBean_1 to process the incoming messages; each instance is assigned a message.

The ability to concurrently process messages makes the message-driven bean an extremely powerful enterprise bean on the same playing field with session and entity beans. They are truly first class components, and an important addition to the Enterprise JavaBeans platform.

The Activation Mechanism

Unlike the other type of enterprise beans, stateful session beans maintain state between method invocations. This is called *conversational state* because it represents the continuing conversation with the stateful session bean's client. The integrity of this conversational state needs to be maintained for the life of the bean's service to the client. Stateful session beans do not participate in instance pooling like stateless session, entity, and message-driven beans. Instead, activation is used with stateful session beans to conserve resources. When an EJB server needs to conserve resources, it can evict stateful session beans from memory. This reduces the number of instances maintained by the system. To passivate the bean and preserve its conversational state, the bean's state is serialized to a secondary storage and maintained relative to its EJB object. When a client invokes a method on the EJB object, a new stateful instance is instantiated and populated from the passivated secondary storage.

Passivation is the act of disassociating a stateful bean instance from its EJB object and saving its state. Passivation requires that the bean instance's state be held relative to its EJB object. After the bean has been passivated, it is safe to remove the bean instance from the EJB object and evict it from memory. Clients are completely unaware of the deactivation process. Remember that the client uses the bean's remote interface, which is implemented by an EJB object, and therefore does not directly communicate with the bean instance. As a result, the client's connection to the EJB object can be maintained while the bean is passivated.

Activating a bean is the act of restoring a stateful bean instance's state relative to its EJB object. When a method on the passivated EJB object is invoked, the container automatically instantiates a new instance and sets its fields equal to the data stored during passivation. The EJB object can then delegate the method invocation to the bean as normal. Figure 3-7 shows activation and passivation of a stateful bean. In Figure 3-7(a), the bean is being passivated. The state of instance B is read and held relative to the EJB object it was serving. In Figure 3-7(b), the bean has been passivated and its state preserved. Here, the EJB object is not associated with a bean instance. In Figure 3-7(c), the bean is being activated. A new instance, instance C, has been instantiated and associated with the EJB object, and is in the process of having its state

populated. The instance C is populated with the state held relative to the EJB object.

[FIGURE]

Figure 3-4: The activation process

The exact mechanism for activating and passivating stateful beans is up to the vendor, but all stateful beans are serializable and thus provide at least one way of temporarily preserving their state. While some vendors take advantage of the Java serialization mechanism, the exact mechanism for preserving the conversational state is not specified. As long as the mechanism employed follows the same rules as Java serialization with regard to transitive closure of serializable objects, any mechanism is legal. Because Enterprise JavaBeans also supports other ways of saving a bean's state, the transient property is not treated the same when activating a passivated bean as it is in Java serialization. In Java serialization, transient fields are always set back to the initial value for that field type when the object is deserialized. Integers are set to zero, Booleans to `false`, object references to `null`, etc. In EJB, transient fields are not necessarily set back to their initial values but can maintain their original values, or any arbitrary value, after being activated. Care should be taken when using transient fields, since their state following activation is implementation specific.

The activation process is supported by the state-management callback methods discussed in Chapter 2. Specifically, the `ejbActivate()` and `ejbPassivate()` methods notify the stateful bean instance that it is about to be activated or passivated, respectively. The `ejbActivate()` method is called immediately following the successful activation of a bean instance and can be used to reset transient fields to an initial value if necessary. The `ejbPassivate()` method is called immediately prior to passivation of the bean instance. These two methods are especially helpful if the bean instance maintains connections to resources that need to be manipulated or freed prior to passivation and reobtained following activation. Because the stateful bean instance is evicted from memory, open connections to resources are not maintained. The exceptions are remote references to other beans and the `SessionContext`, which must be maintained with the serialized state of the bean and reconstructed when the bean is activated. EJB also requires that the references to the JNDI environment context, component interfaces, and the UserTransaction be maintained through passivation.

Entity beans do not have conversational state that needs to be serialized like stateful beans; instead, the state of entity bean instances is persisted directly to the database. Entity beans do, however, leverage the activation callback methods (`ejbActivate()` and `ejbPassivate()`) to notify the instance when it's about to be swapped in or out of the instance pool. The `ejbActivate()` method is invoked immediately after the bean instance is swapped into the EJB object, and the `ejbPassivate()` method is invoked just before the instance is swapped out.

Primary Services

There are many value-added services available for distributed applications. The OMG (the CORBA governing body), for example, has defined 13 services for use in CORBA-compliant ORBs. This book looks at seven value-added services that are called the *primary services*, because they are required to complete the Enterprise JavaBeans platform. The primary services include concurrency, transactions, persistence, distributed objects, asynchronous messaging (EJB 2.0), naming, and security.

The seven primary services are not new concepts; the OMG defined interfaces for these services specific to the CORBA platform some time ago. In most traditional CORBA ORBs, services are add-on subsystems that are explicitly utilized by the application code. This means that the server-side component developer has to write code to use primary service APIs right alongside their business logic. The use of primary services becomes complicated when they are used in combination with resource management techniques because the primary services are themselves complex. Using them in combination only compounds the problem.

As more complex component interactions are required, coordinating these services becomes a difficult task, requiring system-level expertise unrelated to the task of writing the application's business logic. Application developers can become so mired in the system-level concerns of coordinating various primary services and resource management mechanisms that their main responsibility, modeling the business, is all but forgotten.

EJB servers automatically manage all the primary services. This relieves the application developers from the task of mastering these complicated services. Instead, developers can focus on defining the business logic that describes the system, and leave the system-level concerns to the CTM. The following sections describe each of the primary services and explain how they are supported by EJB.

Concurrency

The issue of concurrency is important to all the bean types, but it has a different meaning when applied to EJB 2.0 message-driven beans than it does with the RMI based session and entity beans. This because of the difference in context: with RMI-based beans, concurrency refers to multiple clients accessing the same bean simultaneously; in message-driven beans, concurrency refers to the processing of multiple asynchronous messages simultaneously. For this reason we will discuss the importance of concurrency as primary services separately for these different types of beans.

Concurrency with Session and Entity beans

Session beans do not support concurrent access. This makes sense if you consider the nature of both stateful and stateless session beans. A stateful bean is an extension of one client and only serves that client. It doesn't make sense to make stateful beans concurrent if they are only used by the client that created them. Stateless session beans don't need to be concurrent because they don't maintain state that needs to be shared. The scope of the operations performed by a stateless bean is limited to the scope of each method invocation. No conversational state is maintained.

Entity beans represent data in the database that is shared and needs to be accessed concurrently. Entity beans are shared components. In Titan's EJB system, for example, there are only three ships: *Paradise*, *Utopia*, and *Valhalla*. At any given moment the Ship entity bean that represents the *Utopia* might be accessed by hundreds of clients. To make concurrent access to entity beans possible, EJB needs to protect the data represented by the shared bean, while allowing many clients to access the bean simultaneously.

In a distributed object system, problems arise when you attempt to share distributed objects among clients. If two clients are both using the same EJB object, how do you keep one client from writing over the changes of the other? If, for example, one client reads the state of an instance just before a different client makes a change to the same instance, the data that the first client read becomes invalid. Figure 3-9 shows two clients sharing the same EJB object.

[FIGURE]

Figure 3-5: Clients sharing access to an EJB object

EJB has addressed the dangers associated with concurrency in entity beans by implementing a simple solution: EJB, by default, prohibits concurrent access to bean instances. In other words, several clients can be connected to one EJB object, but only one client thread can access the bean instance at a time. If, for example, one of the clients invokes a method on the EJB object, no other client can access that bean instance until the method invocation is complete. In fact, if the method is part of a larger transaction, then the bean instance cannot be accessed at all, except within the same transactional context, until the entire transaction is complete.

Since EJB servers handle concurrency automatically, a bean's methods do not have to be made thread-safe. In fact, the EJB specification prohibits use of the `synchronized` keyword. Prohibiting the use of the thread synchronization primitives prevents developers from thinking that they control synchronization and enhances the performance of bean instances at runtime. In addition, EJB explicitly prohibits beans from creating their own threads. In other words, as a bean developer you cannot create a thread within a bean. The EJB container has to maintain complete control over the bean to properly manage concurrency, transactions, and persistence. Allowing the bean developer to create arbitrary

threads would compromise the container's ability to track what the bean is doing, and thus would make it impossible for the container to manage the primary services.

Reentrance

When talking about concurrency in entity beans, we need to discuss the related concept of reentrance. Reentrance is when a thread of control attempts to reenter a bean instance. In EJB, entity bean instances are nonreentrant by default, which means that loopbacks are not allowed. Before I explain what a loopback is, it is important that you understand a very fundamental concept in EJB: entity and session beans interact using each other's remote references and do not interact directly. In other words, when bean A operates on bean B, it does so the same way an application client would, by using B's remote or local interface as implemented by an EJB object. This allows the EJB container to interpose between method invocations from one bean to the next to apply security and transaction services.

While most bean-to-bean interactions in EJB 2.0 will take place using local interfaces of co-located enterprise beans, occasionally beans may interact using remote interfaces. Remote interfaces enforce complete location transparency. When interactions between beans take place using remote references, the beans can be relocated—possibly to a different server—with little or no impact on the rest of the application.

Regardless of whether remote or local interfaces are used, from the perspective of the bean servicing the call, all clients are created equal. Figure 3-11 shows that, from a bean's point of view, only clients perform business method invocations. When a bean instance has a business method invoked, it cannot tell the difference between a remote application client and a bean client.

[FIGURE modified version of figure 3-6]

Figure 3-6: Beans access each other through EJB objects

A loopback occurs when bean A invokes a method on bean B that then attempts to make a call back to bean A. Figure 3-13 shows this type of interaction. In Figure 3-13, client 1 invokes a method on bean A. In response to the method invocation, bean A invokes a method on bean B. At this point, there is no problem because client 1 controls access to bean A and bean A is the client of bean B. If, however, bean B attempts to call a method on bean A, it would be blocked because the thread has already entered bean A. By calling its caller, bean B is performing a loopback. This is illegal by default because EJB doesn't allow a thread of control to reenter a bean instance. To say that beans are non-reentrant by default is to say that loopbacks are not allowed.

[FIGURE]

Figure 3-7: A loopback scenario

The nonreentrance policy is applied differently to session beans and entity beans. Session beans can never be reentrant, and they throw a `RemoteException` if a loopback is attempted. The same is true of a nonreentrant entity bean. Entity beans can be configured in the deployment descriptor to allow reentrance at deployment time. Making an entity bean reentrant, however, is discouraged by the specification. The question of reentrancy is not relevant to EJB 2.0's message-driven beans because they do not respond to RMI calls like session and entity beans.

As discussed previously, client access to a bean is synchronized so that only one client can access any given bean at one time. Reentrance addresses a thread of control—initiated by a client request—that attempts to reaccess a bean instance. The problem with reentrant code is that the EJB object—which intercepts and delegates method invocations on the bean instance—cannot differentiate between reentrant code and multithreaded access within the same transactional context. (More about transactional context in Chapter 148.) If you permit reentrance, you also permit multithreaded access to the bean instance. Multithreaded access to a bean instance can result in corrupted data because threads impact each other's work trying to accomplish their separate tasks.

It's important to remember that reentrant code is different from a bean instance that simply invokes its own methods at an instance level. In other words, method `foo()` on a bean instance can invoke its own public, protected, default, or private methods directly as much as it wants. Here is an example of intra-instance method invocation that is perfectly legal:

```
public HypotheticalBean extends EntityBean {
    public int x;

    public double foo() {
        int i = this.getX();
        return this.boo(i);
    }
    public int getX() {
        return x;
    }
    private double boo(int i) {
        double value = i * Math.PI;
        return value;
    }
}
```

In the previous code fragment, the business method, `foo()`, invokes another business method, `getX()`, and then a private method, `boo()`. The method invocations made within the body of `foo()` are intra-instance invocations and are not considered reentrant.

EJB 2.0: Concurrency with Message-Driven Beans

When we are talking about concurrency in message-driven beans we are referring to the processing of more then one message at a time. As mentioned already in this chapter, concurrent processing of messages makes message-driven beans a powerful asynchronous component model. If message-driven beans could only process a single message at time, they would be practically useless in a real-world application because they couldn't handle heavy message loads.

Many JMS clients may be sending messages to the same destination. The ability to process all the messages by a single message-driven bean at the same time is concurrency. If five messages are delivered to a specific destination, then five instances of a message driven bean that subscribes or listens to that destination can be used to process the messages simultaneously. Figure 3-y illustrates.

[Figure 3-y]

Figure 3-y: Concurrent processing with Message-driven beans

In Figure 3-y, the same message-driven bean provides instances to process three messages from three different clients at the same time. This is concurrent processing.

There is actually a lot more to concurrent processing in message-driven beans. There are topic and queue type destinations and these are processed differently, but the basic value of concurrent processing is the same. The book will explore the details behind of the topic and queue type destinations in Chapter 13, *Message-Driven Beans.*

Transactions

Component transaction monitors (CTMs) were developed to bring the robust, scalable transactional integrity of traditional TP monitors to the dynamic world of distributed objects. Enterprise JavaBeans, as a server-side component model for CTMs, provides robust support for transactions for both all the bean types (session, entity and message-driven).

A transaction is a unit-of-work or a set of tasks that are executed together. Transactions are atomic; in other words, *all* the tasks in a transaction must be completed together to consider the transaction a success. In the previous chapter we used the TravelAgent bean to describe how a session bean controls the interactions of other beans. Here is a code snippet showing the `bookPassage()` method described in Chapter 2:

```
public Ticket bookPassage(CreditCard card, double price)
throws IncompleteConversationalState {
```



```
    // EJB 1.0: also throws RemoteException

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome = (ReservationHomeRemote)
            getHome("ReservationHome",ReservationHomeRemote.class);
        ReservationRemote reservation =
        resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            getHome("ProcessPaymentHome",ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        Ticket ticket = new Ticket(customer, cruise, cabin, price);
        return ticket;
    } catch(Exception e) {
        // EJB 1.0: throw new RemoteException("",e);
        throw new EJBException(e);
    }
}
```

The `bookPassage()` method consists of two tasks that must be completed together: the creation of a new Reservation bean and processing of the payment. When the TravelAgent bean is used to book a passenger, the charges to the passenger's credit card and the creation of the reservation must both be successful. It would be inappropriate for the ProcessPayment bean to charge the customer's credit card if the creation of a new Reservation bean fails. Likewise, you can't make a reservation if the customer credit card is not charged. An EJB server monitors the transaction to ensure that all the tasks are completed successfully.

Transactions are managed automatically, so as a bean developer you don't need to use any APIs to explicitly manage a bean's involvement in a transaction. Simply declaring the transactional attribute at deployment time tells the EJB server how to manage the bean at runtime. EJB does provide a mechanism that allows beans to manage transactions explicitly, if necessary. Setting the transactional attributes during deployment is discussed in Chapter 14~~8~~, as is explicit management of transactions and other transactional topics.

Persistence

Entity beans represent the behavior and data associated with real-world people, places, or things. Unlike session and message-driven type beans, entity beans are persistent. That means that the state of an entity is stored permanently in a database. This allows entities to be durable so that both their behavior and data

can be accessed at any time without concern that the information will be lost because of a system failure.

When a bean's state is automatically managed by a persistence service, the container is responsible for synchronizing an entity bean's instance fields with the data in the database. This automatic persistence is called *container-managed* persistence. When beans are designed to manage their own state, as is often the case when dealing with legacy systems, it is called *bean-managed* persistence.

Each vendor gets to choose the exact mechanism for implementing container-managed persistence, but the vendor's implementation must support the EJB callback methods and transactions. The most common mechanisms used in persistence by EJB vendors are *object-to-relational persistence* and *object database persistence*.

Object-to-relational persistence

Object-to-relational persistence is perhaps the most common persistence mechanism used in distributed object systems today. Object-to-relational persistence involves mapping entity bean state to relational database tables and columns.

In EJB 2.0 the abstract accessor methods represents the entity bean's container-managed fields, which we will just call fields. When an entity bean is deployed the container will implement these *virtual* fields for the bean, so its convient to think of the abstract accessor methods as describing persistent fields. For example, when we are talking about the state represented by the `setName()`/`getName()` abstract accessor method, we will refer to as the `name` field. Simularly, the `getId()`/`setId()` is the `id` field, and the `getDeckLevel()`/`setDeckLevel()` is the `deckLevel` field.

In Titan's system, for example, the `CabinBean` models the business concept of a ship's cabin. The CabinBean defined three fields: `String` type `name`, a `int` type `deckLevel`, and an `Integer` type `id`. The following code shows an abbreviated definition of the `CabinBean`:

EJB 2.0: CabinBean

```
public abstract class CabinBean implements javax.ejb.EntityBean {

    public abstract String getName();
    public abstract void setName(String str);

    public abstract int getDeckLevel();
    public abstract void setDeckLevel(int level);

    public abstract Integer getId( );
    public abstract void setId(Integer id);
```



```
}
```

EJB 1.1: CabinBean

```
public class CabinBean implements javax.ejb.EntityBean {

    public int id;
    public String name;
    public int deckLevel;

}
```

With object-to-relational database mapping, the fields of an entity bean correspond to columns in a relational database. The Cabin's `name field`, for example, maps to the column labeled `NAME` in a table called `CABIN` in Titan's relational database. Figure 3-15 shows a graphical depiction of this type of mapping.

[FIGURE 3-8 modified]

Figure 3-8: Object-to-relational mapping of entity beans

Really good EJB systems provide wizards or administrative interfaces for mapping relational database tables to the fields of entity bean classes. Using these wizards, mapping entities to tables is a fairly straightforward process and is usually performed at deployment time. Figure 3-17 shows WebLogic's object-to-relational mapping wizard.

[FIGURE]

Figure 3-9: Object-to-relational mapping wizard

Once a bean's fields are mapped to the relational database, the container takes over the responsibility of keeping the state of an entity bean instance consistent with the corresponding tables in the database. This process is called *synchronizing* the state of the bean instance. In the case of `CabinBean`, bean instances at runtime will map one-to-one to rows in the `CABIN` table of the relational database. When a change is made to a Cabin EJB, it is written to the appropriate row in the database. Frequently, bean types will map to more than one table. These are more complicated mappings, often requiring an SQL join. Good EJB deployment tools should provide wizards that make multitable mappings fairly easy.

In addition, EJB 2.0 container-managed persistence defines entity bean relationships fields, which allow entity beans to have one-to-one, one-to-many, and many-to-many relationships with other beans. Entity beans can maintain collections of other entity beans or single references. The persistence of entity beans in EJB 2.0 is a great deal more complex and powerful then was supported in previous versions of the specification. The new EJB 2.0 container-managed persistence model is covered in Chapters 6, 7 and 8.

In addition to synchronizing the state of an entity, EJB provides mechanisms for creating and removing entities. Calls to the EJB home to create and remove entities will result in a corresponding insertion or deletion of records in the database. Because entities store their state in database tables, new records (and therefore bean identities) can be added to tables from outside the EJB system. In other words, inserting a record into the `CABIN` table— whether done by EJB or by direct access to the database—creates a new Cabin entity. It's not created in the sense of instantiating a Java object, but in the sense that the data that describes a Cabin entity has been added to the system.

Object database persistence

Object-oriented databases are designed to preserve object types and object graphs and therefore are a much better match for components written in an object-oriented language like Java. They offer a cleaner mapping between entity beans and the database than a traditional relational database. However, this is more of an advantage in EJB 1.1 than it is in EJB 2.0. EJB 2.0 container-managed persistence provides a programming model that is expressive enough to accommodate both object-to-relational mapping as well as object databases.

While object databases perform well when it comes to very complex object graphs, they are still fairly new to business systems and are not as widely accepted as relational databases. As a result, they are not as standardized as relational databases, making it more difficult to migrate from one database to another. In addition, fewer third-party products exist that support object databases, like products for reporting and data warehousing.

Several relational databases support extended features for native object persistence. These databases allow some objects to be preserved in relational database tables like other data types and offer some advantages over other databases.

Legacy persistence

EJB is often used to put an object wrapper on legacy systems, systems that are based on mainframe applications or nonrelational databases. Container-managed persistence in such an environment requires a special EJB container designed specifically for legacy data access. Vendors might, for example, provide mapping tools that allow beans to be mapped to IMS, CICS, b-trieve, or some other legacy application.

Regardless of the type of legacy system used, container-managed persistence is preferable to bean-managed persistence. With container-managed persistence, the bean's state is managed automatically, which is more efficient at runtime and more productive during bean development. Many projects, however, require that beans obtain their state from legacy systems that are not supported by the EJB vendor. In these cases, developers must use bean-managed persistence, which

means that the developer doesn't use the automatic persistence service of the EJB server. Chapters 6-11 describes ~~both~~ container-managed and bean-managed persistence in detail.

Distributed Objects

Three main distributed object services are available today: CORBA IIOP, Java RMI, and Microsoft's .NET. Each of these platforms uses a different RMI network protocol, but they all accomplish basically the same thing: location transparency. Microsoft's .NET platform, which relies on DCOM, is used in the Microsoft Windows environment and is not supported by other operating systems. Its tight integration with Microsoft products makes it a good choice for Microsoft-only systems. This may change with the growing support for SOAP (Simple Object Access Protocol), an XML-based protocol that is quickly becoming popular and offers interoperability with non-Microsoft applications. CORBA IIOP is neither operating-system specific nor language specific and has been traditionally been considered the most open distributed object service of the three. It's an ideal choice when integrating systems developed in multiple programming languages. Java RMI is a Java language abstraction or programming model for any kind of distributed object protocol. In the same way that the JDBC API can be used to access any SQL relational database, Java RMI is intended to be used with almost any distributed object protocol. In practice, Java RMI has traditionally been limited to the Java Remote Method Protocol (JRMP)—known as Java RMI over JRMP—which can only be used between Java applications. Recently an implementation of Java RMI over IIOP (Java RMI-IIOP), the CORBA protocol, has been developed. Java RMI-IIOP is a CORBA-compliant version of Java RMI, which allows developers to leverage the simplicity of the Java RMI programming model, while taking advantage of the platform- and language-independent CORBA protocol, IIOP.[2]

When we discuss the component interfaces, and other EJB interfaces and classes used on the client, we are talking about the client's view of the EJB system. The *EJB client view* doesn't include the EJB objects, the EJB container, instance swapping, or any of the other implementation specifics. As far as a remote client is concerned, a bean is defined by its remote interface and home interface. Everything else is invisible. As long as the EJB server supports the EJB client view, any distributed object protocol can be used. However, EJB 2.0 requires that every EJB server support Java RMI-IIOP—but it doesn't limit the protocols a EJB server can support to Java RMI-IIOP.

Regardless of the protocol used, the server must support Java clients using the Java EJB client API, which means that the protocol must map to the Java RMI-

2 Java RMI-IIOP is interoperable with CORBA ORBs that support the CORBA 2.3.1 specification. ORBs that support an older specification cannot be used with Java RMI-IIOP because they do not implement the Object by Value portion of the 2.3.1 specification.

IIOP programming model. Using Java RMI over DCOM seems a little far-fetched, but Java RMI over SOAP is possible. Figure 3-19 illustrates the Java language EJB API supported by different distributed object protocols.

[FIGURE modified 3-10]

Figure 3-10: Java EJB client view supported by various protocols

EJB also allows servers to support access to beans by clients written in languages other than Java. An example of this is the EJB-to-CORBA mapping defined by Sun.[3] This document describes the CORBA IDL (Interface Definition Language) that can be used to access enterprise beans from CORBA clients. A CORBA client can be written in any language, including C++, Smalltalk, Ada, and even COBOL. The mapping also includes details about supporting the Java EJB client view as well as details on mapping the CORBA naming system to EJB servers and distributed transactions across CORBA objects and beans. Eventually, a EJB-to-SOAP mapping may be defined that will allow SOAP client applications written in languages like Visual Basic, Delphi, PowerBuilder, and others to access beans. Figure 3-11 illustrates the possibilities for accessing an EJB server from different distributed object clients.

[FIGURE]

Figure 3-11: EJB accessed from different distributed clients

As a mature, platform-independent and language-independent distributed object protocol, CORBA is currently regarded by many as the superior of the three protocols discussed here. For all its advantages, however, CORBA suffers from some limitations. Pass-by-value, a feature easily supported by Java RMI-IIOP, was only recently introduced in the CORBA 2.3 specification and is not well supported. Another limitation of CORBA is with casting remote proxies. In Java RMI-JRMP, you can cast or widen a proxy's remote interface to a subtype or supertype of the interface, just like any other object. This is a powerful feature that allows remote objects to be polymorphic. In Java RMI-IIOP, you have to call a special narrowing method to change the interface of a proxy to a subtype, which is cumbersome.

However, JRMP is has its own limitations. While JRMP may be a more natural fit for Java-to-Java distributed object systems, it lacks inherent support for both security and transactional services—support that is a part of the CORBA IIOP specification. This limits the effectiveness of JRMP in heterogeneous environments where security and transactional contexts must be passed between systems.

---

3 Sun Microsystems' *Enterprise JavaBeans™ to CORBA Mapping, Version 1.1*, by Sanjeev Krishnan, Copyright 1999 by Sun Microsystems.

EJB 2.0: Asynchronous Enterprise Messaging

In past versions of Enterprise JavaBeans, support for asynchronous enterprise messaging and specifically the Java Message Service was not considered a primary service because it wasn't necessary in order to have a complete Enterprise JavaBeans platform. However, with the introduction of message-driven beans to Enterprise JavaBeans, asynchronous enterprise messaging has become so important that its status must be elevated to a primary service.

Support for this service is complex, but basically it requires that the EJB container system reliably route messages from JMS clients to message-driven beans. This involves more than the simple delivery semantics you associate with e-mail or even the JMS API. With enterprise messaging, messages must be reliably delivered which means that a failure to deliver the message should require the EJB container system to attempt redelivery. What's more, enterprise messages may be persistent, which means they are stored to disk or a database until it can be properly delivered to its intended clients. Persistent messages must survive system failures, if the EJB server crashes the persistent messages must still be available for delivery when the server comes back up.

Most importantly, enterprise messaging is transactional messaging. That means if for any reason a message-driven bean fails while processing a message, that failure will abort the transaction and force the EJB container to redeliver the message to another message-driven bean instance.

In addition to message-driven beans, any stateless, entity, or message-driven bean can also send JMS messages. Support for sending messages is not as critical in Enterprise JavaBeans as delivery of messages to message-driven beans, but support for these facilities tends to go hand in hand. In other words, its unlikely that an EJB server would go to the trouble of supporting the consumption of JMS messages by message-driven beans without also supporting the sending of messages by all different types of enterprise beans.

It's interesting to note that the semantics of supporting message-driven beans requires light coupling between the EJB container system and the JMS message router, so that many EJB container systems will support a limited number of JMS providers. This means that message-driven beans can't consume messages from any arbitrary JMS provider or MOM product. Only the JMS providers supported explicitly by the EJB vendor will be able to deliver messages to message-driven beans.

Naming

All distributed object services use a naming service of some kind. Java RMI-JRMP and CORBA use their own naming services. All naming services do

essentially the same thing regardless of how they are implemented: they provide clients with a mechanism for locating distributed objects or resources.

To accomplish this, a naming service must provide two things: object binding and a lookup API. *Object binding* is the association of a distributed object with a natural language name or identifier. The `CabinHomeRemote` object, for example, might be bound to the name "cabin.Home" or "room." A binding is really a pointer or an index to a specific distributed object, which is necessary in an environment that manages hundreds of different distributed objects. A *lookup API* provides the client with an interface to the naming system. Simply put, lookup APIs allow clients to connect to a distributed service and request a remote reference to a specific object.

Enterprise JavaBeans mandates the use of the Java Naming and Directory Interface (JNDI) as a lookup API on Java clients. JNDI supports just about any kind of naming and directory service. A directory service is a very advanced naming service that organizes distributed objects and other resources—printers, files, application servers, etc.—into hierarchical structures and provides more sophisticated management features. With directory services, metadata about distributed objects and other resources are also available to clients. The metadata provides attributes that describe the object or resource and can be used to perform searches. You can, for example, search for all the laser printers that support color printing in a particular building.

Directory services also allow resources to be linked virtually, which means that a resource can be located anywhere you choose in the directory services hierarchy. JNDI allows different types of directory services to be linked together so that a client can move between different types of services seamlessly. It's possible, for example, for a client to follow a directory link in a Novell NetWare directory into an EJB server, allowing the server to be integrated more tightly with other resources of the organization it serves.

There are many different kinds of directory and naming services; EJB vendors can choose the one that best meets their needs, but all EJB 2.0 platforms must support the CORBA Naming service in addition to any other directory service they choose to support.

A Java client application would use JNDI to initiate a connection to an EJB server and to locate a specific EJB home. The following code shows how the JNDI API might be used to locate and obtain a reference to the EJB home `CabinHome`:

```
javax.naming.Context jndiContext =
    new javax.naming.InitialContext(properties);
Object ref = jndiContext.lookup("cabin.Home");
CabinHome cabinHome = (CabinHome)
    PortableRemoteObject.narrow(ref, CabinHome.class);
```



```
Cabin cabin = cabinHome.create(382, "Cabin 333",3);
cabin.setName("Cabin 444");
cabin.setDeckLevel(4);
```

The properties passed into the constructor of `InitialContext` tell the JNDI API where to find the EJB server and what JNDI service provider (driver) to load. The `Context.lookup()` method tells the JNDI service provider the name of the object to return from the EJB server. In this case, we are looking for the home interface to the Cabin EJB. Once we have the Cabin EJB's home interface, we can use it to create new cabins and access existing cabins.

Enterprise JavaBeans requires the use of the `PortableRemoteObject.narrow()` method to cast remote references obtained from JNDI into the `CabinHomeRemote` interface type. This is addressed in more detail in Chapters 4 and 5 and is not essential to the content covered here – the use of this facility is not required when enterprise beans use the local component interfaces of other co-located enterprise beans.

Security

Enterprise JavaBeans servers might support as many as three kinds of security: authentication, access control, and secure communication. Only access control is specifically addressed by Enterprise JavaBeans.

Authentication

Simply put, authentication validates the identity of the user. The most common kind of authentication is a simple login screen that requires a username and a password. Once users have successfully passed through the authentication system, they are free to use the system. Authentication can also be based on secure ID cards, swipe cards, security certificates, and other forms of identification. While authentication is the primary safeguard against unauthorized access to a system, it is fairly crude because it doesn't police an authorized user's access to resources within the system.

Access control

Access control (a.k.a. authorization) applies security policies that regulate what a specific user can and cannot do within a system. Access control ensures that users only access resources for which they have been given permission. Access control can police a user's access to subsystems, data, and business objects, or it can monitor more general behavior. Certain users, for example, may be allowed to update information while others are only allowed to view the data.

Secure communication

Communication channels between a client and a server are frequently the focus of security concerns. A channel of communication can be secured by physical isolation (like a dedicated network connection) or by encrypting the communication between the client and the server. Physically securing

communication is expensive, limiting, and pretty much impossible on the Internet, so we will focus on encryption. When communication is secured by encryption, the messages passed are encoded so that they cannot be read or manipulated by unauthorized individuals. This normally involves the exchange of cryptographic keys between the client and the server. The keys allow the receiver of the message to decode the message and read it.

Most EJB servers support secure communications—usually through SSL (secure socket layer)—and some mechanism for authentication, but Enterprise JavaBeans only specifies access control in their server-side component models. Authentication may be specified in subsequent versions, but secure communications will probably never be specified. Secure communications is really independent of the EJB specification and the distributed object protocol.

Although authentication is not specified in EJB, it is often accomplished using the JNDI API. In other words, a client using JNDI can provide authenticating information using the JNDI API to access a server or resources in the server. This information is frequently passed when the client attempts to initiate a JNDI connection to the EJB server. The following code shows how the client's password and username are added to the connection properties used to obtain a JNDI connection to the EJB server:

```
properties.put(Context.SECURITY_PRINCIPAL, userName );
properties.put(Context.SECURITY_CREDENTIALS, userPassword);

javax.naming.Context jndiContext =
    new javax.naming.InitialContext(properties);
Object ref= jndiContext.lookup("titan.CabinHome");
CabinHome cabinHome = (CabinHome)
    PortableRemoteObject.narrow(ref, CabinHome.class);
```

EJB specifies that every client application accessing an EJB system must be associated with a security identity. The security identity represents the client as either a user or a role. A user might be a person, security credential, computer, or even a smart card. Normally, the user will be a person whose identity is assigned when he or she logs in. A role represents a grouping of identities and might be something like "manager," which is a group of user identities that are considered managers at a company.

When a remote client logs on to the EJB system, it is associated with a security identity for the duration of that session. The identity is found in a database or directory specific to the platform or EJB server. This database or directory is responsible for storing individual security identities and their memberships to groups.

Once a remote client application has been associated with a security identity, it is ready to use beans to accomplish some task. The EJB server keeps track of each client and its identity. When a client invokes a method on a component interface, the EJB server implicitly passes the client's identity with the method invocation.

When the EJB object or EJB home receives the method invocation, it checks the identity to ensure that the client is allowed to invoke that method.

Role-driven access control

In Enterprise JavaBeans, the security identity is represented by a `java.security.Principle` object. As a security identity, the `Principle` acts as a representative for users, groups, organizations, smart cards, etc., to the EJB access control architecture. Deployment descriptors include tags that declare which logical roles are allowed to access which bean methods at runtime. The security roles are considered logical roles because they do not *directly* reflect users, groups, or any other security identities in a specific operational environment. Instead, security roles are mapped to real-world user groups and users when the bean is deployed. This allows a bean to be portable; every time the bean is deployed in a new system the roles can be mapped to the users and groups specific to that operational environment. Here is a portion of the Cabin EJB's deployment descriptor that defines two security roles, `ReadOnly` and `Administrator`:

```
<security-role>
    <description>
        This role is allowed to execute any method on the bean.
        They are allowed to read and change any cabin bean data.
    </description>
    <role-name>
        Administrator
    </role-name>
</security-role>

<security-role>
    <description>
        This role is allowed to locate and read cabin info.
        This role is not allowed to change cabin bean data.
    </description>
    <role-name>
        ReadOnly
    </role-name>
</security-role>
```

The role names in this descriptor are not reserved or special names, with some sort of predefined meaning; they are simply logical names chosen by the bean assembler. In other words, the role names can be anything you want as long as they are descriptive.[4]

4 For a complete understanding of XML, including specific rules for tag names and data, see *XML Pocket Reference*, by Robert Eckstein (O'Reilly).

How are roles mapped into actions that are allowed or forbidden? Once the `security-role` tags are declared, they can be associated with methods in the bean using `method-permission` tags. Each `method-permission` tag contains one or more `method` tags, which identify the bean methods associated with one or more logical roles identified by the `role-name` tags. The `role-name` tags must match the names defined by the security-role tags shown earlier.

```
<method-permission>
    <role-name>Administrator</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>*</method-name>
    </method>
</method-permission>
</method-permission>
    <role-name>ReadOnly</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>getName</method-name>
    </method>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>getDeckLevel</method-name>
    </method>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>findByPrimaryKey</method-name>
    </method>
</method-permission>
```

In the first `method-permission`, the `Administrator` role is associated with all methods on the Cabin EJB, which is denoted by specifying the wildcard character (`*`) in the `method-name` of the `method` tag. In the second `method-permission` the `ReadOnly` role is limited to accessing only three methods: `getName()`, `getDeckLevel()`, and `findByPrimaryKey()`. Any attempt by a `ReadOnly` role to access a method that is not listed in the `method-permission` will result in an exception. This kind of access control makes for a fairly fine-grained authorization system.

Since an XML deployment descriptor can be used to describe more than one enterprise bean, the tags used to declare method permissions and security roles are defined in a special section of the deployment descriptor, so that several beans can share the same security roles. The exact location of these tags and their relationship to other sections of the XML deployment descriptor will be covered in more detail in Chapter 16.

When the bean is deployed, the person deploying the bean will examine the `security-role` information and map each logical role to a corresponding

user group in the operational environment. The deployer need not be concerned with what roles go to which methods; he can rely on the descriptions given in the `security-role` tags to determine matches based on the description of the logical role. This unburdens the deployer, who may not be a developer, from having to understand how the bean works in order to deploy it.

Figure 3-23 shows the same enterprise bean deployed in two different environments (labeled X and Z). In each environment, the user groups in the operational environment are mapped to their logical equivalent roles in the XML deployment descriptor so that specific user groups have access privileges to specific methods on specific enterprise beans.

[FIGURE]

Figure 3-12: Mapping roles in the operational environment to logical roles in the deployment descriptor

As you can see from the figure, the `ReadOnly` role is mapped to those groups that should be limited to the get accessor methods and the find method. The `Administrator` role is mapped to those user groups that should have privileges to invoke any method on the Cabin EJB.

The access control described here is implicit; once the bean is deployed the container takes care of checking that users only access methods for which they have permission. This is accomplished by propagating the security identity, the `Principle`, with each method invocation from the client to the bean. When a client invokes a method on a bean, the client's `Principle` is checked to see if it is a member of a role mapped to that method. If it's not, an exception is thrown and the client is denied permission to invoke the method. If the client is a member of a privileged role, the invocation is allowed to go forward and the method is invoked.

If a bean attempts to access any other enterprise beans while servicing a client, it will pass along the client's security identity for access control checks by the other beans. In this way, a client's `Principle` is propagated from one bean invocation to the next, ensuring that a client's access is controlled whether or not it invokes a bean method directly. In EJB 2.0 this propagation can be overridden by specifying that the enterprise bean executes under a different security identity called the *runAs* security identity.

EJB 2.0: The runAs Security Identity

In addition to specifying the `Principal`s that have access to an enterprise bean's methods, the deployer can also specify the *runAs* `Principal` for the entire enterprise bean. The runAs security identity was originally specified in EJB 1.0, but was abandoned in EJB 1.1. It has been reintroduced in EJB 2.0 and modified so that its is easier for vendors to implement.

While the `method-permission` elements specify which `Principal`s have access to the bean's methods, the `security-identity` element specifies under which `Principal` the method will run. In other words, the runAs `Principal` is used as the enterprise bean's identity when it tries to invoke methods on other beans—this identity isn't necessarily the same as the identity that's currently accessing the bean.

For example, the following deployment descriptor elements declare that the `create()` method can only be accessed by "JimSmith", but that Cabin EJB always runs under an "Administrator" `Principal` role.

```
<enterprise-beans>
...
    <entity>
        <ejb-name>EmployeeService</ejb-name>
        ...
        <security-identity>
            <run-as>
               <role-name>Administrator</role-name>
            </run-as>
        </security-identity>
        ...
    </entity>
...
</enterprise-beans>
<assembler>
<security-role>
    <role-name>Administrator</role-name>
</security-role>
<security-role>
    <role-name>JimSmith</role-name>
</security-role>
...
<method-permission>
    <role-name>JimSmith</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>create</method-name>
    </method>
</method-permission>
...
</assembler>
```

This is kind of configuration is useful when the enterprise beans or resources accessed in the body of the method require a `Principal` that is different from the one used to gain access to the method. The `create()` method might call a method in enterprise bean X that requires the Administrator's `Principal`. If we want to use enterprise bean X in the `create()` method, but we only want Jim Smith to create new cabins, we would use the `security-identity` and

`method-permission` elements together to give us this kind of flexibility: the `method-permission` for `create()` would specify that only Jim Smith can invoke the method, and the `security-identity` element would specify that the enterprise bean always runs under the Administrator's `Principal`.

In order to specify that an enterprise bean execute under the caller's identity, the security-identity role contains a single empty element, the `use-caller-identity` element. For example, the following declarations specify that the Cabin EJB always execute under the callers identity, so if JimSmith invokes the `create()` method, the bean will run under the JimSmith security identity.

```
<enterprise-beans>
...
    <entity>
        <ejb-name>EmployeeService</ejb-name>
        ...
        <security-identity>
            <use-caller-identity/>
        </security-identity>
        ...
    </entity>
...
</enterprise-beans>
```

Figure 3-27 illustrates how the runAs `Principal` can change in a chain of method invocations. Notice that the runAs `Principal` is the `Principal` used to test for access in subsequent method invocations.

[FIGURE modified figure 3-14]

Figure 3-14: runAs Identity

1. The client, who is identified as "Bill Jones", invokes the method `foo()` on enterprise bean A.
2. Before servicing the method, enterprise bean A is checked to see if "Bill Jones" is included in the method-permissions for `foo()`. It is.
3. The security-identity of enterprise bean A is declared as `use-caller-identity`, so the `foo()` method executes under the caller's `Principal`, in this case "Bill Jones".
4. While `foo()` is executing, it invokes method `bar()` on enterprise bean B using the "Bill Jones" security identity.
5. Enterprise bean B checks method `foo()`'s `Principal` ("Bill Jones") against the allowed identities for method `bar()`. "Bill Jones" is included in the method-permissions, so the method `bar()` of enterprise bean B is allowed to execute.

6. The enterprise bean B specifies the security-identity to be the run-as `Principal` of "Administrator".

7. While `bar()` is executing, enterprise bean B invokes the method `boo()` on enterprise bean C.

8. Enterprise bean C is checked and it's determined that `bar()`'s runAs `Principal` ("Administrator") is included in the method-permissions for method `boo()`.

9. The security-identity for the enterprise bean C specifies a runAs `Principal` of the "System", which is the identity that the `boo()` method executes under.

This protocol applies equally to entity and stateless session beans. However, message-driven beans only have a runAs identity, the will never execute under the caller identity, because there is no "caller". Message-driven beans process asynchronous JMS messages. These messages are not considered "calls" and the JMS client that sent them is not associated with the message. Once a message is sent by a JMS client, is autonomous and is no longer associated with the sending client. So incoming messages do not have a "caller". With no caller security identity to propagate, message-driven beans must always have a runAs security identity specified, and it will always execute under that runAs `Principal`.

What's Next?

The first three chapters gave you a foundation on which to develop Enterprise JavaBeans components and applications. You should have a better understanding of CTMs and the EJB component model.

Beginning with Chapter 4, you will develop your own beans and learn how to apply them in EJB applications.

4

Developing Your First Enterprise Beans

Choosing and Setting Up an EJB Server

One of the most important features of EJB is that enterprise beans should work with containers from different vendors. That doesn't mean that selecting a server and installing your enterprise beans on that server are trivial processes.[1]

The EJB server you choose should be compliant with the EJB 2.0 specification. The first example in this chapter—and most of the examples in this book—assumes that your EJB server supports entity beans and EJB 2.0 container-managed persistence.[2] The EJB server you choose should also provide a utility for deploying an enterprise bean. It doesn't matter whether the utility is

---

[1] To help you work with different vendor's products, free workbooks have been created for specific EJB servers. Each workbook shows you how to download, install, and run the examples in this book for a specific product. We are trying to create a library that covers as many major vendors as possible, though with over 30 EJB servers on the market, we won't be able to cover all of them. The workbook examples cover EJB 2.0, unless the product supports only EJB 1.1. The workbooks are available in PDF form from *http://www.oreilly.com/catalog/entjbeans3/* or *http://www.monson-haefel.com*. If there is sufficient demand, we may make the workbooks available in a printed version.

2 Chapter 11 discusses EJB 1.1 container-managed persistence, which you can use if your server doesn't support EJB 2.0 container-managed persistence.

command-line oriented or graphical, as long as it does the job. The deployment utility should allow you to work with prepackaged enterprise beans, i.e., enterprise beans that have already been developed and archived in a JAR file. Finally, the EJB server should support an SQL-standard relational database that is accessible using JDBC. For the database, you should have privileges sufficient for creating and modifying a few simple tables in addition to normal read, update, and delete capabilities. If you have chosen an EJB server that does not support an SQL standard relational database, you may need to modify the examples to work with the product you are using.

This book does not say very much about how to install and deploy enterprise beans. That task is largely server-dependent. We give some general ideas about how to organize JAR files and create deployment descriptors, but for a complete description of the deployment process, you'll have to refer to your vendor's documentation, or look at the workbook for your vendor (if one is available).

This Chapter provides you with your first opportunity to use a workbook. Throughout the rest of this book you will see these *callouts* which direct you to an exercise in the workbook. A callout will look something like the following.

> 🕮 Exercise 4.2, Develop and Deploy the TravelAgent EJB

As was mentioned in the Preface, the workbooks can be downloaded in PDF format for free from *http://www.oreilly.com/catalog/entjbeans3/* or *http://www.monson-haefel.com* – some workbooks may even be available in paper book form and can be ordered direct from the http://www.monson-haefel.com.Setting Up Your Java IDE

To get the most from this chapter, it helps to have an IDE that has a debugger and allows you to add Java files to its environment. Several Java IDEs, like Symantec's Visual Cafe, IBM's VisualAge, Inprise's JBuilder, and Sun's Forte, fulfill this simple requirement. Some EJB products, like IBM's WebSphere, are tightly coupled with an IDE that makes life a lot easier when it comes to writing, deploying and debugging your applications.

Once you have an IDE set up, you need to include the Enterprise JavaBeans package, `javax.ejb`. You also need the JNDI packages, including `javax.naming`, `javax.naming.directory`, and `javax.naming.spi`. In addition, you will need the `javax.rmi` and `javax.jms` packages. All these packages can be downloaded from Sun's Java site (*http://www. javasoft.com*) in the form of ZIP or JAR files. They may also be accessible in the subdirectories of your EJB server, normally under the *lib* directory.

Developing an Entity Bean

There seems to be no better place to start than the Cabin EJB, which we have been examining throughout the previous chapters. The Cabin EJB is an entity bean that encapsulates the data and behavior associated with a cruise ship cabin in Titan's business domain.

Cabin: The Remote Interface

When developing an entity bean, we first want to define the enterprise bean's remote interface. The remote interface defines the enterprise bean's business purpose; the methods of this interface must capture the concept of the entity. We defined the remote interface for the Cabin EJB in Chapter 2; here, we add two new methods for setting and getting the ship ID and the bed count. The ship ID identifies the ship that the cabin belongs to, and the bed count tells how many people the cabin can accommodate.

```
package com.titan.cabin;

import java.rmi.RemoteException;

public interface CabinRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String str) throws RemoteException;
    public int getDeckLevel() throws RemoteException;
    public void setDeckLevel(int level) throws RemoteException;
    public int getShipId() throws RemoteException;
    public void setShipId(int sp) throws RemoteException;
    public int getBedCount() throws RemoteException;
    public void setBedCount(int bc) throws RemoteException;
}
```

The `CabinRemote` interface defines four properties: the `name`, `deckLevel`, `ship`, and `bedCount`. *Properties* are attributes of an enterprise bean that can be accessed by public set and get methods. The methods that access these properties are not explicitly defined in the `CabinRemote` interface, but the interface clearly specifies that these attributes are readable and changeable by a client.

Notice that we have made the `CabinRemote` interface a part of a new package named `com.titan.cabin`. Place all the classes and interfaces associated with each type of bean in a package specific to the bean.[3] Because our beans are

[3] The examples, which can be downloaded from *www.oreilly.com*, provide a good guide for how to organize your code; the code is organized in a directory structure that's typical

for the use of the Titan cruise line, we place these packages in the `com.titan` package hierarchy. We also create directory structures that match package structures. If you are using an IDE that works directly with Java files, create a new directory somewhere called *dev* (for development) and create the directory structure shown in Figure 4-1. Copy the `CabinRemote` interface into your IDE and save its definition to the *cabin* directory. Compile the `CabinRemote` interface to ensure that its definition is correct. The *CabinRemote.class* file, generated by the IDE's compiler, should be written to the *cabin* directory, the same directory as the *CabinRemote. java* file. The rest of the Cabin bean's classes will be placed in this same directory.

[FIGURE]

Figure 4-1: Directory structure for the Cabin bean

CabinHome: The Home Interface

Once we have defined the remote interface of the Cabin EJB, we have defined the remote view of this simple entity bean. Next, we need to define the Cabin EJB's remote home interface, which specifies how the enterprise bean can be created, located, and destroyed by remote clients; in other words, the Cabin EJB's life-cycle behavior. Here is a complete definition of the `CabinHomeRemote` home interface:

```
package com.titan.cabin;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CabinHomeRemote extends javax.ejb.EJBHome {

    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;

    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

The `CabinHomeRemote` interface extends the `javax.ejb.EJBHome` and defines two life- cycle methods: `create()` and `findByPrimaryKey()`. These methods create and locate remote references to Cabin EJBs. Remove methods (for deleting enterprise beans) are defined in the `javax.ejb.EJBHome` interface, so the `CabinHomeRemote` interface inherits them.

---

for most products. The workbooks provide additional help for organizing your development projects, and will point out any vendor-specific requirements.

CabinBean: The Bean Class

You have now defined the complete client-side API for creating, locating, removing, and using the Cabin EJB. Now we need to define `CabinBean`, the class that provides the implementation on the server for the Cabin EJB. The `CabinBean` class is an entity bean that uses container-managed persistence, so its definition will be fairly simple.

In addition to the callback methods discussed in Chapters 2 and 3, we must also define abstract accessor methods for the methods defined in the `CabinRemote` interface and an implementation of the create method defined in the `CabinHomeRemote` interface.

EJB 2.0: The Cabin Bean

Here is the complete definition of the `CabinBean` class:

```
package com.titan.cabin;

public abstract class CabinBean
implements javax.ejb.EntityBean {

   public Integer ejbCreate(Integer id){
      this.setId(id);
   }
   public void ejbPostCreate(String name){

   }
   public abstract void setId(Integer id);
   public abstract Integer getId();

   public abstract void setShipId(int ship);
   public abstract int getShipId( );

   public abstract void setName(String name);
   public abstract String getName( );

   public abstract void setBedCount(int count);
   public abstract int getBedCount( );

   public abstract void setDeckLevel(int level);
   public abstract int getDeckLevel( );

    public void setEntityContext(EntityContext ctx) {
         // Not implemented.
    }
    public void unsetEntityContext() {
         // Not implemented.
    }
```

```
    public void ejbActivate() {
        // Not implemented.
    }
    public void ejbPassivate() {
        // Not implemented.
    }
    public void ejbLoad() {
        // Not implemented.
    }
    public void ejbStore() {
        // Not implemented.
    }
    public void ejbRemove() {
        // Not implemented.
    }
}
```

The `CabinBean` class can be divided into four sections for discussion: declarations for the container-managed fields, the `ejbCreate()` methods, the callback methods, and the remote interface implementations.

The `CabinBean` defines several abstract accessor methods that appear in pairs. For example, the abstract methods `setName()` and `getName()` are a pair of abstract accessor methods. These methods will be responsible for setting and getting the entity bean's name field. When the bean is deployed, the EJB container automatically implements all the abstract accessor methods so that the bean state can be synchronized with the database. These implementations map the abstract accessor methods to fields in the database. Although all the abstract accessor methods have corresponding methods in the remote interface, `CabinRemote`, it's not necessary that they do so. Some accessor methods are for the entity bean's use only and are never exposed to the client through the remote or local interfaces.

It's customary in EJB 2.0 to consider the abstract accessor methods as providing access to virtual fields and to refer to those fields by their method name, less the get or set prefix. For example, the `getName()/setName()` abstract accessor methods define a virtual container-managed persistence field called `name` – the first letter is always changed to lower case. The `getDeckLevel()`/`setDeckLevel()` abstract accessor methods define a virtual container-managed persistence field called `deckLevel`, and so on.

The `name`, `deckLevel`, `ship`, and `bedCount` virtual container-managed persistence fields represent the Cabin EJB's persistent state. They will be mapped to the database at deployment time. These fields are also publicly available through the entity bean's remote interface. Invoking the `getBedCount()` method on a `CabinRemote` EJB object at runtime causes the container to delegate that call to the corresponding `getBedCount()` method on the `CabinBean` instance. The abstract accessor methods do not

throw the `RemoteException` like the matching methods in the remote interface.

There is no requirement that CMP fields must be exposed. The `id` field is another container-managed field, but its abstract accessor methods are not exposed to the client through the `CabinRemote` interface. This field is the primary key of the Cabin EJB; it's the entity bean's index to its data in the database. It's bad practice to expose the primary key of an entity bean so that it can be modified by a client. You don't want client applications changing that index.

EJB 1.1: The Bean Class

Here is the complete definition of the `CabinBean` class in EJB 1.1:

```
package com.titan.cabin;

import javax.ejb.EntityContext;

public class CabinBean implements javax.ejb.EntityBean {

    public Integer id;
    public String name;
    public int deckLevel;
    public int shipId;
    public int bedCount;

    public Integer ejbCreate(Integer id) {
        this.id = id;
        return null;
    }
    public void ejbPostCreate(Integer id) {
        // Do nothing. Required.
    }
    public String getName() {
        return name;
    }
    public void setName(String str) {
        name = str;
    }
    public int getShipId() {
        return shipId;
    }
    public void setShipId(int sp) {
        shipId = sp;
    }
    public int getBedCount() {
        return bedCount;
    }
    public void setBedCount(int bc) {
        bedCount = bc;
```



```
    }
    public int getDeckLevel() {
        return deckLevel;
    }
    public void setDeckLevel(int level ) {
        deckLevel = level;
    }

    public void setEntityContext(EntityContext ctx) {
         // Not implemented.
    }
    public void unsetEntityContext() {
         // Not implemented.
    }
    public void ejbActivate() {
        // Not implemented.
    }
    public void ejbPassivate() {
        // Not implemented.
    }
    public void ejbLoad() {
        // Not implemented.
    }
    public void ejbStore() {
        // Not implemented.
    }
    public void ejbRemove() {
        // Not implemented.
    }
}
```

Declared fields in a bean class can be persistent fields and property fields. These categories are not mutually exclusive. The persistent field declarations describe the fields that will be mapped to the database. A persistent field is often a property (in the JavaBeans sense): any attribute that is available using public set and get methods. Of course, a bean can have any fields that it needs; they need not all be persistent, or properties. Fields that aren't persistent won't be saved in the database. In `CabinBean`, all the fields are persis tent.

The `id` field is persistent, but it is not a property. In other words, `id` is mapped to the database but cannot be accessed through the remote interface.

The `name`, `deckLevel`, `ship`, and `bedCount` fields are persistent fields. They will be mapped to the database at deployment time. These fields are also properties because they are publicly available through the remote interface.

EJB 2.0 and 1.1: The callback methods

In the case of the Cabin EJB, there was only one `create()` method, so there is only one corresponding `ejbCreate()` method and one `ejbPostCreate()`

method. When a client invokes the `create()` method on the remote home interface, it is delegated to a matching `ejbCreate()` method on the entity bean instance. The `ejbCreate()` method initializes the fields; in the case of the `CabinBean`, it sets the `name` virtual field.

The `ejbCreate()` method always returns the primary key type; with container-managed persistence, this method returns the `null` value. It's the container's responsibility to create the primary key. Why does it return null? Simply put, it makes it easier for a bean-managed enterprise bean to extend a container-managed enterprise bean. This is valuable for EJB vendors who support container-managed persistence beans by extending them with bean-managed persistence beans implementations – it's a technique that was more common in EJB 1.1. Bean-managed persistence beans, which are covered in Chapter 10, always return the primary key type.

Once the `ejbCreate()` method has executed, the `ejbPostCreate()` method is called to perform any follow-up operations. The `ejbCreate()` and `ejbPostCreate()` methods must have signatures that match the parameters and (optionally) the exceptions of the home interface's `create()` method. The `ejbPostCreate()` method is used to perform any post processing on the bean after its created, but before it can be used by the client. Both methods will execute, one right after the other, when the client invokes the `create()` method on the remote home interface.

The `findByPrimaryKey()` method is not defined in container-managed bean classes. Instead, find methods are generated at deployment and implemented by the container. With bean-managed entity beans (entity beans that explicitly manage their own persistence), find methods must be defined in the bean class. In Chapter 10, when you develop bean-managed entity beans, you will define the find methods in the bean classes you develop.

The `CabinBean` class implements `javax.ejb.EntityBean`, which defines five callback methods: `setEntityContext()`, `unsetEntityContext()`, `ejbActivate()`, `ejbPassivate()`, `ejbLoad()`, `ejbStore()`, and `ejbRemove()`. The container uses these callback methods to notify the `CabinBean` of certain events in its life cycle. Although the callback methods are implemented, the implementations are empty. The `CabinBean` is simple enough that it doesn't need to do any special processing during its life cycle. When we study entity beans in more detail in Chapters 6 through 11, we will take advantage of these callback methods.

The Deployment Descriptor

You are now ready to create a deployment descriptor for the `Cabin EJB`. The deployment descriptor performs a function similar to a properties file. It describes

which classes make up a enterprise bean and how the enterprise bean should be managed at runtime. During deployment, the deployment descriptor is read and its properties are displayed for editing. The deployer can then modify and add settings as appropriate for the application's operational environment. Once the deployer is satisfied with the deployment information, he or she uses it to generate the entire supporting infrastructure needed to deploy the enterprise bean in the EJB server. This may include adding the enterprise bean to the naming system and generating the enterprise bean's EJB object and EJB home, persistence infrastructure, transactional support, resolving enterprise bean references, and so forth.

Although most EJB server products provide a wizard for creating and editing deployment descriptors, we will create ours directly so that the enterprise bean is defined in a vendor-independent manner.[4] This requires some manual labor, but it gives you a much better understanding of how deployment descriptors are created. Once the deployment descriptor is finished, the enterprise bean can be placed in a JAR file and deployed on any EJB-compliant server of the appropriate version.

An XML deployment descriptor for every example in this book has already been created and is available from the download site.

Here's a quick peek at the deployment descriptor for the Cabin EJB, so you can get a feel for how an XML deployment descriptor is structured and the type of information it contains:

EJB 2.0: The Cabin EJB's Deployment Descriptor

```
<?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_2_0.dtd">

<ejb-jar>
 <enterprise-beans>
   <entity>
      <ejb-name>CabinEJB</ejb-name>
      <home>com.titan.cabin.CabinHomeRemote</home>
      <remote>com.titan.cabin.CabinRemote</remote>
      <ejb-class>com.titan.cabin.CabinBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <abstract-schema-name>Cabin</abstract-schema-name>
      <cmp-field><field-name>id</field-name></cmp-field>
```

[4] The workbooks show you how to use the vendor's tools for creating deployment descriptors.


```
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>deckLevel</field-name></cmp-field>
      <cmp-field><field-name>shipId</field-name></cmp-field>
      <cmp-field><field-name>bedCount</field-name></cmp-field>
      <primkey-field>id</primkey-field>
      <security-identity><use-callers-identity/><security-identity>
   </entity>
 </enterprise-beans>
 <assembly-descriptor>
  ...
 </assembly-descriptor>
</ejb-jar>
```

EJB 1.1: The Cabin EJB's Deployment Descriptor

```
<?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>
 <enterprise-beans>
   <entity>
      <ejb-name>CabinEJB</ejb-name>
      <home>com.titan.cabin.CabinHomeRemote</home>
      <remote>com.titan.cabin.CabinRemote</remote>
      <ejb-class>com.titan.cabin.CabinBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field><field-name>id</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>deckLevel</field-name></cmp-field>
      <cmp-field><field-name>shipId</field-name></cmp-field>
      <cmp-field><field-name>bedCount</field-name></cmp-field>
      <primkey-field>id</primkey-field>
   </entity>
 </enterprise-beans>
 <assembly-descriptor>
  ...
 </assembly-descriptor>
</ejb-jar>
```

EJB 2.0 and 1.1: Defining the XML elements

The `<!DOCTYPE>` element describes the purpose of the XML file, its root element, and the location of its DTD. The DTD is used to verify that the document is structured correctly. This element is discussed in detail in Chapter 16. One important distinction between EJB 2.0 and EJB 1.1 is that they use different DTD for deployment descriptors. EJB 2.0 specifies the `ejb-jar_2_0.dtd` while EJB 1.1 specifies the `ejb-jar_1_1.dtd`.

The rest of the XML elements are nested one within the other and are delimited by a beginning tag and ending tag. The structure is really not very complicated. If you have done any HTML coding you should already understand the format. An element always starts with <*name of tag*> tag and ends with </*name of tag*> tag. Everything in between—even other elements—is part of the enclosing element.

The first major element is the `<ejb-jar>` element, which is the root of the document. All the other elements must lie within this element. Next is the `<enterprise- beans>` element. Every bean declared in an XML file must be included in this section. This file only describes the Cabin EJB, but we could define several beans in one deployment descriptor.

The `<entity>` element shows that the beans defined within this tag are entity beans. Similarly, a `<session>` element describes session beans; since the Cabin EJB is an entity bean, we don't need a `<session>` element. In addition to a description, the `<entity>` element provides the fully qualified class names of the remote interface, home interface, bean class, and primary key. The `<cmp-field>` elements list all the container-managed fields in the entity bean class. These are the fields that will be persisted in the database and are managed by the container at runtime. The `<entity>` element also includes a `<reentrant>` element that can be set as `True` or `False` depending on whether the bean allows reentrant loopbacks or not.

EJB 2.0 specifies a `name` which is used in EJB QL to identify the entity bean in queries. This isn't important right now. The 2.0 deployment descriptor also specifies `<security-identity>` as `<use-callers-identity>`, which simply means the bean will propagate the calling clients security identity when access resources or other beans. This was covered in detail in Chapter 3.

The next section of the XML file, after the `<enterprise-bean>` element, is enclosed by the `<assembly-descriptor>` element, which describes the security roles and transactional attributes of the bean. This section is the same for both EJB 2.0 and EJB 1.1 in this example.

```
<ejb-jar>
 <enterprise-beans>
   ...
 <enterprise-beans>
 <assembly-descriptor>
   <security-role>
      <description>
          This role represents everyone who is allowed full access
          to the Cabin EJB.
      </description>
     <role-name>everyone</role-name>
   </security-role>
```



```
    <method-permission>
      <role-name>everyone</role-name>
      <method>
          <ejb-name>CabinEJB</ejb-name>
          <method-name>*</method-name>
      </method>
    </method-permission>

    <container-transaction>
      <method>
         <ejb-name>CabinEJB</ejb-name>
         <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

It may seem odd to separate the `<assembly-descriptor>` information from the `<enterprise-beans>` information, since it clearly applies to the Cabin EJB, but in the scheme of things it's perfectly natural. A single XML deployment descriptor can describe several beans, which might all rely on the same security roles and transactional attributes. To make it easier to deploy several beans together, all this common information is separated into the `<assembly-descriptor>` element.

There is another reason (perhaps a more important reason) for separating information about the bean itself from the security roles and transactional attributes. The Enterprise JavaBeans defines the responsibilities of different participants in the development and deployment of beans. We don't address these development roles in this book because they are not critical to learning the fundamentals of EJB. For now, it's enough to know that the person who develops the bean and the person who assembles the beans into an application have separate responsibilities and therefore separate parts of the XML deployment descriptor. The bean developer is responsible for everything within the `<enterprise-beans>` element; the bean assembler is responsible for everything within the `<assembly-descriptor>`. Throughout this book we will play both roles, developing the beans and assembling them. But in a real project, you might buy a set of beans developed by a third-party vendor, who would have no idea how you intend to use the beans, what your security requirements are, etc. There is also the role of deployer, which is the person who actually loads the enterprise bean into the EJB container; and the Administrator who is responsible for tuning the EJB server and managing it at runtime. In some projects all these roles may be filled by on or two people, or by several different individuals or even teams. Again, you'll be assuming all these roles when reading this book, which is only practical since you can read a book as a team, but its also practical since you learn the responsibilities of each role anyway.

The <assembly-descriptor> contains the <security-role> elements and their corresponding <method-permission> elements, which were described in Chapter 3 under "Security." In this example, there is one security role, everyone, which is mapped to all the methods in the Cabin EJB using the <method-permission> element. (The * in the <method-name> element means "all methods"). As already mentioned, for EJB 2.0 you'll have to specify a security-identity; in this case it's the caller's identity.

The container-transaction element declares that all the methods of the Cabin EJB have a Required transactional attribute. Transactional attributes are explained in more detail in Chapter 14, but for now it means that all the methods must be executed within a transaction. The deployment descriptor ends with the enclosing tab of the <ejb-jar> element.

Copy the Cabin EJB's deployment descriptor into the same directory as the class files for the Cabin EJB files (*Cabin.class*, *CabinHome.class*, *CabinBean.class*, and *CabinPK. class*) and save it as *ejb-jar.xml*. You have now created all the files you need to package your EJB 1.1 Cabin EJB. Figure 4-3 shows all the files that should be in the *cabin* directory.

[FIGURE]

Figure 4-2: The Cabin EJB files (EJB 1.1)

cabin.jar: The JAR File

The JAR file is a platform-independent file format for compressing, packaging, and delivering several files together. Based on ZIP file format and the ZLIB compression standards, the JAR (Java archive) packages and tool were originally developed to make downloads of Java applets more efficient. As a packaging mechanism, however, the JAR file format is a very convenient way to "shrink-wrap" components and other software for delivery to third parties. The original JavaBeans component architecture depends on JAR files for packaging, as does Enterprise JavaBeans. The goal in using the JAR file format in EJB is to package all the classes and interfaces associated with a bean, including the deployment descriptor into one file. The process of creating an EJB JAR file is slightly different between EJB 1.1 and EJB 1.0.

Creating the JAR file for deployment is easy. Position yourself in the *dev* directory that is just above the *com/titan/cabin* directory tree, and execute the command:

```
\dev % jar cf cabin.jar com/titan/cabin/*.class META-INF/ejb-jar.xml

F:\..\dev>jar cf cabin.jar com\titan\cabin\*.class META-INF\ejb-jar.xml
```

You might have to create the *META-INF* directory first and copy *ejb-jar.xml* into that directory. The c option tells the *jar* utility to create a new JAR file that

contains the files indicated in subsequent parameters. It also tells the *jar* utility to stream the resulting JAR file to standard output. The `f` option tells *jar* to redirect the standard output to a new file named in the second parameter (*cabin.jar)* . It's important to get the order of the option letters and the command-line parameters to match. You can learn more about the *jar* utility and the `java.util.zip` package in *Java™ in a Nutshell* by David Flanagan, or *Learning Java™* (formerly *Exploring Java™*), by Pat Niemeyer and Jonathan Knudsen (both published by O'Reilly).

The *jar* utility creates the file *cabin.jar* in the *dev* directory. If you're interested in looking at the contents of the JAR file, you can use any standard ZIP application (WinZip, PKZIP, etc.), or you can use the command *jar tvf cabin.jar.*

Creating a CABIN Table in the Database

One of the primary jobs of a deployment tool is mapping entity beans to databases. In the case of the Cabin EJB, we must map its `id`, `name`, `deckLevel`, `ship`, and `bedCount` container-managed fields to some data source. Before proceeding with deployment, you need to set up a database and create a `CABIN` table. You can use the following standard SQL statement to create a `CABIN` table that will be consistent with the examples provided in this chapter:

```
create table CABIN
(
  ID int primary key,
  SHIP_ID int,
  BED_COUNT int,
  NAME char(30),
  DECK_LEVEL int
)
```

This statement creates a `CABIN` table that has five columns corresponding to the container-managed fields in the `CabinBean` class. Once the table is created and connectivity to the database is confirmed, you can proceed with the deployment process.

Deploying the Cabin EJB

Deployment is the process of reading the bean's JAR file, changing or adding properties to the deployment descriptor, mapping the bean to the database, defining access control in the security domain, and generating vendor-specific classes needed to support the bean in the EJB environment. Every EJB server product has its own deployment tools, which may provide a graphical user interface, a set of command-line programs, or both. Graphical deployment "wizards" are the easiest deployment tools to work with.

A deployment tool reads the JAR file and looks for the *ejb-jar.xml* file. In a graphical deployment wizard, the deployment descriptor elements will be presented in a set of property sheets similar to those used to customize visual components in environments like Visual Basic, PowerBuilder, JBuilder, and Symantec Café. Figure 4-7 shows the deployment wizard used in the J2EE Reference Implementation.

[FIGURE]

Figure 4-4: J2EE Reference Implementation's deployment wizard

The J2EE Reference Implementation's deployment wizard has fields and panels that match the XML deployment descriptor. You can map security roles to users groups, set the JNDI look up name, map the container-managed fields to the database, etc.

Different EJB deployment tools will provide varying degrees of support for mapping container-managed fields to a data source. Some provide very robust and sophisticated graphical user interfaces, while others are simpler and less flexible. Fortunately, mapping the `CabinBean`'s container-managed fields to the `CABIN` table is a fairly straightforward process. The documentation for your vendor's deployment tool will show you how to create this mapping. Once you have finished the mapping, you can complete the deployment of the Cabin EJB and prepare to access it from the EJB server.

Creating a Client Application

Now that the Cabin EJB has been deployed in the EJB server, we want to access it from a remote client. When we say remote, we are usally talking about a client application that is located on a different computer, or a different process on the same computer. In this section, we will create a remote client that will connect to the EJB server, locate the EJB remote home for the Cabin EJB, and create and interact with several Cabin EJBs. The following code shows a Java application that is designed to create a new Cabin EJB, set its `name`, `deckLevel`, `ship`, and `bedCount` properties, and then locate it again using its primary key:

```
package com.titan.cabin;

import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;
```

```
public class Client_1 {
    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();
            Object ref = jndiContext.lookup("CabinHome");
            CabinHomeRemote home = (CabinHomeRemote)
                PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
            CabinRemote cabin_1 = home.create(new Integer(1));
            cabin_1.setName("Master Suite");
            cabin_1.setDeckLevel(1);
            cabin_1.setShipId(1);
            cabin_1.setBedCount(3);

            Integer pk = new Integer(1);

            CabinRemote cabin_2 = home.findByPrimaryKey(pk);
            System.out.println(cabin_2.getName());
            System.out.println(cabin_2.getDeckLevel());
            System.out.println(cabin_2.getShipId());
            System.out.println(cabin_2.getBedCount());

        } catch (java.rmi.RemoteException re){re.printStackTrace();}
          catch (javax.naming.NamingException ne){ne.printStackTrace();}
          catch (javax.ejb.CreateException ce){ce.printStackTrace();}
          catch (javax.ejb.FinderException fe){fe.printStackTrace();}
    }

    public static Context getInitialContext()
        throws javax.naming.NamingException {

        Properties p = new Properties();
        // ... Specify the JNDI properties specific to the vendor.
        return new javax.naming.InitialContext(p);
    }
}
```

To access an enterprise bean, a client starts by using the JNDI package to obtain a directory connection to a bean's container. JNDI is an implementation-independent API for directory and naming systems. Every EJB vendor must provide directory services that are JNDI-compliant. This means that they must provide a JNDI service provider, which is a piece of software analogous to a driver in JDBC. Different service providers connect to different directory services—not unlike JDBC, where different drivers connect to different relational databases. The method `getInitialContext()` contains logic that uses JNDI to obtain a network connection to the EJB server.

The code used to obtain the JNDI `Context` will be different depending on which EJB vendor you are using. Consult your vendor's documentation to find out how to obtain a JNDI `Context` appropriate to your product. The code used

to obtain a JNDI `Context` in WebSphere, for example, might look something like the following:

```
public static Context getInitialContext()
 throws javax.naming.NamingException {

    java.util.Properties properties = new java.util.Properties();
    properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
    properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                   "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
    return new InitialContext(properties);
}
```

The same method developed for BEA's WebLogic Server would be different:

```
public static Context getInitialContext()throws javax.naming.NamingException {

   Properties p = new Properties();
   p.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.TengahInitialContextFactory");
   p.put(Context.PROVIDER_URL, "t3://localhost:7001");
   return new javax.naming.InitialContext(p);
}
```

Once a JNDI connection is established and a context is obtained from the `getIntialContext()` method, the context can be used to look up the EJB home of the Cabin EJB:

The `Client_1` application uses the `PortableRemoteObject.narrow()` method as prescribed in EJB 1.1:

```
Object ref = jndiContext.lookup("CabinHome");
CabinHome home = (CabinHomeRemote)
         PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
```

The `PortableRemoteObject.narrow()` method was first introduced in EJB 1.1 and continues to be used on remote clients in EJB 2.0. It is needed to support the requirements of RMI over IIOP. Because CORBA supports many different languages, casting is not native to CORBA (some languages don't have casting). Therefore, to get a remote reference to `CabinHomeRemote`, we must explicitly narrow the object returned from `lookup()`. This has the same effect as casting and is explained in more detail in Chapter 5.

The name used to find the Cabin EJB's EJB home is set by the deployer using a deployment wizard like the one pictured earlier. The JNDI name is entirely up to the person deploying the bean; it can be the same as the bean name set in the XML deployment descriptor or something completely different.

Creating a new Cabin EJB

Once we have a remote reference to the EJB home, we can use it to create a new Cabin entity:

```
CabinRemote cabin_1 = home.create(new Integer(1));
```

We create a new `Cabin` entity using the `create(Integer id)` method defined in the remote home interface of the Cabin EJB. When this method is invoked, the EJB home works with the EJB server to create a Cabin EJB, adding its data to the database. The EJB server then creates an EJB object to wrap the Cabin EJB instance and returns a remote reference to the EJB object to the client. The `cabin_1` variable then contains a remote reference to the Cabin EJB we just created.

We don't need to use the `PortableRemoteObject.narrow()` method to get the EJB object from the home reference, because it was declared as returning the `Cabin` type; no casting was required. We don't need to explicitly narrow remote references returned by `findByPrimaryKey()` for the same reason.

With the remote reference to the EJB object, we can update the `name`, `deckLevel`, `ship`, and `bedCount` of the Cabin EJB:

```
CabinRemote cabin_1 = home.create(new Integer(1));
cabin_1.setName("Master Suite");
cabin_1.setDeckLevel(1);
cabin_1.setShipId(1);
cabin_1.setBedCount(3);
```

Figure 4-11 shows how the relational database table that we created should look after executing this code. It should contain one record.

[FIGURE]

Figure 4-6: CABIN table with one cabin record

After an entity bean has been created, a client can locate it using the `findByPrimaryKey()` method in the home interface. First, we create a primary key of the correct type, in this case `Integer`. When we invoke the finder method on the home interface using the primary key, we get back a remote reference to the EJB object. We can now interrogate the remote reference returned by `findByPrimaryKey()` to get the Cabin EJB's `name`, `deckLevel`, `ship`, and `bedCount`:

```
Integer pk = new Integer(1);

CabinRemote cabin_2 = home.findByPrimaryKey(pk);
System.out.println(cabin_2.getName());
System.out.println(cabin_2.getDeckLevel());
System.out.println(cabin_2.getShipId());
System.out.println(cabin_2.getBedCount());
```


You are now ready to create and run the `Client_1` application against the Cabin EJB you deployed in earlier. Compile the client application and deploy the Cabin EJB into the container system. Then run the `Client_1` application.

📖 Exersize 4.1, Developing and deploying the Cabin EJB

When you run the `Client_1` application, your output should look something like the following:

```
Master Suite
1
1
3
```

Congratulations! You just created and used your first entity bean! Of course, the client application doesn't do much. Before going on to create session beans, create another client that adds some test data to the database. Here we'll create `Client_2` as a modification of `Client_1` that populates the database with a large number of cabins for three different ships:

```
package com.titan.cabin;

import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;

public class Client_2 {

    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();

            Object ref =
                jndiContext.lookup("CabinHome");
            CabinHomeRemote home = (CabinHomeRemote)
              PortableRemoteObject.narrow(ref,CabinHomeRemote.class);
            // Add 9 cabins to deck 1 of ship 1.
            makeCabins(home, 2, 10, 1, 1);
            // Add 10 cabins to deck 2 of ship 1.
            makeCabins(home, 11, 20, 2, 1);
            // Add 10 cabins to deck 3 of ship 1.
            makeCabins(home, 21, 30, 3, 1);

            // Add 10 cabins to deck 1 of ship 2.
```



```
            makeCabins(home, 31, 40, 1, 2);
            // Add 10 cabins to deck 2 of ship 2.
            makeCabins(home, 41, 50, 2, 2);
            // Add 10 cabins to deck 3 of ship 2.
            makeCabins(home, 51, 60, 3, 2);

            // Add 10 cabins to deck 1 of ship 3.
            makeCabins(home, 61, 70, 1, 3);
            // Add 10 cabins to deck 2 of ship 3.
            makeCabins(home, 71, 80, 2, 3);
            // Add 10 cabins to deck 3 of ship 3.
            makeCabins(home, 81, 90, 3, 3);
            // Add 10 cabins to deck 4 of ship 3.
            makeCabins(home, 91, 100, 4, 3);

            for (int i = 1; i <= 100; i++){
                Integer pk = new Integer(i);
                CabinRemote cabin = home.findByPrimaryKey(pk);
                System.out.println("PK = "+i+", Ship = "+cabin.getShipId()
                  + ", Deck = "+cabin.getDeckLevel()
                  + ", BedCount = "+cabin.getBedCount()
                  + ", Name = "+cabin.getName());
            }

        } catch (java.rmi.RemoteException re) {re.printStackTrace();}
          catch (javax.naming.NamingException ne) {ne.printStackTrace();}
          catch (javax.ejb.CreateException ce) {ce.printStackTrace();}
          catch (javax.ejb.FinderException fe) {fe.printStackTrace();}
    }

    public static javax.naming.Context getInitialContext()
        throws javax.naming.NamingException{
      Properties p = new Properties();
      // ... Specify the JNDI properties specific to the vendor.
      return new javax.naming.InitialContext(p);
    }

    public static void makeCabins(CabinHomeRemote home,
                                  int fromId, int toId,
                                  int deckLevel, int shipNumber)
      throws RemoteException, CreateException {

      int bc = 3;
      for (int i = fromId; i <= toId; i++) {
          CabinRemote cabin = home.create(new Integer(i));
          int suiteNumber = deckLevel*100+(i-fromId);
          cabin.setName("Suite "+suiteNumber);
          cabin.setDeckLevel(deckLevel);
          bc = (bc==3)?2:3;
          cabin.setBedCount(bc);
          cabin.setShipId(shipNumber);
```

```
    }
  }
}
```

Create and run the `Client_2` application against the Cabin EJB you deployed in earlier. `Client_2`, produces a lot of output that lists all the new Cabin EJBs you just added to the database.

```
PK = 1, Ship = 1, Deck = 1, BedCount = 3, Name = Master Suite
PK = 2, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 100
PK = 3, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 101
PK = 4, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 102
PK = 5, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 103
PK = 6, Ship = 1, Deck = 1, BedCount = 2, Name = Suite 104
PK = 7, Ship = 1, Deck = 1, BedCount = 3, Name = Suite 105
...
```

You now have 100 cabin records in your `CABIN` table, representing 100 cabin entities in your EJB system. This provides a good set of test data for the session bean we will create in the next section, and for subsequent examples throughout the book.

Developing a Session Bean

Session beans act as agents to the client, controlling workflow (the business process) and filling the gaps between the representation of data by entity beans and the business logic that interacts with that data. Session beans are often used to manage interactions between entity beans and can perform complex manipulations of beans to accomplish some task. Since we have only defined one entity bean so far, we will focus on a complex manipulation of the Cabin EJB rather than the interactions of the Cabin EJB with other entity beans. In Chapter 12, after we have had the opportunity to develop other entity beans, the interactions of entity beans within session beans will be explored in greater detail.

Client applications and other beans use the Cabin EJB in a variety of ways. Some of these uses were predictable when the Cabin EJB was defined, but many were not. After all, an entity bean represents data—in this case, data describing a cabin. The uses to which we put that data will change over time—hence the importance of separating the data itself from the workflow. In Titan's business system, for example, we may need to list and report on cabins in ways that were not predictable when the Cabin EJB was defined. Rather than change the Cabin EJB every time we need to look at it differently, we will obtain the information we need using a session bean. Changing the definition of an entity bean should only be done within the context of a larger process—for example, a major redesign of the business system.

In Chapters 1 and 2, we talked hypothetically about a TravelAgent EJB that was responsible for the workflow of booking a passage on a cruise. This session bean will be used in client applications accessed by travel agents throughout the world. In addition to booking tickets, the TravelAgent EJB also provides information about which cabins are available on the cruise. In this chapter, we will develop the first implementation of this listing behavior in the TravelAgent EJB. The listing method we develop in this example is admittedly very crude and far from optimal. However, this example is useful for demonstrating how to develop a very simple stateless session bean and how these session beans can manage other beans. In Chapter 12, we will rewrite the listing method. This "list cabins" behavior is used by travel agents to provide customers with a list of cabins that can accommodate the customer's needs. The Cabin EJB does not directly support the kind of list, nor should it. The list we need is specific to the TravelAgent EJB, so it's the TravelAgent EJB's responsibility to query the Cabin EJBs and produce the list.

You will need to create a development directory for the TravelAgent EJB, as we did for the Cabin EJB. We name this directory *travelagent* and nest it below the *com/titan* directory, which also contains the *cabin* directory (see Figure 4-13).

[FIGURE]

Figure 4-7: Directory structure for the TravelAgent EJB

You will be placing all the Java files and XML deployment descriptor for the TravelAgent EJB into this directory.

TravelAgentRemote: The Remote Interface

As before, we start by defining the remote interface so that our focus is on the business purpose of the bean, rather than its implementation. Starting small, we know that the TravelAgent EJB will need to provide a method for listing all the cabins available with a specified bed count for a specific ship. We'll call that method `listCabins()`. Since we only need a list of cabin names and deck levels, we'll define `listCabins()` to return an array of `String`s. Here's the remote interface for `TravelAgentRemote`:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.FinderException;

public interface TravelAgentRemote extends javax.ejb.EJBObject {

    // String elements follow the format "id, name, deck level"
    public String [] listCabins(int shipID, int bedCount)
        throws RemoteException;
}
```


TravelAgentHomeRemote: The Remote Home Interface

The second step in the development of the TravelAgent EJB bean is to create the remote home interface. The remote home interface for a session bean defines the create methods that initialize a new session bean for use by a client.

Find methods are not used in session beans; they are used with entity beans to locate persistent entities for use on a client. Unlike entity beans, session beans are not persistent and do not represent data in the database, so a find method would not be meaningful; there is no specific session to locate. A session bean is dedicated to a client for the life of that client (or less). For the same reason, we don't need to worry about primary keys; since session beans don't represent persistent data, we don't need a key to access that data.

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.CreateException;

public interface TravelAgentHomeRemote extends javax.ejb.EJBHome {
    public TravelAgentRemote create()
        throws RemoteException, CreateException;
}
```

In the case of the TravelAgent EJB, we only need a simple `create()` method to get a reference to the bean. Invoking this `create()` method returns a TravelAgent EJB's remote reference that the client can use for the reservation process.

TravelAgentBean: The Bean Class

Using the remote interface as a guide, we can define the `TravelAgentBean` class that implements the `listCabins()` method. The following code contains the complete definition of `TravelAgentBean` for this example.

```
package com.titan.travelagent;

import com.titan.cabin.CabinRemote;
import com.titan.cabin.CabinHomeRemote;
import java.rmi.RemoteException;
import javax.naming.InitialContext;
import javax.naming.Context;
import java.util.Properties;
import java.util.Vector;
import javax.rmi.PortableRemoteObject;
import javax.ejb.EJBException.
```



```
public class TravelAgentBean implements javax.ejb.SessionBean {

   public void ejbCreate() {
   // Do nothing.
   }
   public String [] listCabins(int shipID, int bedCount) {

        try {
            javax.naming.Context jndiContext = new InitialContext();
            Object obj =
                   jndiContext.lookup("java:comp/env/ejb/CabinHome");

            CabinHomeRemote home = (CabinHomeRemote)
                PortableRemoteObject.narrow(obj,CabinHomeRemote.class);

            Vector vect = new Vector();
            for (int i = 1; ; i++) {
              Integer pk = new Integer(i);
              CabinRemote cabin;
               try {
                    cabin = home.findByPrimaryKey(pk);
                } catch(javax.ejb.FinderException fe) {
                    break;
                }
                // Check to see if the bed count and ship ID match.
                if (cabin.getShipId() == shipID &&
                    cabin.getBedCount() == bedCount) {
                  String details =
                    i+","+cabin.getName()+","+cabin.getDeckLevel();
                  vect.addElement(details);
                }
            }

            String [] list = new String[vect.size()];
            vect.copyInto(list);
            return list;

        } catch(Exception e) {throw new EJBException(e);}
   }

   private javax.naming.Context getInitialContext()
   throws javax.naming.NamingException {
      Properties p = new Properties();
      // ... Specify the JNDI properties specific to the vendor.
      return new javax.naming.InitialContext(p);
   }

   public void ejbRemove(){}
   public void ejbActivate(){}
   public void ejbPassivate(){}
```



```
    public void setSessionContext(javax.ejb.SessionContext cntx){}
}
```

Examining the `listCabins()` method in detail, we can address the implementation in pieces, starting with the use of JNDI to locate the `CabinHomeRemote`:

```
javax.naming.Context jndiContext = new InitialContext();

Object obj = jndiContext.lookup("java:comp/env/ejb/CabinHome");

CabinHomeRemote home = (CabinHomeRemote)
    javax.rmi.PortableRemoteObject.narrow(obj, CabinHomeRemote.class);
```

Beans are clients to other beans, just like client applications. This means that they must interact with other beans in the same way that client applications interact with beans. In order for one bean to locate and use another bean, it must first locate and obtain a reference to the bean's EJB home. This is accomplished using JNDI in exactly the same way we used JNDI to obtain a reference to the Cabin EJB in the `Client_1` and `Client_2` applications we developed earlier.

All beans have a default JNDI context called the environment context, which was discussed a little in Chapter 3. The default context exists in the name space (directory) called `"java:comp/env"` and its subdirectories. When the bean is deployed, any beans it uses are mapped into the subdirectory `"java:comp/env/ejb"`, so that bean references can be obtained at runtime through a simple and consistent use of the JNDI default context. We'll come back to this when we take a look at the deployment descriptor for the TravelAgent EJB below.

In the case of the Cabin and TravelAgent EJBs we are working exclusively with there remote component interfaces. As you learned in Chapter 2, enterprise beans may have remote and/or local component interfaces. However, to keep things simple with this first set of examples, we are working with only the remote component interfaces – Chapter 5 will explain how this example may have been implemented with local interfaces.

Once the remote EJB home of the Cabin EJB is obtained, we can use it to produce a list of cabins that match the parameters passed. The following code loops through all the Cabin EJBs and produces a list that includes only those cabins with the ship and bed count specified:

```
Vector vect = new Vector();
for (int i = 1; ; i++) {
   Integer pk = new Integer(i);
   CabinRemote cabin;
   try {
       cabin = home.findByPrimaryKey(pk);
   } catch(javax.ejb.FinderException fe){
       break;
```



```
    }
    // Check to see if the bed count and ship ID match.
    if (cabin.getShipId() == shipID && cabin.getBedCount() == bedCount) {
       String details = i+","+cabin.getName()+","+cabin.getDeckLevel();
       vect.addElement(details);
    }
}
```

This method simply iterates through all the primary keys, obtaining a remote reference to each Cabin EJB in the system and checking whether its `shipId` and `bedCount` match the parameters passed in. The `for` loop continues until a `FinderException` is thrown, which would probably occur when a primary key is used that isn't associated with a bean. (This isn't the most robust code possible, but it will do for now.) Following this block of code, we simply copy the `Vector`'s contents into an array and return it to the client.

While this is a very crude approach to locating the right Cabin EJBs—we will define a better method in Chapter 12—it is adequate for our current purposes. The purpose of this example is to illustrate that the workflow associated with this listing behavior is not included in the Cabin EJB nor is it embedded in a client application. Workflow logic, whether it's a process like booking a reservation or obtaining a list, is placed in a session bean.

TravelAgent EJB's Deployment Descriptor

The TravelAgent EJB uses an XML deployment descriptor similar to the one used for the Cabin entity bean. Here is the *ejb-jar.xml* file used to deploy the TravelAgent. In Chapter 12, you will learn how to deploy several beans in one deployment descriptor, but for now the TravelAgent and Cabin EJBs are deployed separately.

EJB 2.0: Deployment Descriptor

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">
<ejb-jar>
 <enterprise-beans>
  <session>
    <ejb-name>TravelAgentEJB</ejb-name>
    <home>com.titan.travelagent.TravelAgentHomeRemote</home>
    <remote>com.titan.travelagent.TravelAgentRemote</remote>
    <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <ejb-ref>
      <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
```

```
      <home>CabinHomeRemote</home>
      <remote>CabinRemote</remote>
    </ejb-ref>
    <security-identity><use-callers-identity/></security-identity>
  </session>
 </enterprise-beans>
 <assembly-descriptor>
  ...
 </assembly-descriptor>
</ejb-jar>
```

EJB 1.1: Deployment Descriptor

```
<?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>
 <enterprise-beans>
  <session>
    <ejb-name>TravelAgentEJB</ejb-name>
    <home>com.titan.travelagent.TravelAgentHomeRemote</home>
    <remote>com.titan.travelagent.TravelAgentRemote</remote>
    <ejb-class>com.titan.travelagent.TravelAgentBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <ejb-ref>
      <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <home>com.titan.cabin.CabinHomeRemote</home>
      <remote>com.titan.cabin.CabinRemote</remote>
    </ejb-ref>
  </session>
 </enterprise-beans>
 <assembly-descriptor>
  ...
 </assembly-descriptor>
</ejb-jar>
```

EJB 2.0 and 1.1: Defining the XML elements

The only significant difference between the 2.0 and 1.1 deployment descriptors is the name of the DTD and the addition of a `<security-identity>` element in EJB 2.0, which simply propagates the caller's identity.

Other than the `<session-type>` and `<ejb-ref>` elements, the XML deployment descriptor should make sense since it uses many of the same elements as the Cabin EJB's. The `<session-type>` element can be

`Stateful` or `Stateless` to indicate which type of session bean is used. In this case we are defining a stateless session bean.

The `<ejb-ref>` element is used at deployment time to map the bean references used within the TravelAgent EJB. In this case, the `<ejb-ref>` element describes the Cabin EJB, which we already deployed. The `<ejb-ref-name>` element specifies the name that must be used by the TravelAgent EJB to obtain a reference to the Cabin EJB's home. The `<ejb-ref-type>` tells the container what kind of bean it is, `Entity` or `Session`. The `<home>` and `<remote>` elements specify the fully qualified interface names of the Cabin's home and remote bean interfaces.

When the bean is deployed, the `<ejb-ref>` will be mapped to the Cabin EJB in the EJB server. This is a vendor-specific process, but the outcome should always be the same. When the TravelAgent does a JNDI lookup using the context name "`java:comp/env/ejb/CabinHome`" it will obtain a remote reference to the Cabin EJB's home. The purpose of the `<ejb-ref>` element is to eliminate network specific and implementation specific use of JNDI to obtain remote bean references. This makes a bean more portable because the network location and JNDI service provider can change without impacting the bean code or even the XML deployment descriptor.

However, as you learn in Chapter 5, with EJB 2.0 it's always preferable to use local references instead of remote references when beans are access each other with the same server. Local references are specified using the `<ejb-local-ref>` element, which looks just like the `<ejb-ref>` element except it is for local references.

The `assembly-descriptor` section of the deployment descriptor is the same for EJB 2.0 and EJB 1.1.

```
<assembly-descriptor>
   <security-role>
      <description>
         This role represents everyone who is allowed full access
         to the Cabin EJB.
      </description>
     <role-name>everyone</role-name>
   </security-role>

   <method-permission>
     <role-name>everyone</role-name>
     <method>
         <ejb-name>TravelAgentEJB</ejb-name>
         <method-name>*</method-name>
     </method>
   </method-permission>
```

```
    <container-transaction>
      <method>
         <ejb-name>TravelAgentEJB</ejb-name>
         <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Deploying the TravelAgent EJB

Once the XML deployment descriptor is defined you are ready to place the TravelAgent EJB in its own JAR file and deploy it into the EJB server.

To make your TravelAgent EJB available to a client application, you need to use the deployment utility or wizard of your EJB server. The deployment utility reads the JAR file to add the TravelAgent EJB to the EJB server environment. Unless your EJB server has special requirements, it is unlikely that you will need to change or add any new attributes to the bean. You will not need to create a database table for this example, since the TravelAgent EJB is using only the Cabin EJB and is not itself persistent. However, you will need to map the `<ejb-ref>` element in the TravelAgent EJB's deployment descriptor to the Cabin EJB. You EJB server's deployment tool will provide a mechanism for doing this. Deploy the TravelAgent EJB and proceed to the next section.

Use the same process to JAR the TravelAgent EJB as was used for the Cabin EJB. We shrink-wrap the TravelAgent EJB class and its deployment descriptor into a JAR file and save to the *com/titan/travelagent* directory:

```
\dev % jar cf cabin.jar com/titan/travelagent/*.class META-INF/ejb-jar.xml

F:\..\dev>jar cf cabin.jar com\titan\travelagent\*.class META-INF\ejb-jar.xml
```

You might have to create the *META-INF* directory first, and copy *ejb-jar.xml* into that directory. The TravelAgent EJB is now complete and ready to be deployed. Next use your EJB containers proprietary tools to deploy the TravelAgent EJB into the container system.

Creating a Client Application

To show that our session bean works, we'll create a simple client application that uses it. This client simply produces a list of cabins assigned to ship 1 with a bed count of 3. Its logic is similar to the client we created earlier to test the Cabin EJB: it creates a context for looking up `TravelAgentHomeRemote`, creates a TravelAgent EJB, and invokes `listCabins()` to generate a list of the cabins available. Here's the code:


```
import com.titan.cabin.CabinHomeRemote;
import com.titan.cabin.CabinRemote;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;

public class Client_3 {
    public static int SHIP_ID = 1;
    public static int BED_COUNT = 3;

    public static void main(String [] args) {
        try {
           Context jndiContext = getInitialContext();

           Object ref = jndiContext.lookup("TravelAgentHome");
           TravelAgentHomeRemote home = (TravelAgentHomeRemote)
           PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

           TravelAgentRemote travelAgent = home.create();

           // Get a list of all cabins on ship 1 with a bed count of 3.
           String list [] = travelAgent.listCabins(SHIP_ID,BED_COUNT);

           for(int i = 0; i < list.length; i++){
              System.out.println(list[i]);
           }

        } catch(java.rmi.RemoteException re){re.printStackTrace();}
          catch(Throwable t){t.printStackTrace();}
  }
  static public Context getInitialContext() throws Exception {
    Properties p = new Properties();
    // ... Specify the JNDI properties specific to the vendor.
    return new InitialContext(p);
  }
}
```

When you have successfully run `Client_3`, the output should look like this:

```
1,Master Suite               ,1
3,Suite 101                  ,1
5,Suite 103                  ,1
7,Suite 105                  ,1
9,Suite 107                  ,1
12,Suite 201                 ,2
14,Suite 203                 ,2
16,Suite 205                 ,2
```

```
18,Suite 207                ,2
20,Suite 209                ,2
22,Suite 301                ,3
24,Suite 303                ,3
26,Suite 305                ,3
28,Suite 307                ,3
30,Suite 309                ,3
```

You have now successfully created the first piece of the TravelAgent session bean: a method that obtains a list of cabins by manipulating the Cabin EJB entity.

📖 Exercise 4.2, Develop and Deploy the TravelAgent EJB

5

The Client View

Developing the Cabin EJB and the TravelAgent EJB should have raised your confidence, but it should also have raised a lot of questions. So far, we have glossed over most of the details involved in developing, deploying, and accessing these enterprise beans. In this chapter and the ones that follow, we will slowly peel away the layers of the Enterprise JavaBeans onion to expose the details of EJB application development.

This chapter focuses specifically on the client's view of an EJB system. The client, whether it is an application or another enterprise bean, doesn't work directly with the beans in the EJB system. Instead, clients interact with a set of interfaces that provide access to beans and their business logic. These interfaces consist of the JNDI API and an EJB client-side API. JNDI allows us to find and access enterprise beans regardless of their location on the network; the EJB client-side API is the set of interfaces and classes that a developer uses on the client to interact with enterprise beans.

The best approach to this chapter is to read about a feature of the client view and then try working with some of the examples to see the feature in action. This will provide you with hands-on experience and a much clearer understanding of the concepts. Have fun, experiment, and you'll be sure to understand the fundamentals.

Locating Beans with JNDI

In Chapter 4, the client application started by creating an `InitialContext`, which it then used to get a remote reference to the homes of the Cabin and TravelAgent EJBs. The `InitialContext` is part of a larger API called the

Java Naming and Directory Interface (JNDI). We use JNDI to look up an EJB home in an EJB server just like you might use a phone book to find the home number of a friend or business associate.

JNDI is a standard Java optional package that provides a uniform API for accessing a wide range of services. In this respect, it is somewhat similar to JDBC, which provides uniform access to different relational databases. Just as JDBC lets you write code that doesn't care whether it's talking to an Oracle database or a Sybase database, JNDI lets you write code that can access different directory and naming services, like LDAP, Novell Netware NDS, CORBA Naming Service, and the naming services provided by EJB servers. EJB servers are required to support JNDI by organizing beans into a directory structure and providing a JNDI driver, called a *service provider*, for accessing that directory structure. Using JNDI, an enterprise can organize its beans, services, data, and other resources in a large virtual directory structure, which can provide a very powerful mechanism for binding together normally disparate systems.

The great thing about JNDI is that it is virtual and dynamic. JNDI is virtual because it allows one directory service to be linked to another through simple URLs. The URLs in JNDI are analogous to HTML links. Clicking on a link in HTML allows a user to load the contents of a web page. The new web page could be downloaded from the same host as the starting page or from a completely different web site—the location of the linked page is transparent to the user. Likewise, using JNDI, you can drill down through directories to files, printers, EJB home objects, and other resources using links that are similar to HTML links. The directories and subdirectories can be located in the same host or can be physically hosted at completely different locations. The user doesn't know or care where the directories are actually located. As a developer or administrator, you can create virtual directories that span a variety of different services over many different physical locations.

JNDI is dynamic because it allows the JNDI drivers (a.k.a. service providers) for specific types of directory services to be loaded at runtime. A driver maps a specific kind of directory service into the standard JNDI class interfaces. Drivers have been created for LDAP, Novell NetWare NDS, Sun Solaris NIS+, CORBA Naming Service, and many other types of naming and directory services. When a link to a different directory service is chosen, the driver for that type of directory service is automatically loaded from the directory's host, if it is not already resident on the user's machine. Automatically downloading JNDI drivers makes it possible for a client to navigate across arbitrary directory services without knowing in advance what kinds of services it is likely to find.

JNDI allows the application client to view the EJB server as a set of directories, like directories in a common filesystem. After the client application locates and obtains a remote reference to the EJB home using JNDI, the client can use the EJB home to obtain an EJB object reference to an enterprise bean. In the TravelAgent EJB and the Cabin EJB, which you worked with in Chapter 4, you

used the method `getInitialContext()` to get a JNDI `InitialContext` object, which looked as follows:

```
public static Context getInitialContext() throws javax.naming.NamingException {
   Properties p = new Properties();
    // ... Specify the JNDI properties specific to the vendor.
    return new javax.naming.InitialContext(p);
}
```

An initial context is the starting point for any JNDI lookup—it's similar in concept to the root of a filesystem. The way you create an initial context is peculiar, but not fundamentally difficult. You start with a properties table of type `Properties`. This is essentially a hash table to which you add various values that determine the kind of initial context you get.

Of course, as mentioned in Chapter 4, this code will change depending on how your EJB vendor has implemented JNDI. For WebSphere, `getInitialContext()` might look something like this:

```
public static Context getInitialContext()
 throws javax.naming.NamingException {

    java.util.Properties properties = new java.util.Properties();
    properties.put(javax.naming.Context.PROVIDER_URL, "iiop:///");
    properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                   "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
    return new InitialContext(properties);
}
```

For BEA's WebLogic Server, this method would be coded as:

```
public static Context getInitialContext() throws Exception {
    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.T3InitialContextFactory");
    p.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new InitialContext(p);
}
```

For a more detailed explanation of JNDI, see O'Reilly's *Java™ Enterprise in a Nutshell*, by David Flanagan, Jim Farley, William Crawford, and Kris Magnusson.

The Remote Client API

Enterprise bean developers are required to provide a bean class, component interfaces, and for entity beans, a primary key. Of these types, the only the component interfaces and primary key class are visible to the client, while the bean class is not. The component interfaces and primary key contribute to the client-side API in EJB. The methods defined in component interfaces as well as

the methods of their supertypes provide the mechanisms that clients use to interact with an EJB business system.

In EJB 1.1, all clients, whether they are in the same container system or not, *must* use the *Remote Client API*, which means they must use the remote interface and remote home interface and Java RMI is all their interactions. In EJB 2.0, remote clients must continue to use the Remote Client API, but enterprise beans that are located in the same EJB container system have the option of using the *Local Client API*. The Local Client API provides local component interfaces and avoids the restrictions and overhead of the remote client API.

This section examines in more detail the remote component interfaces and the primary key, as well as other types that make up EJB's remote client-side API. This will provide you with a better understanding of how the remote client-side API is used and its relationship with the bean class on the EJB server. In the next major section, The Local Client API, the use of local component interfaces will be examined.

Java RMI-IIOP

Enterprise JavaBeans 2.0 and 1.1 define an EJB's remote interfaces in terms of Java RMI-IIOP, which enforces compliance with CORBA. In other words, the underlying protocol used by remote clients to access enterprise beans can be anything that the vendor wants as long as it supports the types of interfaces and arguments that are compatible with Java RMI-IIOP. EJB 1.1 only required that the wire protocol used by vendors utilize types that would be compatible with Java RMI-IIOP. In other words, the interface types and values used in remote references had to be compliant with the types allowed for Java RMI-IIOP. This ensured that early Java RMI-IIOP adopters were supported and makes for a seamless transition for other vendors who wanted to use real Java RMI-IIOP in EJB 2.0. In EJB 2.0, vendor can still offer other Java RMI-IIOP-compatible protocols, but in addition to any propritary protocols they support, they *must* also support the CORBA IIOP 1.2 protocol as defined in the CORBA 2.3.1.

To be compliant with Java RMI-IIOP types, the EJB vendors have to restrict the definition of interfaces and arguments to types that map nicely to IIOP 1.2. These restrictions are really not all that bad, and you probably won't even notice them while developing your beans, but it's important to know what they are. The next few paragraphs discuss the Java RMI-IIOP programming model for both EJB 2.0 and EJB 1.1.

EJB 2.0's local component interfaces are not Java RMI interfaces and do not have to support IIOP 1.2 or use types compliant with the Java RMI-IIOP protocol. Local component interfaces are discused after remote component interfaces.

Java RMI Return Types, Parameters, and Exceptions

The supertypes of the remote home interface and remote interface, `javax.ejb.EJBHome` and `javax.ejb.EJBObject`, both extend `java.rmi.Remote`. As `Remote` interface subtypes, they are expected to adhere to the Java RMI specification for `Remote` interfaces.

Parameters and return types

As subtypes of the `java.rmi.Remote` interface, the remote component interfaces must follow several guidelines, some of which apply to the return types and parameters that are allowed. To be compatible with Java RMI, the *actual* return types and parameter types used in the `java.rmi.Remote` interfaces must be primitives, `String` types, `java.rmi.Remote` types, or serializable types.

There is a difference between *declared* types, which are checked by the compiler, and *actual* types, which are checked by the runtime. The types that may be used in Java RMI are actual types, which are either primitive types, object types implementing (even indirectly) `java.rmi.Remote`, or object types implementing (even indirectly) `java.io.Serializable`. The `java.util.Collection` type, for example, which does not explicitly extends `java.io.Serializable`, is a perfectly valid return type for a remote finder methods, provided that the concrete class implementing `Collection` does implement `java.io.Serializable`. So Java RMI has *no* special rules regarding declared return types or parameter types. At runtime, a type that is not a `java.rmi.Remote` type is assumed to be serializable; if it is not, an exception is thrown. The actual type passed cannot be checked by the compiler, it must be checked at the runtime.

Here is a list of the types that can be passed as parameters or returned in Java RMI:

- Primitives: byte, boolean, char, short, int, long, double, float.
- Java serializable types: any class that implements or any interface that extends `java.io.Serializable`.
- Java RMI remote types: any class that implements or any interface that extends `java.rmi.Remote`.

Serializable objects are passed by copy (a.k.a. passed by value), not by reference, which means that changes in a serialized object on one tier are not automatically reflected on the others. Objects that implement `Remote`, like `CustomerRemote` or `CabinRemote`, are passed as remote references—which is a little different. A remote reference is a `Remote` interface implemented by a distributed object stub. When a remote reference is passed as a parameter or returned from a method, it is the stub that is serialized and passed by value, not the object server remotely referenced by the stub. In the home interface for the

TravelAgent EJB, the `create()` method takes a reference to a Customer EJB as its only argument.

```
public interface TravelAgentHomeRemote extends javax.ejb.EJBHome {
    public TravelAgentRemote create(CustomerRemote customer)
        throws RemoteException, CreateException;
}
```

The `customer` argument is a remote reference to a Customer EJB that is passed into the `create()` method. When a remote reference is passed or returned in Enterprise JavaBeans, the EJB object stub is passed by copy. The copy of the EJB object stub points to the same EJB object as the original stub. This results in both the enterprise bean instance and the client having remote references to the same EJB object. So changes made on the client using the remote reference will be reflected when the enterprise bean instance uses the same remote reference. Figure 5-1 and Figure 5-3 show the difference between a serializable object and a remote reference argument in Java RMI.

[FIGURE]

Figure 5-1: Serializable arguments in Java RMI

[FIGURE]

Figure 5-2: Remote reference arguments in Java RMI

Exceptions

The Java RMI specification states that every method defined in a `Remote` interface must throw a `java.rmi.RemoteException`. The `RemoteException` is used when problems occur with the distributed object communications, like a network failure or inability to locate the object server. In addition, `Remote` interface types can throw any application-specific exceptions (exceptions defined by the application developer) that are necessary. The following code shows the remote interface to the TravelAgent EJB discussed in Chapter 2. This remote interface is similar to the one defined in Chapter 4. `TravelAgentRemote` has several remote methods, including `bookPassage()`. The `bookPassage()` method can throw a `RemoteException` (as required), in addition to an application exception, `IncompleteConversationalState`.

```
public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public void setCruiseID(int cruise)
        throws RemoteException, FinderException;
    public int getCruiseID() throws RemoteException;

    public void setCabinID(int cabin)
        throws RemoteException, FinderException;
    public int getCabinID() throws RemoteException;
```



```
    public int getCustomerID() throws RemoteException;

    public Ticket bookPassage(CreditCardRemote card, double price)
        throws RemoteException,IncompleteConversationalState;

    public String [] listAvailableCabins(int bedCount)
        throws RemoteException, IncompleteConversationalState;

}
```

Java RMI-IIOP type restrictions

In addition to the Java RMI programming model discussed earlier, Java RMI-IIOP imposes additional restrictions on the remote interfaces and value types used in the Remote Client API. These restrictions are born of limitations inherit in the Interface Definition Language (IDL) upon which CORBA IIOP 1.2 is based. The exact nature of these limitations is outside the scope of this book. Here are two of the restrictions; the others, like IDL name collisions, are so rarely encountered that it wouldn't be constructive to mention them.[1]

- Method overloading is restricted; a remote interface may not directly extend two or more interfaces that have methods with the same name (even if their arguments are different). A remote interface may, however, overload its own methods and extend a remote interface with overloaded method names. Overloading is viewed, here, as including overriding. Figure 5-3 illustrates both of these situations.

[FIGURE]

Figure 5-3: Overloading rules for Remote interface inheritance in Java RMI-IIOP

- Serializable types must not directly or indirectly implement the `java.rmi.Remote` interface.

Explicit narrowing using PortableRemoteObject

In Java RMI-IIOP remote references must be explicitly narrowed using the `javax.rmi.PortableRemoteObject.narrow()` method. The typical practice in Java would be to cast the reference to the more specific type, as follows:

```
javax.naming.Context jndiContext;
...
CabinHomeRemote home = (CabinHomeRemote)jndiContext.lookup("CabinHome");
```

---

1 To learn more about CORBA IDL and its mapping to the Java language consult *The Common Object Request Broker: Architecture and Specification* and *The Java Language to IDL Mapping* available at the OMG site (*www.omg.org*).

The `javax.naming.Context.lookup()` method returns an `Object`. In EJB 2.0's Local Client API, we can assume that it is legal to cast the return argument. However, the Remote Client API must be compatible with Java RMI-IIOP, which means that clients must adhere to limitations imposed by the IIOP 1.2 protocol. To accommodate all languages, many of which have no concept of casting, IIOP 1.2 does not support stubs that implement multiple interfaces. The stub returned in IIOP implements only the interface specified by the return type of the remote method that was invoked. If the return type is `Object`, as is the remote reference returned by the `lookup()` method, the stub will only implement methods specific to the `Object` type.

Of course, some means for converting a remote reference from a more general type to a more specific type is essential in an object-oriented environment, so Java RMI-IIOP provides a mechanism for explicitly narrowing references to a specific type. The `javax.rmi.PortableRemoteObject.narrow()` method abstracts this narrowing to provide narrowing in IIOP as well as other protocols. Remember while the Remote Client API requires that you use Java RMI-IIOP reference and argument types, the wire protocol need not be IIOP 1.2. Other protocols besides IIOP may also require explicit narrowing. The `PortableRemoteObject` abstracts the narrowing process so that any protocol can be used.

To narrow the return argument of the `Context.lookup()` method to the appropriate type, we must explicitly ask for a remote reference that implements the interface we want:

```
import javax.rmi.PortableRemoteObject;
...
javax.naming.Context jndiContext;
...
Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

When the `narrow()` method has successfully executed, it returns a stub that implements the `Remote` interface specified. Because the stub is known to implement the correct type, you can then use Java's native casting to narrow the stub to the correct `Remote` interface. The `narrow()` method takes two arguments: the remote reference that is to be narrowed and the type it should be narrowed to. The definition of the `narrow()` method is:[2]

```
package javax.rmi;

public class PortableRemoteObject extends java.lang.Object {

    public static java.lang.Object narrow(java.lang.Object narrowFrom,
```

2 Other methods included in the `PortableRemoteObject` class are not important to EJB application developers. They are intended for Java RMI developers.


```
                                        java.lang.Class narrowTo)
    throws java.lang.ClassCastException;
    ...
}
```

The `narrow()` method only needs to be used when a remote reference to an EJB home or EJB object is returned without a specific `Remote` interface type. This occurs in six circumstances:

- When a remote EJB home reference is obtained using the `javax.naming.Context.lookup()` method:

```
Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

- When a remote EJB object reference is obtained using the `javax.ejb.Handle.getEJBObject()` method:

```
Handle handle = .... // get handle
Object ref = handle.getEJBObject();
CabinRemote cabin = (CabinRemote)
PortableRemoteObject.narrow(ref,CabinRemote.class);
```

- When a remote EJB home reference is obtained using the `javax.ejb.HomeHandle.getEJBHome()` method:

```
HomeHandle homeHdle = ... // get home handle
EJBHome ref = homeHdle.getEJBHome();
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

- When a remote EJB home reference is obtained using the `javax.ejb.EJBMetaData.getEJBHome()` method:

```
EJBMetaData metaData = homeHdle.getEJBMetaData();
EJBHome ref = metaData.getEJBHome();
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

- When a remote EJB object reference is obtained from a collection returned by a remote home interface finder method:

```
ShipHomeRemote shipHome = ... // get ship home
Enumeration enum = shipHome.findByCapacity(2000);
while(enum.hasMoreElements()){
    Object ref = enum.nextElement();
    ShipRemote ship = (ShipRemote)
              PortableRemoteObject.narrow(ref, ShipRemote.class);
    // do something with Ship reference
}
```

- When a wide remote EJB object type is returned from any business method. Here is a hypothetical example:

```
// Officer extends Crewman
ShipRemote ship = // get Ship remote reference
```



```
CrewmanRemote crew = ship.getCrewman("Burns", "John", "1st Lieutenant");
OfficerRemote burns = (OfficerRemote)
              PortableRemoteObject.narrow(crew, OfficerRemote.class);
```

The `PortableRemoteObject.narrow()` method is not required when the remote type is specified in the method signature. This is true of the `create()` methods and find methods in remote home interfaces that return a single bean. For example, the `create()` and `findByPrimaryKey()` methods defined in the `CabinHomeRemote` interface (Chapter 4) do not require the use of `narrow()` method because these methods already return the correct EJB object type. Business methods that return the correct type do not need to use the `narrow()` method either, as the following code illustrates:

```
/* The CabinHomeRemote.create() method specifies
 * the Cabin remote interface as the return type
 * so explicit narrowing is not needed.*/
CabinRemote cabin = cabinHome.create(12345);

/* The CabinHomeRemote.findByPrimaryKey() method specifies
 * the Cabin remote interface as the return type
 * so explicit narrowing is not needed.*/
CabinRemote cabin = cabinHome.findByPrimaryKey(12345);

/* The ShipRemote.getCrewman() business method specifies
 * the Crewman remote interface as the return type
 * so explicit narrowing is not needed.*/
CrewmanRemote crew = ship.getCrewman("Burns", "John", "1st Lieutenant");
```

The Remote Home Interface

The remote home interface provides life-cycle operations and metadata for the bean. When you use JNDI to access a bean, you obtain a remote reference, or stub, to the bean's EJB home, which implements the remote home interface. Every bean type may have one home interface, which extends the `javax.ejb.EJBHome` interface.

Here is the `EJBHome` interface:

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {
    public abstract EJBMetaData getEJBMetaData()
      throws RemoteException;
    public HomeHandle getHomeHandle()    // new in 1.1
      throws RemoteException;
    public abstract void remove(Handle handle)
      throws RemoteException, RemoveException;
    public abstract void remove(Object primaryKey)
      throws RemoteException, RemoveException;
}
```


Removing beans

The `EJBHome.remove()` methods are responsible for deleting an enterprise bean. The argument is either the `javax.ejb.Handle` of the enterprise bean or, if it's an entity bean, its primary key. The `Handle` will be discussed in more detail later, but it is essentially a serializable pointer to a specific enterprise bean. When either of the `EJBHome.remove()` methods are invoked, the remote reference to the enterprise bean on the client becomes invalid: the stub to the enterprise bean that was removed no longer works. If for some reason the enterprise bean can't be removed, a `RemoveException` is thrown.

The impact of the `EJBHome.remove()` on the enterprise bean itself depends on the type of bean. For session beans, the `EJBHome.remove()` methods end the session's service to the client. When `EJBHome.remove()` is invoked, the remote reference to the session beans becomes invalid, and any conversational state maintained by the session bean is lost. The TravelAgent EJB you created in Chapter 4 is stateless, so no conversational state exists (more about this in Chapter 7).

When a `remove()` method is invoked on an entity bean, the remote reference becomes invalid, and any data that it represents is actually deleted from the database. This is a far more destructive activity because once an entity bean is removed, the data that it represents no longer exists. The difference between using a `remove()` method on a session bean and using `remove()` on an entity bean is similar to the difference between hanging up on a telephone conversation and actually killing the caller on the other end. Both end the conversation, but the end results are a little different.

The following code fragment is taken from the `main()` method of a client application that is similar to the clients we created to exercise the Cabin and TravelAgent EJBs. It shows that you can remove enterprise beans using a primary key (entity only) or a handle. Removing an entity bean deletes the entity from the database; removing a session bean results in the remote reference becoming invalid.

```
Context jndiContext = getInitialContext();

// Obtain a list of all the cabins for ship 1 with bed count of 3.

Object ref = jndiContext.lookup("TravelAgentHome");
TravelAgentHomeRemote agentHome = (TravelAgentHomeRemote)
    PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

TravelAgentRemote agent = agentHome.create();
String list [] = agent.listCabins(1,3);
System.out.println("1st List: Before deleting cabin number 30");
for(int i = 0; i < list.length; i++){
    System.out.println(list[i]);
}
```



```
// Obtain the home and remove cabin 30. Rerun the same cabin list.

ref = jndiContext.lookup("CabinHome");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

Integer pk = new Integer(30);
c_home.remove(pk);
list = agent.listCabins(1,3);
System.out.println("2nd List: After deleting cabin number 30");
for (int i = 0; i < list.length; i++) {
    System.out.println(list[i]);
}
```

First, the application creates a list of cabins, including the cabin with the primary key 30. Then it removes the Cabin EJB with this primary key and creates the list again. The second time the iteration is performed, cabin 30 will not listed. Because it was removed, the `listCabin()` method was unable to find a cabin with a primary key equal to 30, so it stopped making the list. The bean, including its data, is no longer in the database.

Your output should look something like the following:

```
1st List: Before deleting cabin number 30
1,Master Suite                  ,1
3,Suite 101                     ,1
5,Suite 103                     ,1
7,Suite 105                     ,1
9,Suite 107                     ,1
12,Suite 201                     ,2
14,Suite 203                     ,2
16,Suite 205                     ,2
18,Suite 207                     ,2
20,Suite 209                     ,2
22,Suite 301                     ,3
24,Suite 303                     ,3
26,Suite 305                     ,3
28,Suite 307                     ,3
30,Suite 309                     ,3
2nd List: After deleting cabin number 30
1,Master Suite                  ,1
3,Suite 101                     ,1
5,Suite 103                     ,1
7,Suite 105                     ,1
9,Suite 107                     ,1
12,Suite 201                     ,2
14,Suite 203                     ,2
16,Suite 205                     ,2
18,Suite 207                     ,2
20,Suite 209                     ,2
22,Suite 301                     ,3
```



```
24,Suite 303                    ,3
26,Suite 305                    ,3
28,Suite 307                    ,3
```

Bean metadata

`EJBHome.getEJBMetaData()` returns an instance of `javax.ejb.EJBMetaData` that describes the remote home interface, remote interface, and primary key classes, plus whether the enterprise bean is a session or entity bean[3]. This type of metadata is valuable to Java tools like IDEs that have wizards or other mechanisms for interacting with an enteprise bean from a client's perspective. A tool could, for example, use the class definitions provided by the `EJBMetaData` with Java reflection to create an environment where deployed enterprise beans can be "wired" together by developers. Of course, information such as the JNDI names and URLs of the enterprise beans is also needed.

Most application developers rarely use the `EJBMetaData`. Knowing that it's there, however, is valuable when you need to create automatic code generators or some other automatic facility. In those cases, familiarity with the Reflection API is necessary.[4] The following code shows the interface definition for `EJBMetaData`. Any class that implements the `EJBMetaData` interface must be serializable; it cannot be a stub to a distributed object. This allows IDEs and other tools to save the `EJBMetaData` for later use.

```
public interface javax.ejb.EJBMetaData {
    public abstract EJBHome getEJBHome();
    public abstract Class getHomeInterfaceClass();
    public abstract Class getPrimaryKeyClass();
    public abstract Class getRemoteInterfaceClass();
    public abstract boolean isSession();
}
```

The following code shows how the `EJBMetaData` for the Cabin EJB could be used to get more information about the enterprise bean. Notice that there is no way to get the bean class using the `EJBMetaData`; the bean class is not part of the client API and therefore doesn't belong to the metadata.

```
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);
```

---

[3] Message-driven beans in EJB 2.0 don't have component interfaces and can't be accessed by Java RMI-IIOP.

4 The Reflection API is outside the scope of this book, but it is covered in *Java™ in a Nutshell*, by David Flanagan (O'Reilly).


```
EJBMetaData meta = c_home.getEJBMetaData();

System.out.println(meta.getHomeInterfaceClass().getName());
System.out.println(meta.getRemoteInterfaceClass().getName());
System.out.println(meta.getPrimaryKeyClass().getName());
System.out.println(meta.isSession());
```

This application creates output like the following:

```
com.titan.cabin.CabinHome
com.titan.cabin.Cabin
com.titan.cabin.CabinPK
false
```

In addition to providing the class types of the enterprise bean, the `EJBMetaData` also makes available the remote EJB home for the bean. By obtaining the remote EJB home from the `EJBMetaData`, we can obtain references to the remote EJB object and perform other functions. In the following code, we use the `EJBMetaData` to get the primary key class, create a key instance, obtain the remote EJB home, and from it, get a remote reference to the EJB object for a specific cabin entity:

```
Class primKeyType = meta.getPrimaryKeyClass();
If(primKetType instanceof java.lang.Integer){
   Integer pk = new Integer(1);

   Object ref = meta.getEJBHome();
   CabinHomeRemote c_home2 = (CabinHomeRemote)
   PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

   CabinRemote cabin = c_home2.findByPrimaryKey(pk);
   System.out.println(cabin.getName());
}
```

The HomeHandle

EJB 1.1 provides a new object called a `HomeHandle`, which is accessed by calling the `EJBHome.getHomeHandle()` method. This method returns a `javax.ejb.HomeHandle` object that provides a serializable reference to an enterprise bean's remote home. The `HomeHandle` allows a remote home reference to be stored and used later. It is similar to the `javax.ejb.Handle` and is discussed in more detail a little later.

Creating and finding beans

In addition to the standard `javax.ejb.EJBHome` methods that all remote home interfaces inherit, remote home interfaces also include special *create* and *find* methods for the bean. We have already talked about create and find methods, but a little review will solidify your understanding of the remote home interface's role in the Remote Client API. The following code shows the remote home interface defined for the Cabin EJB:


```
public interface CabinHomeRemote extends javax.ejb.EJBHome {
    public CabinRemote create(Integer id)
        throws CreateException, RemoteException;

    public CabinRemote findByPrimaryKey(Integer pk)
        throws FinderException, RemoteException;
}
```

Create methods throw a `CreateException` if something goes wrong during the creation process; find methods throw a `FinderException` if the requested bean can't be located. Since these methods are defined in an interface that subclasses `Remote`, they must also declare that they throw `RemoteException`.

The create and find methods are specific to the enterprise bean, so it is up to the bean developer to define the appropriate create and find methods in the remote home interface. `CabinHomeRemote` currently has only one create method that creates a cabin with a specified ID and a find method that looks up an enterprise bean given its primary key, but it's easy to imagine methods that would create and find a cabin with particular properties—for example, a cabin with three beds, or a deluxe cabin with blue wallpaper.

Only entity beans have find methods; session beans do not. Entity beans represent unique identifiable data within a database and therefore can be found. Session beans, on the other hand, do not represent data: they are created to serve a client application and are not persistent, so there is nothing to find. A find method for a session bean would be meaningless.

In EJB 2.0, the create methods were expanded so that a method name could be used as suffix. In other words, all create methods may take the form `create<SUFFIX>()`. For example, the Customer EJB might define a remote home interface with several create methods, each of which take a different String type parameters, but have different methods names.

```
public interface CustomerHome extends javax.ejb.EJBHome {

    public CustomerRemote createWithSSN(Integer id,
                                String socialSecurityNumber)
                throws CreateException, RemoteException;

    public CustomerRemote createWithPIN(Integer personalIdNubmer)
                throws CreateException, RemoteException;

    public CustomerRemote createWithBLN(Integer id,
                                String businessLicenseNumber)
                throws CreateException, RemoteException;

    public Customer findByPrimaryKey(Integer id)
    throws FinderException, RemoteException;
 }
```


While the use of a suffix in the create method names in EJB 2.0 is allowed it is not required. EJB 1.1 doesn't support the use of suffixes in create method names.

The create and find methods defined in the remote home interfaces are straightforward and can be easily employed by the client. The create methods on the home interface have to match the `ejbCreate()` methods on the bean class. `create()` and `ejbCreate()` match when they have the same parameters, when the arguments are of same type and in the same order, and when their method names are the same.

This way, when a client calls the create method on the home interface, the call can be delegated to the corresponding `ejbCreate()` method on the bean instance. The find methods in the home interface work similarly for bean-managed entities in EJB 2.0 and 1.1. Every `find<SUFFIX>()` method in the home interface must correspond to an `ejbFind<SUFFIX>()` method in the bean itself. Container-managed entities do not implement `ejbFind()` methods in the bean class; the EJB container supports find methods automatically. You will discover more about how to implement the `ebjCreate()` and `ejbFind()` methods in the bean in Chapters 6 and 8.

The Remote Interface

The business methods of an enterprise bean can be defined by the remote interface provided by the enterprise bean developer. The `javax.ejb.EJBObject` interface, which extends the `java.rmi.Remote` interface, is the base class for all remote interfaces.

The following code is the remote interface for the TravelAgent bean that we developed in Chapter 4:

```
public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public String [] listCabins(int shipID, int bedCount)
        throws RemoteException;
}
```

Figure 5-7 shows the `TravelAgentRemote` interface's inheritance hierarchy.

[FIGURE see modified figure 5-4]

Figure 5-4: Enterprise bean interface inheritance hierarchy

Remote interfaces are focused on the business problem and do not include methods for system-level operations such as persistence, security, concurrency, or transactions. System-level operations are handled by the EJB server, which relieves the client developer of many responsibilities. All remote interface methods for beans must throw, at the very least, a `java.rmi.RemoteException`, which identifies problems with distributed communications. In addition, methods in the remote interface can throw as many custom exceptions as needed to indicate abnormal business-related conditions or

errors in executing the business method. You will learn more about defining custom exceptions in Chapters 12 and 14.

Exercise 5.1, The remote component interfaces

EJBObject, Handle, and Primary Key

All remote interfaces extend the `javax.ejb.EJBObject` interface, which provides a set of utility methods and return types. These methods and return types are valuable in managing the client's interactions with beans. Here is the definition for the `EJBObject` interface:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
    public abstract EJBHome getEJBHome()
        throws RemoteException;
    public abstract Handle getHandle()
        throws RemoteException;
    public abstract Object getPrimaryKey()
        throws RemoteException;
    public abstract boolean isIdentical(EJBObject obj)
        throws RemoteException;
    public abstract void remove()
        throws RemoteException, RemoveException;
}
```

When the client obtains a reference to the remote interface, it is actually obtaining a remote reference to an EJB object. The EJB object implements the remote interface by delegating business method calls to the bean class; it provides its own implementations for the `EJBObject` methods. These methods return information about the corresponding bean instance on the server. As discussed in Chapter 2, the EJB object is automatically generated when deploying the bean in the EJB server, so the bean developer doesn't need to write an `EJBObject` implementation.

Getting the EJBHome

The `EJBObject.getEJBHome()` method returns a remote reference to the EJB home for the bean. The remote reference is returned as a `javax.ejb.EJBHome` object, which can be narrowed to the specific enterprise bean's remote home interface. This method is useful when an EJB object has left the scope of the remote EJB home that manufactured it. Because remote references can be passed as references and returned from methods, like any other Java object on the remote client, a remote reference can quickly find itself in a completely different part of the application from its remote home. The following code is contrived, but it illustrates how a remote reference can move out of the scope of its home and how `getEJBHome()` can be used to get a new reference to the EJB home at any time:

```
public static void main(String [] args) {
```



```
    try {
        Context jndiContext = getInitialContext();
        Object ref = jndiContext.lookup("TravelAgentHomeRemote");
        TravelAgentHomeRemote home = (TravelAgentHomeRemote)
            PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);

        // Get a remote reference to the bean (EJB object).
        TravelAgentRemote agent = home.create();
        // Pass the remote reference to some method.
        getTheEJBHome(agent);

        } catch (java.rmi.RemoteException re){re.printStackTrace();}
          catch (Throwable t){t.printStackTrace();}
}

public static void getTheEJBHome(TravelAgentRemote agent)
    throws RemoteException {

    // The home interface is out of scope in this method,
    // so it must be obtained from the EJB object.
    // EJB 1.0: Use native cast instead of narrow()
    Object ref = agent.getEJBHome();
    TravelAgentHomeRemote home = (TravelAgentHomeRemote)
        PortableRemoteObject.narrow(ref,TravelAgentHomeRemote.class);
// Do something useful with the home interface.
}
```

Primary key

`EJBObject.getPrimaryKey()` returns the primary key for an entity bean. This method is only supported by EJB objects that represent entity beans. Entity beans represent specific data that can be identified using this primary key. Session beans represent tasks or processes, not data, so a primary key would be meaningless. To better understand the nature of a primary key, we need to look beyond the boundaries of the client's view into the EJB container's layer, which was introduced in Chapters 2 and 3.

The EJB container is responsible for persistence of the entity beans, but the exact mechanism for persistence is up to the vendor. In order to locate an instance of a bean in a persistent store, the data that makes up the entity must be mapped to some kind of unique key. In relational databases, data is uniquely identified by one or more column values that can be combined to form a primary key. In an object-oriented database, the key wraps an object ID (OID) or some kind of database pointer. Regardless of the mechanism—which isn't really relevant from the client's perspective—the unique key for an entity bean's data is encapsulated by the primary key, which is returned by the `EJBObject.getPrimaryKey()` method.

The primary key can be used to obtain remote references to entity beans using the `findByPrimaryKey()` method on the remote home interface. From the

client's perspective, the primary key object can be used to identify a unique entity bean. Understanding the context of a primary key's uniqueness is important, as the following code shows:

```
Context jndiContext = getInitialContext()

Object ref = jndiContext.lookup("CabinHomeRemote");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

Cabin cabin_1 = home.create(101);
Integer pk = (Integer)cabin_1.getPrimaryKey();
Cabin cabin_2 = home.findByPrimaryKey(pk);
```

In this code, the client creates a Cabin EJB, retrieves its primary key and then uses the key to get a new reference to the same Cabin EJB. Thus, we have two variables, `cabin_1` and `cabin_2`, which are remote references to EJB objects. These both reference the same Cabin bean, with the same underlying data, because they have the same primary key.

The primary key must be used for the correct bean in the correct container. While this seems fairly obvious, the primary key's relationship to a specific container and home interface is important. The primary key can only be guaranteed to return the same entity if it is used within the container that produced the key. As an example, imagine that a third-party vendor sells the Cabin EJB as a product. The vendor sells the Cabin EJB to both Titan and to a competitor. Both companies deploy the entity bean using their own relational databases with their own data. An `Integer` primary key with value of 20 in Titan's EJB system will not map to the same Cabin entity in the competitor's EJB system. Both cruise companies have a Cabin bean with a primary key equal to 20, but they represent different cabins for different ships. The Cabin EJBs come from different EJB containers, so their primary keys are not equivalent. Every entity EJB object has a unique identity with its EJB home. If two EJB objects have the same home and same primary key, they are considered identical.

A primary key must implement the `java.io.Serializable` interface. This means that the primary key, regardless of its form, can be obtained from an EJB object, stored on the client using the Java serialization mechanism, and deserialized when needed. When a primary key is deserialized, it can be used to obtain a remote reference to that entity using `findByPrimaryKey()`, provided that the key is used on the right remote home interface and container. Preserving the primary key using serialization might be useful if the client application needs to access specific entity beans at a later date.

The following code shows a primary key that is serialized and then deserialized to reobtain a remote reference to the same bean:

```
// Obtain cabin 101 and set its name.
Context jndiContext = getInitialContext();
```



```
Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

Integer pk_1 = new Integer(101);
Cabin cabin_1 = home.findByPrimaryKey(pk_1);
cabin_1.setName("Presidential Suite");

// Serialize the primary key for cabin 101 to a file.
FileOutputStream fos = new FileOutputStream("pk101.ser");
ObjectOutputStream outStream = new ObjectOutputStream(fos);
outStream.writeObject(pk_1);
outStream.flush();
outStream.close();
pk_1 = null;

// Deserialize the primary key for cabin 101.
FileInputStream fis = new FileInputStream("pk101.ser");
ObjectInputStream inStream = new ObjectInputStream(fis);
Integer pk_2 = (Integer)inStream.readObject();
inStream.close();

// Re-obtain a remote reference to cabin 101 and read its name.
Cabin cabin_2 = home.findByPrimaryKey(pk_2);
System.out.println(cabin_2.getName());
```

Comparing beans for identity

The `EJBObject.isIdentical()` method compares two EJB object remote references. It's worth considering why `Object.equals()` isn't sufficient for comparing EJB objects. An EJB object is a distributed object stub and therefore contains a lot of networking and other state. As a result, references to two EJB objects may be unequal, even if they both represent the same unique bean. The `EJBObject.isIdentical()` method returns `true` if two EJB object references represent the same bean, even if the EJB object stubs are different object instances.

The following code shows how this might work. It starts by creating two remote references to the TravelAgent EJB. These remote EJB objects both refer to the same type of enterprise bean; comparing them with `isIdentical()` returns `true`. The two TravelAgent EJBs were created separately, but because they are stateless they are considered to be equivalent. If TravelAgent EJB had been a stateful bean (which it becomes in Chapter 12) the outcome would have been very different. Comparing two stateful beans in this manner will result in `false` because stateful beans have conversational state, which makes them unique. When we use `CabinHome.findByPrimaryKey()` to locate two EJB objects that refer to the same Cabin entity bean, we know the entity beans are identical, because we used the same primary key. In this case, `isIdentical()` also returns `true` because both remote EJB object references point to the same entity.


```
Context ctx  = getInitialContext();

Object ref = ctx.lookup("TravelAgentHomeRemote");
TravelAgentHomeRemote agentHome =(TravelAgentHomeRemote)
    PortableRemoteObject.narrow(ref, TravelAgentHomeRemote.class);

TravelAgentRemote agent_1 = agentHome.create();
TravelAgentRemote agent_2 = agentHome.create();
boolean x = agent_1.isIdentical(agent_2);
// x will equal true; the two EJB objects are equal.

ref = ctx.lookup("CabinHomeRemote");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref, CabinHomeRemote.class);

Integer pk_1 = new Integer(101);
Integer pk_2 = new Integer(101);
Cabin cabin_1 = c_home.findByPrimaryKey(pk_1);
Cabin cabin_2 = c_home.findByPrimaryKey(pk_2);
x = cabin_1.isIdentical(cabin_2);
// x will equal true; the two EJB objects are equal.
```

The Integer primary key used in the Cabin bean is simple. More complex custom defined primary keys require us to override `Object.equals()` and `Object.hashCode()` in order for the `EJBObject.isIdentical()` method to work. Chapter 9 discusses this the development of more complex custom primary keys, which are called *compound primary keys*.

Removing beans

The `EJBObject.remove()` method is used to remove the session or entity bean. The impact of this method is the same as the `EJBHome.remove()` method discussed previously. For session beans, `remove()` causes the session to be released and the remote EJB object reference to become invalid. For entity beans, the actual entity data is deleted from the database and the remote reference becomes invalid. The following code shows the `EJBObject.remove()` method in use:

```
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote c_home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

Integer pk = new Integer(101);
CabinRemote cabin = c_home.findByPrimaryKey(pk);
cabin.remove();
```

The `remove()` method throws a `RemoveException` if for some reason the reference can't be deleted.

The enterprise bean handle

The `EJBObject.getHandle()` method returns a `javax.ejb.Handle` object. The `Handle` is a serializable reference to the remote EJB object. This means that the client can save the `Handle` object using Java serialization and then deserialize it to reobtain a reference to the same remote EJB object. This is similar to serializing and reusing the primary key. The `Handle` allows us to recreate a remote EJB object reference that points to the same *type* of session bean or the same unique entity bean that the handle came from.

Here is the interface definition of the `Handle`:

```
public interface javax.ejb.Handle {
    public abstract EJBObject getEJBObject()
      throws RemoteException;
}
```

The `Handle` interface specifies only one method, `getEJBObject()`. Calling this method returns the remote EJB object from which the handle was created. Once you've gotten the object back, you can narrow it to the appropriate remote interface type. The following code shows how to serialize and deserialize the EJB `Handle` on a client:

```
// Obtain cabin 100.
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

Integer pk_1 = new Integer(101);
CabinRemote cabin_1 = home.findByPrimaryKey(pk_1);

// Serialize the Handle for cabin 100 to a file.
Handle handle = cabin_1.getHandle();
FileOutputStream fos = new FileOutputStream("handle100.ser");
ObjectOutputStream outStream = new ObjectOutputStream(fos);
outStream.writeObject(handle);
outStream.flush();
fos.close();
handle = null;

// Deserialize the Handle for cabin 100.
FileInputStream fis = new FileInputStream("handle100.ser");
ObjectInputStream inStream = new ObjectInputStream(fis);
handle = (Handle)inStream.readObject();
fis.close();

// Reobtain a remote reference to cabin 100 and read its name.

ref = handle.getEJBObject();
CabinRemote cabin_2 = (CabinRemote)
```



```
    PortableRemoteObject.narrow(ref, CabinRemote.class);

if(cabin_1.isIdentical(cabin_2))
    // this will always be true.
```

At first glance, the `Handle` and the primary key appear to do the same thing, but in truth they are very different. Using the primary key requires you to have the correct remote EJB home—if you no longer have a reference to the EJB remote home, you must look up the container using JNDI and get a new home. Only then can you call `findByPrimaryKey()` to locate the actual enterprise bean. The following code shows how this might work:

```
// Obtain the primary key from an input stream.
Integer primaryKey = (Integer)inStream.readObject();

// The JNDI API is used to get a root directory or initial context.
javax.naming.Context ctx = new javax.naming.InitialContext();

// Using the initial context, obtain the EJBHome for the Cabin bean.

Object ref = ctx.lookup("CabinHome");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

// Obtain a reference to an EJB object that represents the entity instance.
CabinRemote cabin_2 = CabinHome.findByPrimaryKey(primaryKey);
```

The `Handle` object is easier to use because it encapsulates the details of doing a JNDI lookup on the container. With a `Handle`, the correct EJB object can be obtained in one method call, `Handle.getEJBObject()`, rather than using the three method calls required to look up the context, get the home, and find the actual bean.

Furthermore, while the primary key can be used to obtain remote references to unique *entity* beans, it is not available for session beans; a handle can be used with either type of enterprise bean. This makes using a handle more consistent across bean types. Consistency is, of course, good in its own right, but it isn't the whole story. Normally, we think of session beans as not having identifiable instances because they exist for only the life of the client session, but this is not exactly true. We have mentioned (but not yet shown) stateful session beans, which retain state information between method invocations. With stateful session beans, two instances are not equivalent. A handle allows you to work with a stateful session bean, deactivate the bean, and then reactivate it at a later time using the handle.

A client could, for example, be using a stateful session bean to process an order when the process needs to be interrupted for some reason. Instead of losing all the work performed in the session, a handle can be obtained from the EJB object and the client application can be closed down. When the user is ready to continue the order, the handle can be used to obtain a reference to the stateful

session EJB object. Note that this process is not as fault tolerant as using the handle or primary key of an entity object. If the EJB server goes down or crashes, the stateful session bean will be lost and the handle will be useless. It's also possible for the session bean to time out, which would cause the container to remove it from service so that it is no longer available to the client.

Changes to the container technology can invalidate both handles and primary keys. If you think your container technology might change, be careful to take this limitation into account. Primary keys obtain EJB objects by providing unique identification of instances in persistent data stores. A change in the persistence mechanism, however, can impact the integrity of the key.

HomeHandle

The `javax.ejb.HomeHandle` is similar in purpose to `javax.ejb.Handle`. Just as the `Handle` is used to store and retrieve references to remote EJB objects, the `HomeHandle` is used to store and retrieve references to remote EJB homes. In other words, the `HomeHandle` can be stored and later used to access an EJB home's remote reference the same way that a `Handle` can be serialized and later used to access an EJB object's remote reference. The following code shows how the `HomeHandle` can be obtained, serialized, and used.

```
// Obtain cabin 100.
Context jndiContext = getInitialContext();

Object ref = jndiContext.lookup("CabinHome");
CabinHomeRemote home = (CabinHomeRemote)
    PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

// Serialize the HomeHandle for the cabin bean.
HomeHandle homeHandle = home.getHomeHandle();
FileOutputStream fos = new FileOutputStream("handle.ser");
ObjectOutputStream outStream = new ObjectOutputStream(fos);
outStream.writeObject(homeHandle);
outStream.flush();
fos.close();
homeHandle = null;

// Deserialize the HomeHandle for the cabin bean.
FileInputStream fis = new FileInputStream("handle.ser");
ObjectInputStream inStream = new ObjectInputStream(fis);
homeHandle = (HomeHandle)inStream.readObject();
fis.close();

EJBHome home = homeHandle.getEJBHome();
CabinHomeRemote home2 = (CabinHomeRemote)
    PortableRemoteObject.narrow(home,CabinHomeRemote.class);
```


Inside the Handle

Different vendors define their concrete implementations of the EJB handle differently. However, thinking about a hypothetical implementation of handles will give you a better understanding of how they work. In this example, we define the implementation of a handle for an entity bean. Our implementation encapsulates the JNDI lookup and use of the home's `findByPrimaryKey()` method so that any change that invalidates the key invalidates preserved handles that depend on that key. Here's the code for our hypothetical implementation of a `Handle`:

```
package com.titan.cabin;

import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.ejb.EJBObject;
import javax.ejb.Handle;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject

public class VendorX_CabinHandle
    implements javax.ejb.Handle, java.io.Serializable {

    private Integer primary_key;
    private String home_name;
    private Properties jndi_properties;

    public VendorX_CabinHandle(Integer pk, String hn, Properties p) {
        primary_key = pk;
        home_name = hn;
        jndi_properties = p;
    }

    public EJBObject getEJBObject() throws RemoteException {
      try {
        Context ctx = new InitialContext(jndi_properties);

       Object ref = ctx.lookup(home_name);
       CabinHomeRemote home =(CabinHomeRemote)
           PortableRemoteObject.narrow(ref,CabinHomeRemote.class);

       return home.findByPrimaryKey(primary_key);
      } catch (javax.ejb.FinderException fe) {
            throw new RemoteException("Cannot locate EJB object",fe);
      } catch (javax.naming.NamingException ne) {
            throw new RemoteException("Cannot locate EJB object",ne);
      }
    }
}
```


The `Handle` is less stable than the primary key because it relies on the networking configuration and naming—the IP address of the EJB server and the JNDI name of the bean's home—to remain stable. If the EJB server's network address changes or the name used to identify the home changes, the handle becomes useless.

In addition, some vendors choose to implement a security mechanism in the handle that prevents its use outside the scope of the client application that originally requested it. How this mechanism would work is unclear, but the security limitation it implies should be considered before attempting to use a handle outside the client's scope.

📖 Exercise 5.2, The EJBObject, Handles and Primary Key

EJB 2.0: The Local Client API

6

EJB 2.0 CMP: Basic Persistence

Overview

In Chapter 4, we started developing some simple enterprise beans, skipping over a lot of the details about developing enterprise beans. In this chapter, we'll take a thorough look at the process of developing entity beans. On the surface, some of this material may look familiar, but it is much more detailed and specific to entity beans.

Entity beans model business concepts that can be expressed as nouns. This is a rule of thumb rather than a requirement, but it helps in determining when a business concept is a candidate for implementation as an entity bean. In grammar school you learned that nouns are words that describe a person, place, or thing. The concepts of "person" and "place" are fairly obvious: a person EJB might represent a customer or a passenger, and a place EJB might represent a city or a port-of-call. Similarly, entity beans often represent "things": real-world objects like ships, credit cards, and so on. An EJB can even represent a fairly abstract "thing," such as a ticket or a reservation. Entity beans describe both the state and behavior of real-world objects and allow developers to encapsulate the data and business rules associated with specific concepts; a Customer EJB encapsulates the data and business rules associated with a customer, and so on. This makes it possible for data associated with a concept to be manipulated consistently and safely.

In Titan's cruise ship business, we can identify hundreds of business concepts that are nouns and therefore could conceivably be modeled by entity beans. We've already seen a simple Cabin EJB in Chapter 4, and we'll develop Customer and Address EJBs in this chapter. Titan could clearly make use of a Cruise EJB, a

Reservation EJB, and many others. Each of these business concepts represents data that needs to be tracked and possibly manipulated. Entities really represent data in the database, so changes to an entity bean result in changes to the database.

There are many advantages to using entity beans instead of accessing the database directly. Utilizing entity beans to objectify data provides programmers with a simpler mechanism for accessing and changing data. It is much easier, for example, to change a customer's name by calling `Customer.setName()` than to execute an SQL command against the database. In addition, objectifying the data using entity beans also provides for more software reuse. Once an entity bean has been defined, its definition can be used throughout Titan's system in a consistent manner. The concept of customer, for example, is used in many areas of Titan's business, including booking, scheduling, and marketing. A Customer EJB provides Titan with one complete way of accessing customer information, and thus it ensures that access to the information is consistent and simple. Representing data as entity beans makes development easier and more cost effective.

When a new EJB is created, a new record must be inserted into the database and a bean instance must be associated with that data. As the EJB is used and its state changes, these changes must be synchronized with the data in the database: entries must be inserted, updated, and removed. The process of coordinating the data represented by a bean instance with the database is called *persistence*.

There are two basic types of entity beans, and they are distinguished by how they manage persistence. *Container-managed persistence* beans have their persistence automatically managed by the EJB container. The container knows how a bean instance's persistent fields and relationships map to the database and automatically takes care of inserting, updating, and deleting the data associated with entities in the database. Entity beans using *bean-managed persistence* do all this work explicitly: the bean developer must write the code to manipulate the database. The EJB container tells the bean instance when it is safe to insert, update, and delete its data from the database, but it provides no other help. The bean instance does all the persistence work itself. Bean-managed persistence is covered in Chapter 10.

Container-managed persistence has undergone a dramatic change in EJB 2.0, which is so different that it's not backward compatible with EJB 1.1. For that reason, EJB 2.0 vendors must support both EJB 2.0's container-managed persistence model and EJB 1.1 container-managed persistence model. The EJB 1.1 model is supported purely so that application developers can migrate their existing applications to the new EJB 2.0 platform as painlessly as possible. It's expected that all new entity beans and new applications will use the EJB 2.0 container-managed persistence and not EJB 1.1 version. Although EJB 1.1 container-managed persistence is covered in this book, it should be avoided

unless you have a legacy EJB 1.1 system that you maintain. EJB 1.1 container-managed persistence is covered in Chapter 9.

The next three chapters focus on developing entity beans that use EJB 2.0 container-managed persistence. In EJB 2.0, the data associated with an entity bean can be much more complex than was possible in EJB 1.1 or EJB 1.0. In EJB 2.0, container-managed persistence entity beans can have relationships with other entity beans, which wasn't well supported in the older version. In addition, container-managed persistence entity beans can be finer in granularity so that they can easily model things like Address, LineItem, or Cabin.

This chapter develops two very simple entity beans, the Customer and Address EJBs, which will be used to explain how Enterprise JavaBeans 2.0 container-managed persistence entity beans are defined and operate at runtime. The Customer EJB has relationships with other several entities including address, phone, credit card, cruise, ship, cabin, and reservation EJBs. In the next few chapters, you'll learn how to leverage EJB 2.0's powerful support for entity bean-to-bean relationships as well as understanding their limitations. In addition, you will learn about the Enterprise JavaBeans Query Language (EJB QL) in Chapter 8, which is used to define how the find methods and the new select methods should behave at runtime.

It is common to refer to Enterprise JavaBeans 2.0 container-managed persistence as simply *CMP 2.0*. In the chapters that follow, we will use this abbreviation to distinguish between *CMP 2.0* and CMP 1.1 (Enterprise JavaBeans 1.1 container-managed persistence).

The abstract programming model

In *CMP 2.0*, entity beans have their state managed automatically by the container. The container will take care of enrolling the entity bean in transactions and persisting its state to the database. The enterprise bean developer describes the attributes and relationships of an entity bean using *virtual* persistent fields and relationship fields. They are called virtual fields because the bean developer does not declare these fields explicitly; instead, abstract assessor (get and set) methods are declared in the entity bean class. The implementations of these methods are generated at deployment time by EJB vendor's container tools. So it's important to remember that the terms *relationship field* and *persistent field* are referring to the abstract accessor methods and not to actual fields declared in the classes. This use of terminology is a convention in EJB 2.0 that you should become confortable with.

In Figure 6-1, the Customer EJB has four sets of accessor methods. The first two read and update the last and first names of the customer. These are examples of persistent fields; simple direct attributes of the entity bean. The other accessor methods obtain and set references to the Address EJB through its local interface, `Address`. This is an example of a relationship field called the `address` field.

[FIGURE (note 7-1 and 6-1 are the same figure)]

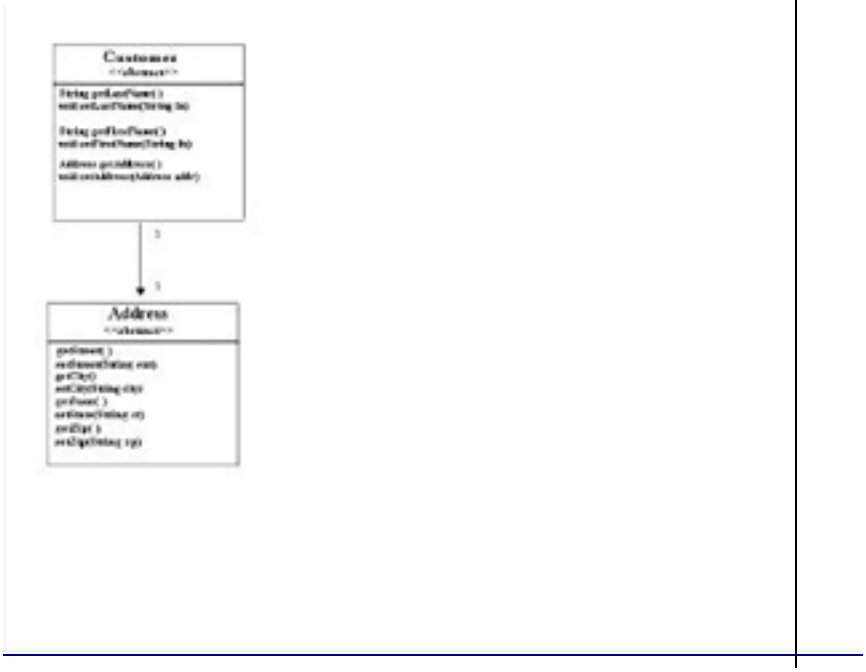


Figure 6-1 Class Diagram of Customer and Address EJBs

Abstract persistence schema

The *CMP 2.0* entity bean classes are defined using abstract accessor methods that represent virtual persistent and relationship fields. As already mentioned, the actual fields themselves are not declared in the entity classes. Instead, the characteristics of these fields are described in detail in the XML deployment descriptor used by the entity bean. The abstract persistence schema is the set of XML elements in the deployment descriptor that describe the relationship fields and the persistent fields. Together with the abstract programming model (the abstract accessor methods) and some help from the deployer, the container tool will have enough information to map the entity and its relationships with other entity beans in the database.

Container Tools & Persistence

One of the responsibilities of the vendor's container deployment tool is generating concrete implementations of the abstract entity beans. The concrete classes generated by the container tool are called *persistent classes*. Instances of the persistent classes will be responsible for working with the container to read and write data between the entity bean and the database at run time. Once the persistent classes are generated, they can be deployed into the EJB container. The container informs the *persistent instances* (instances of persistent

classes) when it's a good time to read and write data to the database. The persistent instances perform the reading and writing in a way that is optimized for the database being used.

The persistent classes will include database access logic tailored to a particular database. For example, an EJB product might provide a container that can map an entity beans to a specific database like the Oracle relational database or the POET object database. This specificity allows the persistent classes to employ native database optimizations particular to a brand or kind of database, schema, and configuration. Persistent classes may employ other optimizations like lazy loading and optimistic locking to further improve performance.

The container tool generates all the database access logic at deployment time, which it imbeds in the persistent classes. This means that the bean developers do not have to write this database access logic themselves, saving them a lot of work, and can also results in better performing entity beans because they are optimized implementations. As an entity bean developer, you will never have to deal with any database access code when working with *CMP 2.0* entities. In fact, you won't have access to the persistent classes that contain that logic because they are generated by container tool automatically. In most cases, the source code is not available to the bean developer.

Figures 7-2 and 7-3 show different container tools both of which are being used to map the Customer entity bean to a relational database.

[Figure 7-2 need screen shot]

BEA's Weblogic deployment tool

[Figure 7-3 need screen shot]

Sun Microsystem's J2EE RI deployment tool

The Customer EJB

In the following example we will develop a simple *CMP 2.0* entity bean, the Customer EJB. The Customer EJB models the concept of a cruise customer or passenger, but its design and use is applicable across many commercial domains.

As the chapter progresses the Customer EJB will be expanded and its complexity will increase to illustrate concepts discussed in each section. So this section serves only to introduce you to the entity bean and some basic concepts regarding its development, packaging and deployment. To simply things, we will skim over some concepts that are discussed in detail later in the chapter.

The Customer Table

Although *CMP 2.0* is database independent, the examples through out this book assume that you are using a relational database. For a relational database we will need a `CUSTOMER` table from which we get our customer data. The relational database table definition in SQL is as follows:

```
CREATE TABLE CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20)
)
```

The CustomerBean

The `CustomerBean` class is an abstract class that will be used by the container tool for generating concrete implementation, the persistent entity class, which will run in EJB container. The mechanism used by the container tool for generating a persistent entity class varies, but most vendors will generate a subclass of the abstract class provide by bean developer.

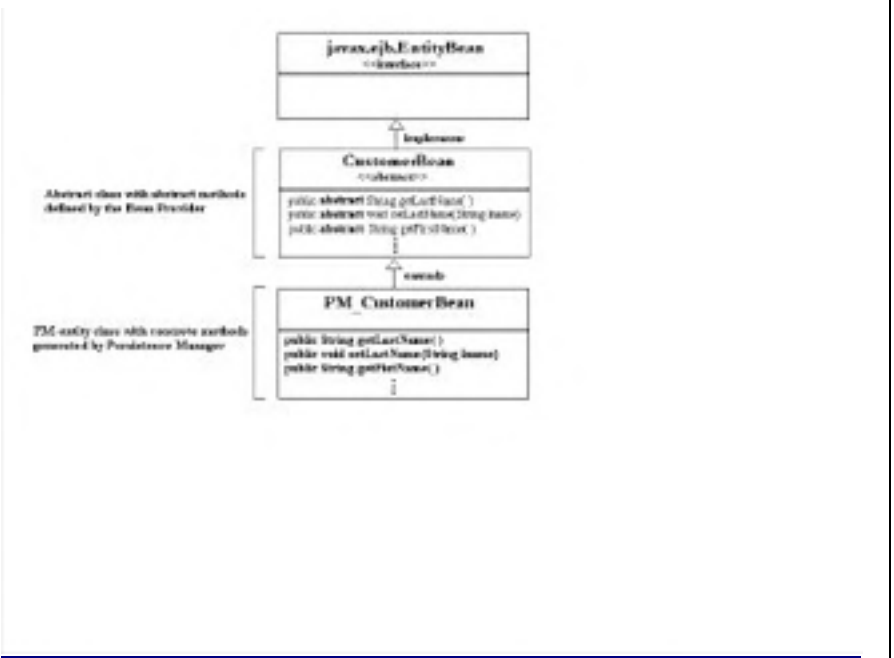


Figure 6-4

The container tool typically extends the bean class

The bean class must declare accessor (set and get) methods for each persistent and relationship field defined in the abstract persistence schema of the deployment descriptor. In truth, it's somewhat of a chicken-and-egg scenario, since the container tool needs both the abstract accessor methods (defined in the entity bean class) and the XML elements of the deployment descriptor to fully describe the bean's persistence schema. In this book, the entity bean class is always defined before the XML elements, because it's a more natural approach to developing entity beans.

Here is a very simple definition of the `CustomerBean` class which is developed and packaged for deployment by the bean developer.

```
import javax.ejb.EntityContext;

public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
    }

    // abstract accessor methods

    public abstract Integer getId();
    public abstract void setId(Integer id);

    public abstract String getLastName();
    public abstract void setLastName(String lname);

    public abstract String getFirstName();
    public abstract void setFirstName(String fname);

    // standard call back methods

    public void setEntityContext(EntityContext ec){}
    public void unsetEntityContext(){}
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void ejbRemove(){}
}
```

The `CustomerBean` class is defined as an abstract class. This is required by *CMP 2.0* to reinforce the idea that the `CustomerBean` is not deployed directly into the container system. Since abstract classes cannot be instantiated, the bean class must be subclassed by a persistence class generated by the deployment tool in order to be deployed. Also, the accessor methods are themselves

declared as `abstract`, which necessitates that container tool implement them and that the bean class declared `abstract`.

The `CustomerBean` extends the `javax.ejb.EntityBean` interface, which defines several callback methods including `setEntityContext()`, `unsetEntityContext()`, `ejbLoad()`, `ejbStore()`, `ejbActivate()`, `ejbPassivate()`, and `ejbRemove()`. These methods are important for notifying the bean instance about events in its life cycle, but they are not important to us at this point. We will discuss these methods in detail in Chapter 11.

The first method in the entity bean class is `ejbCreate()`, which takes a reference to an `Integer` object as its only argument. The `ejbCreate()` method is called when the remote client invokes the `create()` method on the entity bean's home interface. This concept should be familiar, since it's the same way `ejbCreate()` worked in the cabin bean developed in Chapter 4. The `ejbCreate()` method is responsible for initializing any persistent fields before the entity bean is created. In this first example, the `ejbCreate()` method is used to initialize the `id` persistent field, which is represented by the `setId()`/`getId()` accessor methods.

The return type of the `ejbCreate()` method is an `Integer` type, which is the *primary key* of the entity bean. The primary key is a unique identifier that can take a variety of forms, including wrappers for primitive types and custom-defined classes. The primary key in this case is an `Integer`, which is mapped to the `ID` field in the `CUSTOMER` table. This will become more evident when we define the XML deployment descriptor. Although the return type of the `ejbCreate()` method is the primary key, the value that is actually returned by the `ejbCreate()` method is `null`. The EJB container and persistence class will take care to extract the primary key from the bean when needed. The reason `ejbCreate()` has a return type is the result of a decision in EJB 1.1 that is explained in the side bar, *Why ejbCreate() returns null*.

Why ejbCreate() returns null

In EJB 1.0, the first release of EJB, the `ejbCreate()` method in container managed persistence was declared as returning `void`, but it was changed to the primary key types in EJB 1.1 with an actual return value of `null`.

EJB 1.1 changed its return value from `void` to the primary key type to facilitate subclassing; the change was made so that it's easier for a bean-managed entity bean to extend a container-managed entity bean. In EJB 1.0, this is not possible because Java doesn't allow you to overload methods with different return values. By changing this definition so that a bean-managed entity bean can extend a container-managed entity bean, the EJB 1.1 allowed vendors to support container-managed persistence by extending the container-managed bean with a generated bean-managed bean—a fairly simple solution to a difficult problem.

With the introduction of *CMP 2.0*, this little trick is not as useful to EJB vendors as it once was. The abstract persistence schema of EJB *CMP 2.0* beans is, in many cases, too complex for a simple BMP container. However, it remains a part of the programming model for backward compatibility and to facilitate bean-managed persistence subclassing if needed.

The `ejbPostCreate()` method is used to perform initialization after the entity bean is created, but before it services any requests from the client. Usually this method is used to perform work on the entity bean's relationship fields, which can only occur after the bean's `ejbCreate()` method is invoked and it's added to the database. For each `ejbCreate()` method there must be a matching `ejbPostCreate()` method that has the same method name and arguments, but returns a `void`. This pairing of `ejbCreate()` and `ejbPostCreate()` ensures that the container calls the correct methods together. We'll explore the use of the `ejbPostCreate()` in more detail later, for now it's not needed, so its implementation is left empty.

The abstract accessor methods represent the persistent fields in the `CustomerBean` class. These methods are defined as `abstract` without method bodies. As was already mentioned, when the bean is processed by a container tool, these methods will be implemented by a *persistence class* based on the abstract persistence schema (XML deployment descriptor elements), the particular EJB container and the database used. Basically these method fetch

and update values in the database and are not implemented by the bean developer.

The Remote Interface

For the Customer EJB we will need a `CustomerRemote` remote interface, because the bean will be accessed by clients outside the container system. The remote interface defines the business methods that clients will use to interact with the entity bean. The remote interface should define methods that model the public aspects of the business concept being modeled—those behaviors and data that should be exposed to client applications. Here is the remote interface for `CustomerRemote`:

```
import java.rmi.RemoteException;

public interface CustomerRemote extends javax.ejb.EJBObject {

    public String getLastName() throws RemoteException;
    public void setLastName(String lname) throws RemoteException;

    public String getFirstName() throws RemoteException;
    public void setFirstName(String fname) throws RemoteException;
}
```

Any methods defined in the remote interface must match the signatures of methods defined in the bean class. In this case, several accessor methods in the `CustomerRemote` interface match persistent field accessor methods in the `CustomerBean` class. When the remote interface methods match the persistent field methods, the client has direct access to the entity bean's persistent fields.

You are not required to match abstract accessor methods in the bean class with methods in the remote interface. In fact, it's recommended that the remote interface be as independent of the abstract programming model as possible. Notice that the remote interface does not define `getId()` and `setId()` methods, as does the `CustomerBean` class. While remote methods can match persistent fields in the bean class, the specification prohibits the remote methods from matching relationship fields, which access other entity beans.

The Remote Home interface

The remote home interface of any entity bean is used to create, locate, and remove entities from the EJB container. Each entity bean type may have its own remote home interface, or a local home interface or both. As you learned in chapter 5, the remote and local home interfaces perform essentially the same function. The home interfaces define three basic kinds of methods: home business methods, zero or more `create()` methods and one or more find

methods.[1] The `create()` methods act like remote constructors and define how new entity beans are created. In our remote home interface, we only provide a single `create()` method, which matches the corresponding `ejbCreate()` method in the bean class. The find method is used to locate a specific Customer EJB using the primary key as a unique identifier.

The following code contains the complete definition of the `CustomerHomeRemote` interface:

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface CustomerHomeRemote extends javax.ejb.EJBHome {

    public Customer create(Integer id)
    throws CreateException, RemoteException;

    public Customer findByPrimaryKey(Integer id)
    throws FinderException, RemoteException;

}
```

A `create()` method may be suffixed with a name in order to further qualify it when overloading method arguments. This is useful if you have two `create()` methods that take different arguments of the same type. For example, we could declare two `create()` methods for Customer which both declare an `Integer` argument. The `Integer` argument might be a social security number (SSN) in one case and a tax identification number (TIN) in another—individuals have social security numbers while corporations have tax identification number.

```
public interface CustomerHomeRemote extends javax.ejb.EJBHome {

    public Customer createWithSSN(Integer id,
                                  String socialSecurityNumber)
                   throws CreateException, RemoteException;

    public Customer createWithTIN(Integer id,
                                  String taxIdentificationNumber)
                   throws CreateException, RemoteException;

    public Customer findByPrimaryKey(Integer id)
    throws FinderException, RemoteException;
 }
```

The use of suffixes is useful when you need `create()` methods to be more descriptive, or need to further qualify them for method overloading. Each

1 Chapter 15 explains when you should not define any create methods in the home interface.

`create<SUFFIX>()` method must have a corresponding `ejbCreate<SUFFIX>()` in the bean class. For example, the `CustomerBean` class would need to define a `ejbCreateWithSSN()` and `ejbCreateWithTIN()` methods. We are keeping this example simple, so we only need one `create()` method and therefore, no suffix.

Enterprise JavaBeans specifies that `create()` methods in the remote home interface must throw the `javax.ejb.CreateException`. In the case of container-managed persistence, the container needs a common exception for communicating problems experienced during the create process.

Entity remote home interfaces must define a `findByPrimaryKey()` method which takes the entity bean's primary key type as its only argument, but a matching method is not defined in the entity bean class. The implementation of the `findByPrimaryKey()` is generated automatically by the deployment tool. At runtime the `findByPrimaryKey()` method will automatically locate and return a remote reference to the entity bean with the matching primary key.

Other find methods can also be declared by the bean developer. For example, the `CustomerHomeRemote` interface could define a `findByLastName(String lnam)` method, which locates all the Customer entities with the specified last name. These types of finder methods are implemented by the deployment automatically based on the method signature and an EJB-QL statement, which is similar to SQL but is specific to EJB. Custom finder methods and EJB-QL are discussed in detail in Chapter 8.

The XML Deployment Descriptor

All *CMP 2.0* entity beans must be packaged for deployment with an XML deployment descriptor that describes the bean and its abstract persistence schema. In most cases the bean developer is not directly exposed to the XML deployment descriptor, but will use container's visual deployment tools to package beans. It is convention in this book, however, to describe the declarations of the deployment descriptor in detail so that you have a full understanding of their content and organization.

The XML deployment descriptor, for our simple Customer EJB, contains many elements that are familiar to you from chapter 4. The elements specific to entity beans and persistence are most important to us in this chapter. The following is the complete XML deployment descriptor for the Customer EJB.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
```



```
            <ejb-name>CustomerEJB</ejb-name>
            <home>com.titan.customer.CustomerHomeRemote</home>
            <remote>com.titan.customer.CustomerRemote</remote>
            <ejb-class>com.titan.customer.CustomerBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>lastName</field-name></cmp-field>
            <cmp-field><field-name>firstName</field-name></cmp-field>
            <primkey-field>id</primkey-field>
            <security-identity><use-caller-identity/></security-identity>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
        <security-role>
            <role-name>Employees</role-name>
        </security-role>
        <method-permission>
            <role-name>Employees</role-name>
            <method>
                <ejb-name>CustomerEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <container-transaction>
            <method>
                <ejb-name>CustomerEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        <container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

The first few elements, which declare the Customer EJB name, (`CustomerEJB`) as well as its home, remote, and bean class, should already be familiar to you from Chapter 4. The `<security-identity>` element was covered in Chapter 3.

The `<assembly-descriptor>` elements, which declare the security and transaction attributes of the bean, were also covered briefly in chapter 4. Basically all employees can access any `CustomerEJB` method and all methods use the `Required` transaction attribute.

Container managed persistence entities also need to declare their persistence type, version, and whether they are reentrant. These elements are declared under the entity element.

The <persistence-type> tells the container system whether the bean will be a container-managed persistence entity or a bean-managed persistence entity. In this case it's container-managed, so we use Container. Had it been bean-managed persistence, the value would have been Bean.

The <cmp-version> tells the container system which version of container-managed persistence is being used. Enterprise JavaBeans 2.0 containers must support the new container-managed persistence model as well as the old one defined in Enterprise JavaBeans 1.1. This is required for backward compatibility, so that organizations can migrate to EJB 2.0 without having to redefine all their established container-managed persistence entity beans at once. The value of the <cmp-version> element can be either 2.x or 1.x for versions EJB 2.0 and EJB 1.1 respectively. The <cmp-version> element is optional. If its not declared, the default value is 2.x, so its not really needed here but it's specified as an aid to other developers who are reading the deployment descriptor.

The <reentrant> element indicates whether reentrant behavior or loop-backs are allowed. In this case the value is False, which indicates that the CustomerEJB is not reentrant. A value of True would indicate that the CustomerEJB is reentrant. Reentrant behavior was covered in chapter 3.

The entity bean will also declare its container managed persistence fields and its primary key.

```
<entity>
    <ejb-name>CustomerEJB</ejb-name>
    <home>com.titan.customer.CustomerHomeRemote</ejb-home>
    <remote>com.titan.customer.CustomerRemote</ejb-remote>
    <ejb-class>com.titan.customer.CustomerBean</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Integer</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-version>2.x</cmp-version>
    <cmp-field><field-name>id</field-name></cmp-field>
    <cmp-field><field-name>lastName</field-name></cmp-field>
    <cmp-field><field-name>firstName</field-name></cmp-field>
    <primkey-field>id</primkey-field>
</entity>
```

The container-managed persistent fields are the id, lastName, and firstName as indicated by the <cmp-field> elements. The <cmp-field> elements must have matching accessor methods in the CustomerBean class. As you can see from the following table, the values declared in the <cmp-field> match the names of abstract accessor methods we declared in the CustomerBean class—the get and set part of the method names are ignored when matching methods to <cmp-field> declarations.

Cmp-field	Abstract accessor method
id	public abstract Integer getId() public abstract void setId(Integer id)
lastName	public abstract String getLastName() public abstract void setLastName(String lname)
firstName	public abstract String getFirstName() public abstract void setFirstName(String lname)

CMP 2.0 requires that the <cmp-field> values start with a lower case letter while its matching accessor methods take the form get<cmp-field value>(), set<cmp-field value>() where the first letter of the <cmp-field> is capitalized. The return type of the get method and the parameter of the set method determine the type of the <cmp-field>. It's the convention of this book, but not a requirement of CMP 2.0, that field names with multiple words are declared using "camel case", where each new word starts with a capital letter (e.g. lastName).

Finally, we declare the primary key using two fields, the <prim-key-class> and the <primkey-field>. The <prim-key-class> indicates the type of the primary key and the <primkey-field> indicates which of the <cmp-fields> elements designates the primary key. This is an example of *single-field* primary key, where only one field of the entity beans container managed fields describes a unique identifier for the bean. In many cases a compound primary key, which uses more then one of the persistent fields as a key, is used. In addition, an unknown primary key may be defined; unknown keys use a field that may not be declared in the bean at all. The different types of primary keys are covered in more detail in Chapter 11, *Entity-Container Contract*.

The EJB JAR file

Now that you have created the interfaces, bean class, and deployment descriptor, you're ready to package the bean for deployment. As you learned in Chapter 4, the JAR file provides a way to "shrink-wrap" a component so that it can be sold and or deployed in an EJB container. The examples available from *http://www.oreilly.com* contain a properly prepared JAR file that includes the Customer EJB's interfaces, bean class, and deployment descriptor. You may use these files or develop them yourself. The command for creating a new EJB JAR file is:

```
\dev % jar cf customer.jar com/titan/customer/*.class
com/titan/customer/META-INF/ejb-jar.xml
```



```
F:\..\dev>jar cf cabin.jar com\titan\customer\*.class com\titan\customer
\META-INF\ejb-jar.xml
```

Most EJB servers provide graphical or command line tools that will create the XML deployment descriptor and package the enterprise bean into a JAR file automatically. Some of these tools will even create the home and remote interfaces automatically, based input from the developer. If you prefer to use these tools, the workbooks will step you through the process of deploying an entity bean using specific vendor's container deployment tools.

Deployment

Once the `CustomerEJB` is packaged in a JAR file, it's ready to be processed by the deployment tools. For most vendors these tools will be combined into one graphical user interface used at deployment time. The point is to map the container-managed persistence fields of the bean to fields of data objects in the database. Figures 7-2 and 7-3 show visual tools used to map the Customer EJB's persistent fields.

In addition, the security roles need to be mapped to the subjects in the security realm of the target environment and the bean needs to be added to the naming service and given a JNDI lookup name (name binding). These tasks are also accomplished using the deployment tools provided by your vendor. The workbooks provide step-by-step instructions for deploying the `CustomerEJB` in specific vendor environments.

The Client application

The client application is a remote client to the `CustomerEJB`, which will create several customers, find them, and then remove them. The following is the complete definition of the `Client` application.

```
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Properties;

public class Client {
    public static void main(String [] args) throws Exception {
            // obtain CustomerHome
            Context jndiContext = getInitialContext();
            Object obj=jndiContext.lookup("CustomerEJB");
            CustomerHomeRemote home = (CustomerHomeRemote)
            javax.rmi.PortableRemoteObject.narrow(obj,
                                        CustomerHomeRemote.class);

            // create Customers
```

```
            for(int i = 0; i < args.length;i++){
                Integer primaryKey = new Integer(args[i]);
                String firstName = args[++i];
                String lastName = args[++i];
                CustomerRemote customer = home.create(primaryKey);
                customer.setFirstName(firstName);
                customer.setLastName(lastName);
            }
            // find and remove Customers
            for(int i = 0; i < args.length;){
                Integer primaryKey = new Integer(args[i]);
                CustomerRemote customer
                                = home.findByPrimaryKey(primaryKey);
                String lastName = customer.getLastName( );
                String firstName = customer.getFirstName( );
                System.out.print(primaryKey+" = ");
                System.out.println(firstName+" "+lastName);

                // remove Customer
                customer.remove();
            }
    }

    public static Context getInitialContext()
                        throws javax.naming.NamingException {
        Properties p = new Properties();
        // ... Specify the JNDI properties specific to the vendor.
        //return new javax.naming.InitialContext(p);
        return null;
    }
}
```

The client application creates several Customer EJBs, sets their first and last names, prints out the persistent field values, and then removes the entities from the container system, and effectively the database.

📖 Exercise 6.1, Deploying the Customer EJB

Persistent Fields

Container-managed persistent fields are those virtual fields whose values map directly to the database. Persistent fields can be Java serializable types and Java primitive types.

The Java serializable types can be any class that implements the `java.io.Serializable` interface. Most deployment tools will handle `java.lang.String`, `java.util.Date` and the primitive wrappers (`Byte`, `Boolean`, `Short`, `Integer`, `Long`, `Double`, and `Float`) easily, because

these types of objects are part of the Java core and map naturally to fields in relational and other databases. The `CustomerEJB` declares three serializable fields `id`, `lastName`, and `firstName`, which map naturally to the `INT` and `CHAR` fields of the `CUSTOMER` table in the database.

You can also define your own serializable types, called *dependent values classes*, and declare them as container-managed persistent fields. However, arbitrary dependent values classes usually will not map naturally to database types, so they must be stored in their serializable form in some type of binary database field. Serializable objects are always returned as copies and not references, so a change to a serializable object will not impact its database value. The entire value must be updated using the abstract `set<FIELD-NAME>` method. This is normally not an issue with `String`, `Date`, and the primitive wrappers types since they are immutable objects. This book recommends that you don't use custom serializable objects as persistent field types unless it's absolutely necessary.

The primitive types (`byte`, `short`, `int`, `long`, `double`, `float` and `boolean`) are also allowed to be container-managed persistence fields. These types are easily mapped to the database and are supported by all deployment tools. As an example, the `CustomerEJB` might declare a `boolean` that represents a customer's credit worthiness.

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }

    // abstract accessor methods
    public abstract boolean getHasGoodCredit( );
    public abstract void setHasGoodCredit(boolean creditRating);
```

Dependent value classes

As discussed in the previous section, dependent values classes are custom serializable objects, which can be used as persistent fields -- although its not recommended. However, dependent values classes are valuable for packaging data and moving it between an entity bean and its clients. Dependent values classes can separate the client's view of the entity bean from its abstract persistent model, which makes it easier for the entity bean class to change without impacting existing clients.

The remote and local interface methods of an entity bean should be defined independently of the anticipated abstract persistent schema. In other words, you should design the remote interfaces to model the business concepts, not the underlying persistent programming model. Dependent value classes can help separate the client's view from the persistence model by providing objects that fill the gaps in these perspectives. Dependent value classes are used a lot in remote interfaces where packaging data together can reduce network traffic, but they are also useful in local interfaces.

For example, the `CustomerEJB` could be modified so that its `lastName` and `firstName` fields are not exposed directly to remote clients through their accessor methods. This is a reasonable design approach, since most clients access the entire name of the customer at once. In this case, the remote interface might modified to look as follows:

```
import java.rmi.RemoteException;

public interface CustomerRemote extends javax.ejb.EJBObject {

    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;

}
```

The remote interface here is simpler than the one we saw earlier. It allows the remote client to get all the name information in one method call instead of two—this reduces network traffic and improves performance for remote clients. The use of the `Name` dependent value is also semantically more consistent with how the client interacts with the Customer EJB, which is useful in both remote and local interfaces.

To implement these interfaces, the `CustomerBean` class adds a business method that matches the remote interface methods. The `setName()` method updates the `lastName` and `firstName` fields, while the `getName()` method constructs a `Name` object from these fields.

```
import javax.ejb.EntityContext;

public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
    }
    // business methods
    public Name getName( ){
        Name name = new Name(getLastName(),getFirstName());
        return name;
```



```
    }
    public void setName(Name name){
        setLastName(name.getLastName());
        setFirstName(name.getFirstName());
    }
    // abstract accessor methods

    public abstract String getLastName();
    public abstract void setLastName(String lname);

    public abstract String getFirstName();
    public abstract void setFirstName(String fname);
```

This is a good example of how dependent value classes can be used to separate the client's view from the abstract persistence schema.

The `getName()` and `setName()` methods are not abstract persistence methods, they are business methods. Entity beans can have as many business methods as needed. Business methods introduce business logic to the Customer EJB; otherwise the bean would only be a data wrapper. For example, validation logic could be added to the `setName()` method to ensure that the data is correct before applying the update. In addition, the entity bean class can use other methods that help with processing data—these are just instance methods and may not be exposed as business methods in the remote interface.

How dependent value classes are defined is important to understanding how they should be used. The `Name` dependent values class is defined as follows:

```
public class Name implements java.io.Serializable {
    private String lastName;
    private String firstName;

    public Name(String lname, String fname){
        lastName = lname;
        firstName = fname;
    }
    public String getLastName() {
        return lastName;
    }
    public String getFirstName() {
        return firstName;
    }
}
```

You'll notice that `Name` dependent values class has `get` accessor methods but not `set` methods. It's immutable. This is a design strategy used in this book and is not a requirement of the specification; CMP 2.0 does not specify how dependent value classes are defined.

We make dependent values immutable so that clients cannot change the `Name` object's fields. The reason is quite simple: the `Name` object is a copy, not a remote reference. Changes to `Name` objects are not reflected in the database. Making the `Name` immutable helps to ensure that clients do not mistake this dependent value for a remote object reference, thinking that a change to the `Name` object is automatically reflected on the database. To change the customer's name, the client is required to create a new `Name` object and use the `setName()` method to update the Customer EJB.

The following code listing from illustrates how a client would modify the name of a customer using the `Name` dependent values class.

```
// find Customer
customer = home.findByPrimaryKey(primaryKey);
name = customer.getName();
System.out.print(primaryKey+" = ");
System.out.println(name.getFirstName( )+" "+name.getLastName( ));

// change customer's name
name = new Name("Monson-Haefel", "Richard");
customer.setName(name);
name = customer.getName();
System.out.print(primaryKey+" = ");
System.out.println(name.getFirstName( )+" "+name.getLastName( ));
```

The output will look as follows:

```
1 = Richard Monson
1 = Richard Monson-Haefel
```

Defining the bean's interfaces according to the business concept and not the underlying data is not always reasonable, but you should try to employ this strategy when the underlying data model doesn't clearly map to the business purpose or concept being modeled by the entity bean. The bean's interfaces may be used by developers who know the business, and not the abstract programming model. It is important to them that the entity beans reflect the business concept. In addition, defining the interfaces independent of the persistence model enables the component interfaces and persistence model to evolve separately. This is important because it allows the abstract persistent programming model to change over time; it also allows for new behavior to be added to the entity bean as needed.

While the dependent values classes serve a purpose, they *should not* be used indiscriminately. In many cases it would be foolish to use dependent values classes when the container-managed persistent field will do just fine. For example, checking a client's credit worthiness before processing an order can be accomplished easily using the `getHasGoodCredit()` method directly. In this case a dependent object class would serve no purpose.

📖 Exercise 6.2, Using Dependent value classes

Relationship Fields

Entity beans can form relationships with other entity beans. In figure 6-1, at the beginning of this chapter, the Customer EJB is shown to have a one-to-one relationship with the Address EJB. The Address EJB is a fine-grained business object that should always be accessed in the context of another entity bean, which means it should only have local interfaces and not remote interfaces. An entity bean can have relationships with many different entity beans at the same time. For example, we could easy add relationship fields for Phone, CreditCard and other entity beans. At this point, however, we choose to keep the Customer EJB simple.

Following Figure 7-1 as guide we define the Address EJB as follows.

```
public abstract class AddressBean
extends javax.ejb.EntityBean {

    public Object ejbCreateAddress
                        (String street, String city,
                         String state,  String zip )
    {
         setStreet(street);
         setCity(city);
         setState(state);
         setZip(zip);
         return null;
    }
    public void ejbPostCreateAddress
                    (String street, String city,
                     String state,  String zip){
    }

    // persistent fields
    public abstract String getStreet();
    public abstract void setStreet(String street);
    public abstract String getCity();
    public abstract void setCity(String city);
    public abstract String getState();
    public abstract void setState(String state);
    public abstract String getZip();
    public abstract void setZip(String zip);

    // standard call back methods

    public void setEntityContext(EntityContext ec){}
    public void unsetEntityContext(){}
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbActivate(){}
```



```
    public void ejbPassivate(){}
    public void ejbRemove(){}

}
```

The `AddressBean` class defines an `ejbCreateAddress()` method that is called when a new Address EJB is created as well as several persistent fields (`street`, `city`, `state`, and `zip`). The persistent fields are represented by the abstract accessor methods, which is the idiom required for persistent fields in all entity bean classes. These abstract accessor methods are matched with their own set of XML deployment descriptor elements which define the abstract persistent schema of the Address EJB. At deployment time the container's deployment tool will map the Customer EJB's persistent fields and the Address EJB's persistent fields to the database. This means that there must be a table in our relational database that contains columns that match the persistent fields in the Address EJB. In this example we will use a separate `ADDRESS` table for storing address information, but the data could just as easily been declared in other table.

```
CREATE TABLE ADDRESS
(
 ID INT PRIMARY KEY,
 STREET CHAR(40),
 CITY CHAR(20),
 STATE CHAR(2),
 ZIP CHAR(10)
)
```

You'll have noticed that the table includes a column that has no corresponding persistent field in the Address EJB, the `ID` column. Entity beans do not have to define all of the columns from corresponding tables, as persistent fields. In fact, an entity bean may not even have a single corresponding table; it may be persisted to several tables. The bottom line is that the container's deployment tool allows the abstract persistence schema of entity beans to be mapped to a database in a variety of ways, allowing a clean separation between the persistent classes and the database. In this case the `ID` column is an auto-increment field, which is created automatically by the database or container system. It serves the primary key of the Address EJB and is not part of the bean's abstract persistence schema. It's invisible.

In addition to the bean class, we will also define the local interface for the Address EJB, which allows it to be accessed by other entity beans (namely the Customer EJB) within the same address space or process.

```
// Address EJB's local interface
public interface AddressLocal extends javax.ejb.EJBLocalObject {
  public String getStreet();
  public void setStreet(String street);
  public String getCity();
  public void setCity(String city);
```



```
  public String getState();
  public void setState(String state);
  public String getZip();
  public void setZip(String zip);
}

// Address EJB's local home interface
public interface AddressLocalHome extends javax.ejb.EJBLocalHome {
  public AddressLocal create(String street, String city,
                             String state,  String zip )
                      throws javax.ejb.CreateException;
  public AddressLocal findByPrimaryKey(Object primaryKey)
                      throws javax.ejb.FinderException;

}
```

You may have noticed that the `ejbCreate()` method of the `AddressBean` class and the `findByPrimaryKey()` method of the home interface both define the primary key type as `java.lang.Object` instead of `java.lang.Integer`. When a primary key type is defined as an Object type, it's said to be undefined, which means the exact type of key used is not known until the bean is deployed. In this case, an undefined type allows us to use the auto-increment facilities of the native database. If we were to define the primary key type, then we would have to set the primary key value in the `ejbCreate()` method, which would make it impossible to use auto-increment for the `id` field. This is a concept that is explored in detail in Chapter 11.

The relationship field for the Address EJB is defined in the `CustomerBean` class using an abstract accessor method, the same way that persistent fields are declared. In the following code the `CustomerBean` has been modified to include the Address EJB as a relationship field.

```
import javax.ejb.EntityContext;
import javax.ejb.CreateException;

public abstract class CustomerBean implements javax.ejb.EntityBean {
    ...

    // persistent relationships
    public abstract AddressLocal getHomeAddress();
    public abstract void setHomeAddress(AddressLocal address);

    // persistent fields
    public abstract boolean getHasGoodCredit( );
    public abstract void setHasGoodCredit(boolean creditRating);
    ...
```

The `getHomeAddress()` and `setHomeAddress()` accessor methods are self-explanatory; they allow the bean to access and modify its `homeAddress` relationship. These accessor methods represent a *relationship field*, which is a

virtual field that references another entity bean. The name of the accessor method is determined by the name of the relationship field, as declared in the XML deployment descriptor. In this case we have named the customer's address `homeAddress`, so the corresponding accessor method names will be `getHomeAddress()` and `setHomeAddress()`.

To accommodate the relationship between the Customer EJB and the home address a foreign key, `ADDRESS_ID`, will be added to the `CUSTOMER` table that points to the `ADDRESS` record. In practice this schema is actually the reverse of what is usually done, where the ADDRESS table contains a foreign key to the CUSTOMER table. However, the schema used here is useful in demonstrating alternative database mappings and is utilized again in Chapter 7.

```
CREATE TABLE CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20),
 ADDRESS_ID INT
)
```

When a new Address EJB is created and set as the Customer EJB's `homeAddress` relationship, the Address EJB's primary key will be placed in the `ADDRESS_ID` column of the `CUSTOMER` table creating a relationship in the database. In other words, it's the act of setting the relationship field that creates the relationship between the beans.

```
// get local reference
AddressLocal address = ……

// establish the relationship
setHomeAddress(address);
```

To give the Customer a home address we will need to deliver the address information to the Customer. This appears to be a simple matter of declaring matching `setHomeAddress()`/`getHomeAddress()` in the remote interface, but it's not! While it's valid to make persistent fields directly available to clients, persistent relationships are more complicated.

The remote interface of a bean is not allowed to expose its relationship fields *if* the relationship references another bean's local interface. In the case of the `homeAddress` field we have declared the type to be `AddressLocal`, which is a local interface, so the `setHomeAddress()/getHomeAddress()` assessors cannot be declared in the remote interface of the Customer EJB.

Remote interfaces *may*, however, expose relationship fields that use remote interface types. So, for example, if we had declared the `homeAddress` field as a remote interface (an interface that extends `javax.ejb.EJBObject`), we could expose that relationship field in the remote interface of the Customer EJB.

The reason for this restriction on remote interfaces is fairly simple: The `EJBLocalObject`, which implements the local interface, is optimized for use within the same address space or process as the client, and is not capable of being used across the network. In other words, references that implement the local interface of a bean cannot be passed across the network, so it cannot declared as a return type of a parameter of a remote interface.

We take advantage of the `EJBLocalObject` optimization for better performance, but that same advantage limits location transparency; we must only use it within the same address space.

Local interfaces (an interface that extends `javax.ejb.EJBLocalObject`) on the other hand, can expose any kind of relationship field regardless of whether it's a remote or local interface. With local interfaces, the caller and the enterprise bean being called are located in the same address space, so they can pass around local references without a problem. So for example, if we had defined a local interface for the Customer EJB, it could include a method that allows local clients to access its Address relationship directly.

```
public interface CustomerLocal extends javax.ejb.EJBLocalObject {
    public AddressLocal getHomeAddress( );
    public void setHomeAddress(AddressLocal address);
}
```

Unlike local interfaces, remote interfaces can be used as return values or parameters in the methods of both remote and local interfaces because remote interfaces are location transparent. The networking capabilities of a remote interface reference work within the same address space as easily as across address spaces.

When it comes to the Address EJB, it's better to define a local interface only because it's such a fine-grained bean. To get around remote interface restrictions, the business methods in the bean class exchange address data instead of Address references. For example, we can declare a method that allows the client to send address information to create a home address for the Customer.

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
    }
    // business method
    public void setAddress(String street,String city,
                           String state,String zip)
                           throws CreateException    {

        AddressLocal addr = this.getHomeAddress( );
```

```
        if(addr == null){
           // Customer doesn't have an address yet. Create a new one.
           InitialContext cntx = new InitialContext( );
           AddressHomeLocal addrHome =
                        (AddressHomeLocal)cntx.lookup("homeAddress");
           addr = addrHome.createAddress(street, city, state, zip);
           this.setHomeAddress(addr);
        }else{
           // Customer already has an address. Change its fields
           addr.setStreet(street);
           addr.setCity(city);
           addr.setState(state);
           addr.setZip(zip);
        }
    }

    ...
```

The `setAddress()` business method in the `CustomerBean` class is also declared in the remote interface of the Customer EJB, so that it can be called by remote clients.

```
public interface Customer extends javax.ejb.EJBObject {

    public void setAddress(String street,String city,
                           String state,String zip)
                throws CreateException;

    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;

    public boolean getHasGoodCredit( ) throws RemoteException;
    public void setHasGoodCredit(boolean creditRating)
    throws RemoteException;

}
```

When the `CustomerRemote.setAddress()` business method is invoked on the `CustomerBean`, the method's arguments are used to create a new Address EJB and set it as the `homeAddress` relationship field if one doesn't already exist. If the Customer EJB already has a `homeAddress` relationship, that Address EJB is modified to reflect the new address information.

When creating a new `Address EJB`, the home object is obtained from the JNDI ENC. and its `createAddress()` method is called. This results in the creation of a new Address EJB and the insertion of a corresponding `ADDRESS` record into the database. After the Address EJB is created, it's used in the `setHomeAddress()` method. The `CustomerBean` class must explicitly call

the `setHomeAddress()` method, otherwise the new address will not be assigned to the customer. In fact, simply creating an Address EJB, without assigning it to the customer using the `setHomeAddress()` method, will result in a *disconnected* Address EJB. More precisely, it will result in an `ADDRESS` record in the database that is not referenced by any `CUSTOMER` records. Disconnected entity beans are fairly normal and even desirable in many cases. In this case, however, we want the new Address EJB to be assigned to the `homeAddress` relationship field of the Customer EJB.

> The viability of disconnected entities depends, in part, on the referential integrity of the database. If the database requires that a foreign key contain a pointer to an existing record, then creating a disconnected entity would result in a database error.

When the `setHomeAddress()` method is invoked, the container links the `ADDRESS` record to the `CUSTOMER` record automatically. In this case, it places the `ADDRESS` primary key in the `CUSTOMER` record's `ADDRESS_ID` field and creates a reference from the `CUSTOMER` record to the `ADDRESS` record.

If the Customer EJB already has a `homeAddress`, then we want to change its values instead of creating a new one. Once the values of the existing Address EJB have been updated, we don't need to use `setHomeAddress()` since the Address EJB we modified already has a relationship with the entity bean.

The `AddressHome.createAddress()` method is declared as throwing a `CreateException`, as are all create methods. This requires that the `setAddress()` business method either wrap the `createAddress()` call in a try/catch block or propagate the exception to the client. In the above example, we choose to propagate the exception because it's more expedient. As an alternative you could catch the `CreateException` and throw a new application exception. Either approach is perfectly acceptable.

We will also want to provide clients with a business method for obtaining a Customer EJB's home address information. Since we are prohibited from sending an instance of the Address EJB directly to the client (because it's a local interface), we must package the address data in some other form and send that the client. There are two solutions to this problem: acquire the remote interface of the Address EJB and return that; or return the data as a dependent value object.

We can only obtain the remote interface for the Address EJB if one was defined. Entity beans can have a set of local interfaces or remote interfaces or both. In this situation the Address EJB is too fine-grained to justify creating a remote interface, but in many other circumstances a bean may indeed want to have a remote interface. If for example, the Customer EJB referenced a SalesPerson EJB, the `CustomerBean` would need to convert the local reference into a remote

reference. This would be done by accessing the local EJB object, getting its primary key (`EJBLocalObject.getPrimaryKey()`), obtaining the SalesPerson EJB's remote home from the JNDI ENC, and then using the primary key and remote home reference to find a remote interface reference.

```
public SalesRemote getSalesRep( ){
    SalesLocal local = getSalesPerson( );
    Integer primKey = local.getPrimaryKey();

    Object ref = jndiEnc.lookup("SalesHomeRemote");
    SalesHomeRemote home = (SalesHomeRemote)
    PortableRemoteObject.narrow(ref, SalesHomeRemote.class);

    SalesRemote remote = home.findByPrimaryKey( primKey );
    return remote;
}
```

The other option is to use a dependent value to pass the Address EJB's data between remote clients and the Customer EJB. This is the approach recommended for fine-grained beans like the Address EJB—in general we don't want to expose these beans directly to remote clients.

The following shows how the dependent values class, `AddressDO`, is used in conjunction with the local component interfaces of the Address EJB. The `DO` in `AddressDO` is a convention used in this book; it's a qualifier that stands for Dependent Object.

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id){
        setId(id);
        return null;
    }
    public void ejbPostCreate(Integer id){
    }
    // business method
    public AddressDO getAddress( ){
        AddressLocal addrLocal = getHomeAddress();
        String street = addrLocal.getStreet();
        String city = addrLocal.getCity();
        String state = addrLocal.getState();
        String zip = addrLocal.getZip();
        Address addrValue = new Address(street,city,state,zip);
        return addrValue;
    }
    public void setAddress(AddressDO addrValue)
    throws CreateException    {

        String street = addrValue.getStreet();
        String city = addrValue.getCity();
        String state = addrValue.getState();
```

```
        String zip = addrValue.getZip();

        AddressDO addr = getAddressDO();

        if(addr == null){
           // Customer doesn't have an address yet. Create a new one.
           InitialContext cntx = new InitialContext( );
           AddressHome addrHome = (AddressHome)cntx.lookup("homeAddress");
            addr = addrHome.createAddress(street, city, state, zip);
           this.setHomeAddress(addr);
        }else{
           // Customer already has an address. Change its fields
           addr.setStreet(street);
           addr.setCity(city);
           addr.setState(state);
           addr.setZip(zip);
        }

    }
    ...
```

Here is the definition for an `AddressDO` dependent value class, which is used by the enterprise bean to send address information to the client.

```
public class AddressDO implements java.io.Serializable {
    private String street;
    private String city;
    private String state;
    private String zip;

    public AddressDO(String street, String city,
                     String state, String zip ) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }
    public String getStreet(){
        return street;
    }
    public String getCity(){
        return city;
    }
    public String getState(){
        return state;
    }
    public String getZip(){
        return zip;
    }
}
```

The `AddressDO` dependent value follows the conventions laid out in this book. It's immutable, which means it cannot be altered once its created. As stated earlier, immutability helps to reinforce that fact that the dependent values class is a copy and is not a remote reference.

You can now use a client application to test the Customer EJBs relationship with the Address EJB. The following code shows the client code that creates a new Customer, gives it an address, then changes the address using the method defined above.

```
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Properties;

public class Client {
    public static void main(String [] args) throws Exception {
            // obtain CustomerHome
            Context jndiContext = getInitialContext();
            Object obj=jndiContext.lookup("CustomerEJB");
            CustomerHome home = (CustomerHome)
            javax.rmi.PortableRemoteObject.narrow(obj,
                                                  CustomerHome.class);

            // create a Customer
            Integer primaryKey = new Integer(1);
            Customer customer = home.create(primaryKey);

            // create an address
            AddressDO address = new Address("1010 Colorado",
                                            "Austin","Texas", "78701");
            // set address
            customer.setAddress(address);

            address = customer.getAddress();
            System.out.print(primaryKey+" = ");
            System.out.println(address.getStreet( ));
            System.out.println(address.getCity( )+","+
                               address.getState()+" "+
                               address.getZip());


            // create a new address
            address = new Address("1600 Pennsylvania Avenue NW",
                                  "DC","WA", "20500");

            // change customer's address
            customer.setAddress(address);
```

```
            address = customer.getAddress();
            System.out.print(primaryKey+" = ");
            System.out.println(address.getStreet( ));
            System.out.println(address.getCity( )+","+
                               address.getState()+" "+
                               address.getZip());

            // remove Customer
            customer.remove();
    }

    public static Context getInitialContext()
                        throws javax.naming.NamingException {
        Properties p = new Properties();
        // ... Specify the JNDI properties specific to the vendor.
        //return new javax.naming.InitialContext(p);
        return null;
    }
}
```

The following listing shows the deployment descriptor for Customer EJB and Address EJB. To avoid confusion we will not discuss this deployment descriptor in detail in this chapter because its covered in detail in Chapter 7. Don't be too concerned about the details until they are explained in the next chapter.

📖 Exercise 6.3, Relationships Fields

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <home>com.titan.customer.CustomerHomeRemote</home>
            <remote>com.titan.customer.CustomerRemote</remote>
            <ejb-class>com.titan.customer.CustomerBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-version>2.x</cmp-version>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>lastName</field-name></cmp-field>
            <cmp-field><field-name>firstName</field-name></cmp-field>
            <primkey-field>id</primkey-field>
            <security-identity><use-caller-identity/></security-identity>
        </entity>
        <entity>
            <ejb-name>AddressEJB</ejb-name>
            <local-home>com.titan.address.AddressHomeLocal</home>
```

```
        <local>com.titan.address.AddressLocal</remote>
        <ejb-class>com.titan.address.AddressBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>java.lang.Object</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-version>2.x</cmp-version>
        <cmp-field><field-name>street</field-name></cmp-field>
        <cmp-field><field-name>city</field-name></cmp-field>
        <cmp-field><field-name>state</field-name></cmp-field>
        <cmp-field><field-name>zip</field-name></cmp-field>
        <security-identity><use-caller-identity/></security-identity>
    </entity>
</enterprise-beans>
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-Address
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-a-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>address
                </cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Address-belongs-to-Customer
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>AddressEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
<assembly-descriptor>
    <security-role>
        <role-name>Employees</role-name>
    </security-role>
    <method-permission>
        <role-name>Employees</role-name>
        <method>
            <ejb-name>CustomerEJB</ejb-name>
            <method-name>*</method-name>
        </method>
```



```
            <method>
                <ejb-name>AddressEJB</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <container-transaction>
            <method>
                <ejb-name>AddressEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <method>
                <ejb-name>CustomerEJB</ejb-name>
                <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
        <container-transaction>
    </assembly-descriptor>
</ejb-jar>
```

7

EJB 2.0 CMP: Entity Relationships

In Chapter 6 you learned about basic EJB 2.0 container-managed persistence. This included coverage of container-managed persistence fields and an introduction to a basic container-managed relationship field. In this chapter we will continue to develop the Customer EJB and discuss in detail each of seven possible relationships that entity beans can have with each other.

In order for entity beans to model real world business concepts they must be capable of forming complex relationships with each other. This was difficult to accomplish in EJB 1.1 container-managed persistence because of the simplicity of the programming model. In EJB 1.1, entity beans could have persistent fields but not relationship fields.

Relationship fields in EJB 2.0 can model very complex relationships between entity beans. In Chapter 6 you created a one-to-one relationship between the Customer and Address EJBs. This relationship was unidirectional; the Customer had a reference to the Address, but the Address did not have a reference back to the Customer. This is a perfectly legitimate relationship between these entities, but other more complicated relationships are also possible. Each Address could also reference its Customer. This is an example of bi-directional navigation, where both participants in the relationship maintain references to each other. In addition to one-to-one relationships, entity beans can also have one-to-many, many-to-one and many-to-many relationships. For example, the Customer EJB may have many phone numbers, but each phone number belongs to only one Customer (a one-to-many relationship). A Customer may also have been on many Cruises in the past and each Cruise will have had many Customers (a many-to-many relationship).

The Seven Relationship Types

Seven types of relationships can exist between EJBs. This chapter examines those relationships and how the beans' code and deployment descriptor work together to define the relationships. First, let's look at the different types of relationships that are possible. There are four different types of cardinality: one-to-one, one-to-many, many-to-one, and many-to-many. On top of that, each relationship can be either unidirectional or bidirectional. That yields eight possibilities, but if you think about it, you'll realize that one-to-many bidirectional and many-to-one bidirectional relationships are actually the same thing, yielding 7 distinct relationship types.

To understand the relationships, it helps to think about some simple examples. We'll expand on these examples in the course of the chapter.

one-to-one, unidirectional
: The relationship between a customer and an address. You clearly want to be able to look up a customer's address, but you probably don't care about looking up an address's customer.

one-to-one, bidirectional
: The relationship between a customer and a credit card number. Given a customer, you obviously want to be able to look up his or her credit card number. And, given a credit card number, it is also conceivable that you would want to look up the customer who owns the credit card.

one-to-many, unidirectional
: The relationship between a customer and a phone nubmer. A customer can have many phone numbers (business, home, cell, etc.). You probably wouldn't want to look up a customer given his phone number.

one-to-many, bidirectional
: The relationship between a cruise and a reservation. Given a reservation, you want to be able to look up the cruise that the reservation is for. And given a cruise, you want to be able to look up all reservations for that cruise. Note that a many-to-one bidirectional relationship is just another perspective on the same concept.

many-to-one, unidirectional
: The relationship between a cruise and a ship. You obviously want to look up the ship that will be used for a particular cruise, and many cruises share the same ship, though at different times. It's less useful to be able to look up the cruises that are associated with the given ship, though if you want this relationship, you can implement a many-to-one bidirectional relationship.

many-to-many, unidirectional
: The relationship between a reservation and a cabin. It's possible to make a reservation for multiple cabins, and you clearly want to be able to look up the cabin assigned to a reservation. But you're not likely to want to look up

the reservation associated with a particular cabin. (If you think you need to do so, you'd implement it as a bidirectional relationship.)

many-to-many, bidirectional
: The relationship between a cruise and a customer. A customer can make reservations on many cruises, and each cruise has many customers. You clearly want to be able to look up both the cruises on which a customer has a booking, and the customers that will be going on any given cruise.

Abstract Persistence Schema

In Chapter 6 you learned how to form a basic relationship between the Customer and Address entity beans using the abstract programming model. In reality, the abstract programming model is only half the equation. In addition to declaring abstract accessor methods, a bean developer must further describe the cardinality and direction of the entity-to-entity relationships in the bean's deployment descriptor. This is handled in the relationships section of the XML deployment descriptor. As we discuss each type of relationship in the following sections, both the abstract programming model and the XML elements will be examined. It's the purpose of this section to introduce you to the basic elements used in the XML deployment descriptor to better prepare you for subsequent sections on specific relationship types.

In this book we always refer to the Java programming idioms used to describe relationships, specifically the abstract accessor methods, as the *abstract programming model*. When referring to the XML deployment descriptor elements we use the term *abstract persistence schema*. In the EJB 2.0 specification, the term *abstract persistence schema* takes on a more general meaning referring to both the Java idioms and the XML elements, but this book separate these concepts so that they can be discussed more easily.

The abstract persistence schema of an entity bean is defined in the `<relationships>` section of the XML deployment descriptor for that bean. The `<relationships` section falls between the `<enterprise-beans>` section and the `<assembly-descriptor>` section. Within the relationships element each entity-to-entity relationship is defined in separate `<ejb-relation>` elements.

```
<ejb-jar>
<enterprise-beans>
    …
</enterprise-beans>
<relationships>
    <ejb-relation>
      …
    </ejb-relation>
    <ejb-relation>
      …
```

```
    </ejb-relation>
</relationships>
<assembly-descriptor>
…
</assembly-descriptor>
```

Defining relationship fields requires that an `<ejb-relation>` element be added to the XML deployment descriptor for each entity-to-entity relationship. These `<ejb-relation>` elements complement the abstract programming model. For each pair of abstract accessor methods that defined a relationship field, there is an `<ejb-relation>` element in the deployment descriptor. EJB 2.0 requires that the entity beans that participate in a relationship be defined in the same XML deployment descriptor.

Here is a partial listing of the deployment descriptor for the Customer and Address EJBs with the emphasis on the elements that define the relationship.

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CustomerEJB</ejb-name>
        <local-home>com.titan.customer.CusomterLocalHome</local-home>
        <local>com.titan.customer.CustomerLocal</local>
        …
    </entity>
    <entity>
        <ejb-name>AddressEJB</ejb-name>
        <local-home>com.titan.address.AddressLocalHome</local-home>
        <local>com.titan.address.AddressLocal</local>
        …
    </entity>
    …
</enterprise-beans>

<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-Address
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-a-Address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>homeAddress
                </cmr-field-name>
            </cmr-field>
```



```
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Address-belongs-to-Customer
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>AddressEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
```

All relationships between the Customer EJB and other entity beans, such as CreditCard, Address, and Phone EJBs will require that we define a `<ejb-relation>` element to complement the abstract accessor methods.

Every relationship may, optionally, have a relationship name, which is declared in the `<ejb-relation-name>` element. This serves to identify the relationship for individuals reading the deployment descriptor or for deployment tools, but it's not required.

Every `<ejb-relation>` element has exactly two `<ejb-relationship-role>` elements, one for each participant in a relationship. In the previous example, the first `<ejb-relationship-role>` declares the Customer EJB's role in the relationship. We know this because the `<relationship-role-source>` element specifies the `<ejb-name>` as `CustomerEJB`. `CustomerEJB` is the `<ejb-name>` used in the Customer EJB's original declaration in the `<enterprise-beans>` section. The `<relationship-role-source>` element's `<ejb-name>` must always match an `<ejb-name>` element in the enterprise-beans section.

The `<ejb-relationship-role>` element also declares the cardinality, or multiplicity of the role. The `<multiplicity>` element can either be `One` or `Many`. In the case of the Customer EJB's `<ejb-relationship-role>` element, the `<multiplicity>` element has a value of `One`, which means that every Address EJB has a relationship with exactly *one* Customer EJB. The Address EJB's `<ejb-relationship-role>` specifies `One` also, which means that every Customer EJB has exactly *one* Address EJB. If the Customer had a relationship with many Address EJBs, the Address EBJs' `<multiplicity>` would be `Many`.

In Chapter 6, we defined the Customer EJB has having abstract accessor methods for getting and setting the Address EJB in the `homeAddress` field, but the Address EJB did not have abstract accessor methods for the Customer EJB. In this case the Customer EJB maintains a reference to the Address EJB, but the Address EJB doesn't maintain a reference back to the Customer EJB. This is a

unidirectional relationship, which means that only one of the entity beans in the relationship maintains a container-managed relationship field.

If the bean that is described by the `<ejb-relationship-role>` element maintains references to the other bean in the relationship, then that reference must be declared as a *container-managed relationship field* in the `<cmr-field>` element. The `<cmr-field>` element is declared under the `<ejb-relationship-role>` element.

```
<ejb-relationship-role>
    <ejb-relationship-role-name>
        Customer-has-a-Address
    </ejb-relationship-role-name>
    <multiplicity>One</multiplicity>
    <relationship-role-source>
        <ejb-name>CustomerEJB</ejb-name>
    </relationship-role-source>
    <cmr-field>
        <cmr-field-name>homeAddress</cmr-field-name>
    </cmr-field>
</ejb-relationship-role>
```

The field name declared in the `<cmr-field-name>` element must match a pair of abstract accessor methods in the bean class. In above example, the `<cmr-field-name>` is `homeAddress`, which corresponds to the pair of abstract accessor methods `getHomeAddress()` and `setHomeAddress()` defined in the `CustomerBean` class. EJB 2.0 requires that the `<cmr-field-name>` begin with a lower case letter. For every relationship field defined by a `<cmr-field>` element, there must be a pair of matching abstract accessor methods in the bean class. One method in this pair must be defined with the method name `set<cmr-field-name>()` where the first letter of the `<cmr-field-name>` value is changed to upper case. The other method is defined as `get<cmr-field-name>()` with the first letter of the `<cmr-field-name>` value in upper case. So, for example, the `<cmr-field-name>` value of `homeAddress` would have a corresponding abstract accessor methods `getHomeAddress()` and `setHomeAddress()`.

```
// bean class code
public abstract void setHomeAddress(AddressLocal address);
public abstract AddressLocal getHomeAddress( );

// XML deployment descriptor declaration
<cmr-field>
    <cmr-field-name>homeAddress</cmr-field-name>
</cmr-field>
```

The return type of the `get<cmr-field-name>()` method and the parameter type of the `set<cmr-field-name>()` must be exactly the same type. The type must be either the remote or local interface of the bean that is referenced or

one of two `java.util.Collection` types. In the case of the `homeAddress` relationship field, we are using the Address EJB's local interface, `AddressLocal`. Collection types are discussed in more detail in one-to-many, many-to-one and many-to-many relationships later in the chapter.

Having established a basic understanding of how elements are declared in the abstract persistence schema, you are now ready to discuss each of the seven types of relationships in more detail. In the process we will be introducing additional entity beans that have relationships with the Customer EJB including the CreditCard, Phone, Ship, and Reservation EJBs.

It's important to understand that although entity beans may have both local and remote interfaces, a container-managed relationship field may only use the entity bean's local interface when persisting a relationship. So for example, it would be illegal to define abstract accessor methods that have an argument type of `javax.ejb.EJBObject` (remote interface type). All container-managed relationships are based on `javax.ejb.EJBLocalObject` (local interface) types.

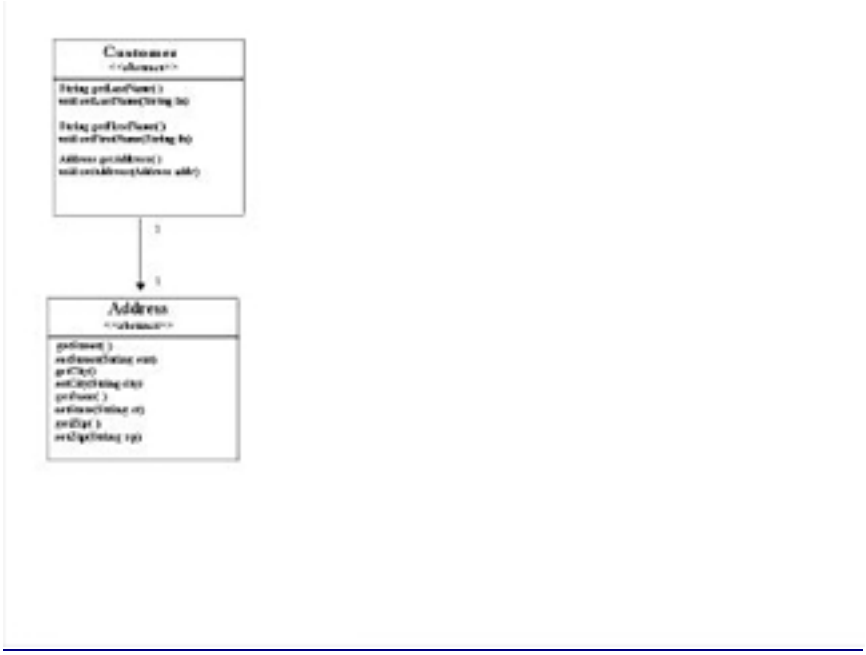
Database Modeling

This chapter discusses several ~~Through out this chapter~~ different database table schemas ~~are discussed~~. These schemas are intended ~~purely illustrative and are used only to~~ to demonstrate possible ~~manifestations of~~ relationships between entities in the database; they are not prescriptive. For example, the Address-Customer relationship is manifested by having `ADDRESS` table maintain foreign keys into the `CUSTOMER` table. This is not how most databases will be organized – instead they will use a link table or have the `ADDRESS` table maintain a foreign key to the `CUSTOMER`. ~~— h~~However, this schema shows ~~is useful in showing~~ how ~~different database schemas can be supported by~~ EJB 2.0's container-managed persistence can support different database organizations.

~~Its assumed t~~Through-out this chapter, we assume that the database tables are created before the EJB application. ~~— I~~n other words, that the EJB application is mapped to a legacy database. Some vendors ~~will~~ offer tools that generate tables automatically according to the relationships defined among entity beans. These tools may create schemas that are very different from the ones explored here. In other cases, vendors that support established database schemas may not have the flexibility to support the schemas illustrated in this chapter. As an EJB developer, you must be flexible enough to adapt to the facilities provided by your EJB vendor.

One-to-one Unidirectional Relationship

An example of a one-to-one unidirectional relationship is the relationship between the Customer EJB and the Address EJB defined in Chapter 6. In this case, a Customer has exactly one Address and every Address has exactly one Customer. Which bean references which determines the direction of navigation. While the Customer has a reference to the Address, the Address doesn't reference the Customer. This is a unidirectional relationship because you can only go from the Customer to the Address, and not the other way around. In other words, an Address EJB has no idea who owns it. Figure 7-1 shows this relationship.



[Figure 8-1 figure 7-1 and 6-1 are the same]

Figure 7-1: One-to-one Unidirectional Relationship

Relational Database Schema

One-to-one unidirectional relationships normally use a fairly typical schema in relational databases where one table contains a foreign key (pointer) to another table. The CUSTOMER table contains a foreign key to the ADDRESS table, but the ADDRESS table doesn't contain a foreign key to the CUSTOMER table. This allows records in the ADDRESS table to be shared by other tables, a scenario explored in section Many-to-many Unidirectional Relationships.FigureHolder

Figure 7-2: One-to-one Unidirectional Relationship in RDBMS

Abstract Programming Model

As you learned in Chapter 6, the abstract accessor methods are used to define relationship fields in the bean class. When an entity bean maintains a reference to another bean, it defines a pair of abstract accessor methods to model that reference. In unidirectional relationships, only one of the enterprise beans will define abstract accessor methods. It's called unidirectional because you can only navigate the relationship one-way. Inside the `CustomerBean` class you can call the `getHomeAddress()`/`setHomeAddress()` to access the Address EJBs, but inside the `AddressBean` class there are no methods to access the Customer EJB.

Although the relationship is unidirectional, the Address EJB can be shared between relationship fields of the same enterprise bean, but it may not be shared between Customer EJBs. If, for example, the Customer EJB defined two relationship fields, `billingAddress` and `homeAddress`, as one-to-one unidirectional relationships with the Address EJB, these two fields could conceivably reference the same Address EJB.

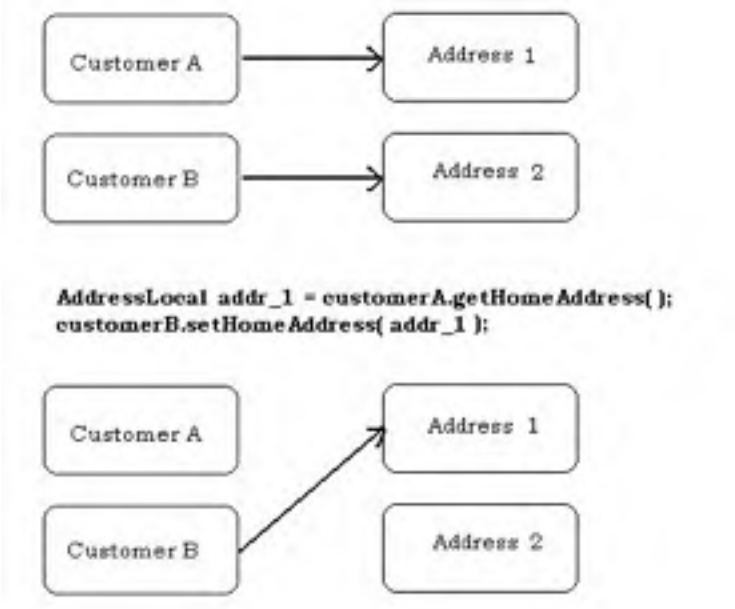
```
public class CustomerBean implements javax.ejb.EntityBean {
   …
   public void setAddress(String street,String city,
                          String state,String zip)
                          throws CreateException    {
      …

      address = addressHome.createAddress
                    (street, city, state, zip);

      this.setHomeAddress(address);
      this.setBillingAddress(address);

      AddressLocal billAddr, homeAddr;


      if(billAddr.isIdentical(homeAddr))
       // always true

      …
   }
   …
}
```

It's possible for two fields in a bean to reference the same relationship if the relationship type is the same. In this case, both the `homeAddress` and `billingAddress` have to be defined as one-to-one unidirectional relationships that utilize the Address EJB's local interface. At any time, if you want to make the `billingAddress` different from the `homeAddress`, you could be simply set it equal to a different Address EJB. Sharing a reference to

another bean between two relationship fields in the same entity is sometimes very convenient. In order to support this type of relationship a new billing address field might be added to the CUSTOMER table.

```
CREATE TABLE CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20),
 ADDRESS_ID INT,
 BILLING_ADDRESS_ID INT
}
```

However, it would not be possible to share the Address EJB between two different Customer EJBs. If, for example, the home Address of Customer A were assigned as the home Address of Customer B, the Address would be moved, not shared, so that Customer A wouldn't have a home Address any longer. As you can see in Figure 7-3, Address 2 is initially assigned to Customer B, but becomes disconnected when Address 1 is re-assigned to Customer B.



FigureHolder

Figure 7-3: Exchanging references in a One-to-One Unidirectional Relationship

This seemingly strange side affect is simply a natural result of how the relationship is defined. The Customer-to-Address EJB relationship was defined

as one-to-one, so the Address EJB is allowed to be referenced by only one Customer EJB.

Abstract Persistence Schema

The XML elements for the Customer-Address relationship were already defined in the Abstract Persistence Schema section, so we won't go over them again. The `<ejb-relation>` element used in that section declared a one-to-one unidirectional relationship. If, however, the Customer EJB did maintain two relationship fields with the Address EJB, `homeAddress`, and `billingAddress`, each of these relationships would have to be described in its own `<ejb-relation>` element.

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-HomeAddress
        </ejb-relation-name>
        <ejb-relationship-role>
            …
            <cmr-field>
                <cmr-field-name>homeAddress
                </cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            …
        </ejb-relationship-role>
    <ejb-relation>
    <ejb-relation>
        <ejb-relation-name>Customer-BillingAddress
        </ejb-relation-name>
        <ejb-relationship-role>
            …
            <cmr-field>
                <cmr-field-name>billing
                </cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            …
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
```

One-to-one Bi-directional Relationship

We can expand our Customer EJB to include a reference to a CreditCard EJB, which maintains credit card information. The Customer EJB will maintain a reference to its CreditCard EJB and the CreditCard EJB will maintain a reference

back to the Customer—this makes good sense, since a CreditCard should be aware of who owns it. When each CreditCard has a reference back to *one* Customer, and each Customer references *one* CreditCard, we have a *one-to-one bi-directional* relationship.

Relational Database Schema

The CreditCard EJB will have a corresponding `CREDIT_CARD` table and we neeed to add a CREDIT_CARD foreign key to the CUSTOMER table:

```
CREATE TABLE CREDIT_CARD
(
 ID INT PRIMARY KEY,
 EXP_DATE DATE,
 NUMBER CHAR(20),
 NAME CHAR(40),
 ORGANIZATION CHAR(20),
 CUSTOMER_ID INT
}

CREATE TABLE CUSTOMER
(
 ID INT PRIMARY KEY,
 LAST_NAME CHAR(20),
 FIRST_NAME CHAR(20),
 HOME_ADDRESS_ID INT,
 ADDRESS_ID INT,
 CREDIT_CARD_ID INT
)
```

One-to-one bi-directional relationships may model relational database schemas where the two tables each hold a foreign key for the other table. Specifically, two rows in different tables point to each other. Figure 7-4 illustrates how this schema would be implemented for rows in the `CUSTOMER` and `CREDIT_CARD` tables.

FigureHolder

Figure 7-4: One-to-one Bi-directional Relationship in RDBMS

Its also possible for a one-to-one bi-directional relationship to be established through a linking table where each foreign key column in the table must be unique, this is convenient when you do not want to impose relationship on the original tables. We will use linking tables in one-to-many and many-to-many relationships later in the chapter.

Abstract Programming Model

To model the relationship between the Customer and CreditCard, we'll need to declare a relationship field named `customer` in the `CreditCardBean` class.

```
public abstract class CreditCardBean extends javax.ejb.EntityBean {

    …

    // relationship fields
    public abstract CustomerLocal getCustomer( );
    public abstract void setCustomer(CustomerLocal local);

    // persistent fields
    public abstract Date getExpirationDate();
    public abstract void setExpirationDate(Date date);
    public abstract String getNumber();
    public abstract void setNumber(String number);
    public abstract String getNameOnCard();
    public abstract void setNameOnCard(String name);
    public abstract String getCreditOrganization();
    public abstract void setCreditOrganization(String org);

    // standard call back methods
    …

}
```

In this case, we use the Customer EJB's local interface (assume one has been created) because relationship fields require local interfaces types. All the relationships explored in the rest of this chapter assume local interfaces. Of course, the limitation of using local interfaces instead of remote interfaces is that you don't have location transparency. All the entity beans must be located in the same process or Java Virtual Machine. Although relationships fields using remote interfaces are not supported in EJB 2.0, it's likely that support for remote relationship fields will be added in a subsequent version of the specification.

We can also add a set of abstract accessor methods in the `CustomerBean` class for the `creditCard` relationship field.

```
public class CustomerBean implements javax.ejb.EntityBean {
    …
    public abstract void setCreditCard(CreditCardLocal card)
    public abstract CreditCardLocal getCreditCard( );
    …
}
```

Although a `setCustomer()` method is available in the `CreditCardBean`, we do not have to set the Customer reference on the CreditCard EJB explicitly. When a CreditCard EJB reference is passed into the `setCreditCard()` method on the `CustomerBean` class, the EJB Container will automatically

establish the customer relationship on the Address EJB to point back to the Customer EJB.

```
public class CustomerBean implements javax.ejb.EntityBean {
   …
   public void setCreditCard(Date exp, String numb,
                             String name, String org)
   throws CreateException    {
       …

       card = creditCardHome.create(exp,numb,name,org);

       // the Address EJB's customer field will be set automatically
       this.setCreditCard(card);

       Customer customer = card.getCustomer( );

       if(customer.isIdentical(ejbContext.getEJBLocalObject())
        // always true

       …
   }
   …
}
```

Attempting to share a CreditCard in a one-to-one bi-directional relationship has the same affect as in one-to-one unidirectional relationships. While the CreditCard EJB may be shared between relationship fields of the same entity identity, the CreditCard entity can't be shared between different Customer EJBs. Assigning the CreditCard of Customer A to Customer B disassociates that CreditCard from A, and moves it to B.

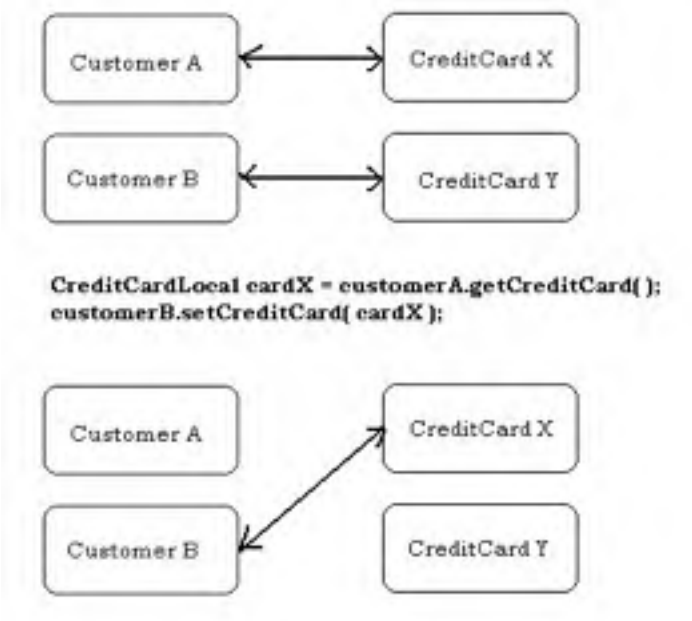


Figure holder

Figure 7-5: Exchanging references in a One-to-One Bi-idirectional Relationship

Abstract Persistence Schema

The `<ejb-relation>` element that defined the Customer-to-CreditCard relationship is very similar to the one used for the Customer-to-Address relationship, except for one important difference: both `ejb-relationship-role` elements have a `cmr-field`.

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-CreditCard
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-a-CreditCard
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>creditCard
                </cmr-field-name>
            </cmr-field>
```

```
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                CreditCard-belongs-to-Customer
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CreditCardEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>customer
                </cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
```

The fact that both participants in the relationship define `<cmr-field>` elements (relationship fields) tells up immediately that the relationship is bi-directional.

One-to-many Unidirectional Relationship

Entity beans can also maintain relationships with multiplicity. This means that one entity bean can aggregate or contain many other entity beans. For example, the Customer EJB may have many Phone EJBs, each of which represents a phone number. This is very different from the simple one-to-one relationship. One-to-many and many-to-many relationships require the developer to work with a collection of references when accessing the relationship field, instead of a single reference.

Relational Database

To illustrate a one-to-many unidirectional relationship, we will use a new entity bean, the Phone EJB, for which we must define a table, the `PHONE` table.

```
CREATE TABLE PHONE
(
 ID INT PRIMARY KEY,
 NUMBER CHAR(20),
 TYPE INT,
 CUSTOMER_ID INT
}
```

One-to-many unidirectional relationships between the `CUSTOMER` and `PHONE` tables could be manifested in a relational database in a variety of ways. For this example, we chose to have the `PHONE` table include a foreign key to the `CUSTOMER` table

The table of aggregated data can maintain a column of non-unique foreign keys to the aggregating table. In the case of the Customer and Phone EJBs, the `PHONE` table maintains a foreign key for the `CUSTOMER` table; one or more `PHONE` records may contain foreign keys the same `CUSTOMER` record. Here the pointer is reversed in the database, so that the `PHONE` records point to the `CUSTOMER` records. Although the database has the `PHONE` records pointing to the `CUSTOMER` records, the abstract programming model would have the Customer EJB pointing to the Phone EJBs. The two schemas are reversed, so how can it work? The container system will hide this reverse pointer so that it appears as if the Customer is aware of the Phone number and not the other way around. When you ask the container to return a `Collection` of Phone EJBs (invoking the `getPhoneNumgers()` method), it will query the `PHONE` table for all the records with a foreign key matching the Customer EJB's primary key.

FigureHolder

Figure 7-6: One-to-many Unidirectional Relationship in RDBMS using reverse pointers

This database schema, with reverse pointers, illustrates that the structure and the relationships of the database can be very different than the relationships as defined in the abstract programming model. In this case the tables are set up somewhat in reverse, but the EJB container system will manage the beans to meet the specification of the bean developer. This isn't always possible; in some cases, the database schema is incompatible with a desired relationship field. When dealing with legacy databases, databases that were established before the EJB application, a reverse pointer scenario like the one illustrated here is very common, so supporting this kind of relationship mapping is important.

A simpler implementation could use a link table that maintains two columns with foreign keys pointing to both the `CUSTOMER` and `PHONE` records. In this case we can constrain the link table so that the `PHONE` foreign key column requires unique entries, ensuring that every phone has only one customer, while the Customer foreign key column may have duplicates. The advantage of the link table is that it doesn't impose the relationship between the `CUSTOMER` and the `PHONE` onto either of the tables.

Abstract Programming Model

In the abstract programming model, we represent multiplicity by defining a relationship field that can point to many entity beans. This is accomplished by employing the same abstract accessor methods used for one-to-one relationships, except the field type is either a `java.util.Collection` or `java.util.Set`. The collection maintains a homogeneous group of local EJB object references, which means it contains many references to one kind of entity bean. The `Collection` type may contain duplicate references to the same entity bean, while the `Set` type may not.

For example, the Customer EJB may have many different phone numbers: a home phone, work phone, cell phone, fax, etc. Instead of having a single relationship field for each of these different Phone EJBs, the Customer EJB keeps all the Phone EJBs in a `Collection` relationship field, which can be accessed through abstract accessor methods:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    …
    // relationship fields
    public java.util.Collection getPhoneNumbers( );
    public void setPhoneNumbers(java.util.Collection phones);

    public AddressLocal getHomeAddress( );
    public void setHomeAddress(AddressLocal local);

    …
```

The Phone EJB, like other entity beans, has a bean class and local interface as shown in the next listing. Notice that the `PhoneBean` doesn't provide a relationship field for the Customer EJB. It's a unidirectional relationship; the Customer maintains a relationship with many Phone EJBs, but the Phone EJBs do not maintain a relationship field back to the Customer. Only the Customer EJB is aware of the relationship.

```
// The local interface for the Phone EJB
public interface PhoneLocal
extends javax.ejb.EJBLocalObject {
   public String getNumber();
   public void setNumber(String number);
   public byte getType();
   public void setType(byte type);
}

// The bean class for the Phone EJB
public class PhoneBean
implements javax.ejb.EntityBean {

   public Integer ejbCreate(String number, byte type){
         setNumber(number);
         setType(type);
   }
   public void ejbPostCreate(String number,byte type)
   {}

   // persistent fields
   public abstract String getNumber();
   public abstract void setNumber(String number);
   public abstract byte getType();
   public abstract void setType(byte type);
```



```
    // standard callback methods
    …
}
```

To illustrate how an entity bean uses a collection-based relationship field, we will define a method in the Customer EJB class that allows clients to add new phone numbers. The method, `addPhoneNumber()`, uses the phone number arguments to create a new Phone EJB and then add that Phone EJB to a `Collection` named `phoneNumbers`.

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    // business methods
    public void addPhoneNumber(String number, String type){

        InitialContext jndiEnc = new InitialContext( );
        PhoneHomeLocal phoneHome = jndiEnc.lookup("PhoneNumber");
        PhoneLocal phone = phoneHome.create(number,type);

        Collection phoneNumbers = this.getPhoneNumbers( );
        phoneNumbers.add(phone);

    }
    …
    // relationship fields
    public java.util.Collection getPhoneNumbers( );
    public void setPhoneNumbers(java.util.Collection phones);

    …
```

What is important with the above example is that the Phone EJB is first created, and then added to the `phoneNumbers` Collection. The `phoneNumbers Collection` is obtained from the `getPhoneNumbers()` accessor method and then the new Phone number EJB is added to the `Collection` just as you would add any object to a collection. The simple act of adding the Phone EJB to the `Collection` causes the EJB container to set the foreign key on the new `PHONE` record so that it points back to the Customer EJB's `CUSTOMER` record. If a link table had been used, a new link record would have been created. From this point forward, the new Phone EJB will be available from the `phoneNumbers` Collection.

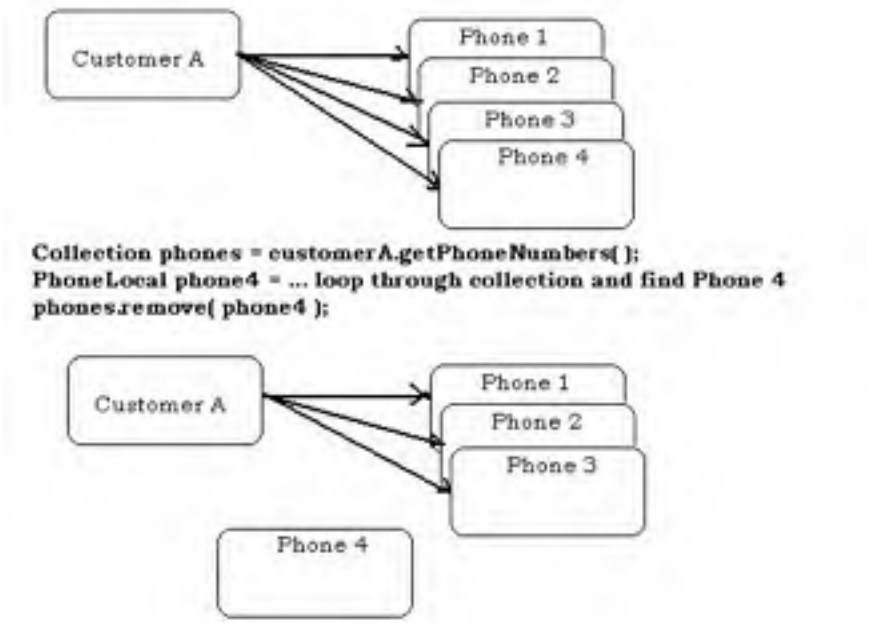
References in a `Collection`-based relationship field can also be updated or removed from the relationship using the relationship field accessor method. For example, the following code defines two methods in the `CustomerBean` class that allow clients to remove or update phone numbers in the bean's `phoneNumbers` relationship field.

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    // business methods
```



```
public void removePhoneNumber(String typeToRemove){

    Collection phoneNumbers = this.getPhoneNumbers( );
    Iterator iterator = phoneNumbers.iterator();
    while(iterator.hasNext()){
       PhoneLocal phone = (PhoneLocal)iterator.next();
       if(phone.getType().equals(typeToRemove)){
            iterator.remove(phone);
            break;
       }
    }
}
public void updatePhoneNumber(String number,String typeToUpdate){
    Collection phoneNumbers = this.getPhoneNumbers( );
    Iterator iterator = phoneNumbers.iterator();
    while(iterator.hasNext()){
       PhoneLocal phone = (PhoneLocal)iterator.next();
       if(phone.getType().equals(typeToUpdate)){
            phone.setNumber(number);
            break;
       }
    }
}
…
// relationship fields
public java.util.Collection getPhoneNumbers( );
public void setPhoneNumbers(java.util.Collection phones);
```

In the `removePhoneNumber()` business method, a Phone EJB with the matching type was found and then removed from the collection. This has the effect of actually disassociating the phone number from Customer EJB so that its not referenced by any Customer. The phone number is not deleted from the database, it's just not referenced by a Customer.



FigureHolder

Figure 7-9: Removing a bean reference from a relationships field collection

The `updatePhoneNumber()` method actually modifies an existing Phone EJB, changing its state in the database. The Phone EJB is still referenced by the `Collection`, but its data has changed.

Both `removePhoneNumber()` and `updatePhoneNumber()` illustrate that a collection-based relationship can be accessed and updated just like any other `Collection` object. In addition, a `java.util.Iterator` can be obtained from the `Collection` for looping operations. However, caution should be exercised while using an iterator over a collection-based relationship. You *must not* add or remove elements from the `Collection` while using its iterator. The only exception to this rule is that the `Iterator.remove()` method may be called to remove an entry. Although the `Collection.add()` and `Collection.remove()` methods *can* be used in other circumstances, calling these methods *while* an iterator is in use will result in a `java.util.IllegalStateException` exception.

If the `phoneNumbers` relationship field has never had any beans added to it, the `getPhoneNumbers()` method will return an empty `Collection`. Multiplicity relationship fields never return `null`. The `Collection` object used with the relationship field is implemented by the container system and is proprietary to the vendor and tightly coupled with the inner workings of the container. This allows the EJB container to implement performance

enhancements like lazy loading or optimistic concurrency seamlessly, without exposing those proprietary mechanisms to the bean developer. Because the `Collection` is implemented and tightly coupled to the vendor's EJB container, its illegal to use application defined `Collection` objects in relationship fields. For example, it is illegal to create a new `Collection` object and then attempt to add that `Collection` object to the Customer EJB using the `setPhoneNumbers()` method.

```
public void addPhoneNumber(String number, String type){

  …
  PhoneLocal phone = phoneHome.create(number,type);

  Collection phoneNumbers = java.util.Vector( );
  phoneNumers.add(phone);

  // this is illegal. An exception will be thrown
  this.setPhoneNumbers(phoneNumbers);

}
// relationship fields
public java.util.Collection getPhoneNumbers( );
public void setPhoneNumbers(java.util.Collection phones);
```

We have used the `getPhoneNumbers()` method extensively but have not yet used the `setPhoneNumbers()`. In most cases, this method will not be used, because it updates an entire collection of phone numbers. However, in some scenarios it can be very useful for exchanging *like* relationships between entity beans.

If two Customer EJBs want to exchange phone numbers, they can do so in a variety of ways. The most important thing to keep in mind is that a Phone EJB, as the subject of the one-to-many unidirectional relationship, may only reference one Customer EJB. So a Phone EJB cannot be shared between Customer EJBs. It can be copied, so that both Customers have Phone EJBs with similar data, but the Phone EJB itself cannot be shared.

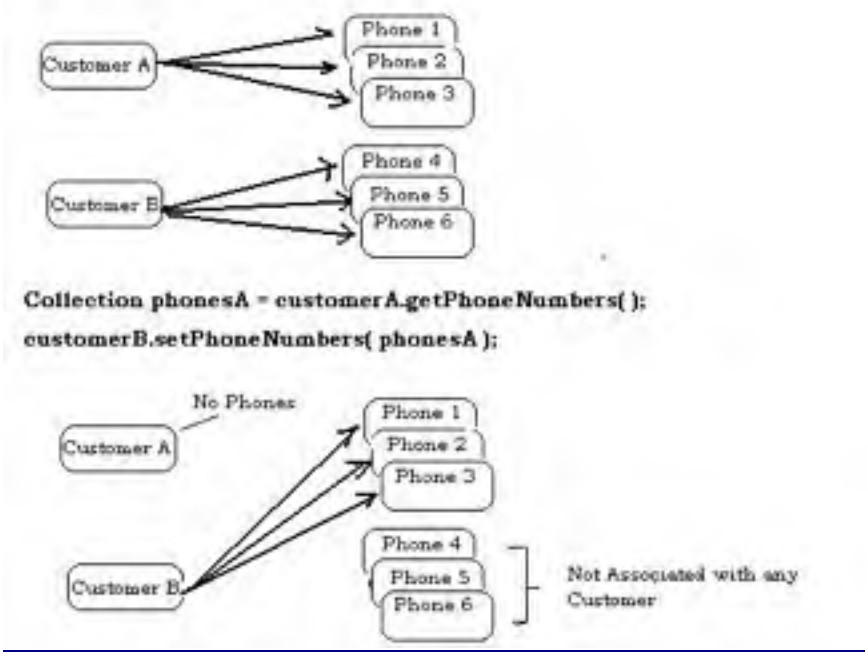
Imagine, for example, that Customer A wants to transfer all of its phone numbers to Customer B. It can accomplish this by using the `setPhoneNumbers()` method of Customer B as shown in the listing below. (We assume the Customer EJBs are interacting through their local interfaces.)

```
Customer customerA = … get Customer A
Customer customerB = … get Customer B

Collection phonesA = customerA.getPhoneNumbers( );
customerB.setPhoneNumbers( phonesA );

if( customerA.getPhoneNumbers( ).isEmpty())
    // this will be true
```



```
if( customerB.getPhoneNumbers().equals( phonesA ) )
    // this will be true
```

As the previous code and Figure 7-10 illustrate, passing one collection-based relationship to another actually disassociates those relationships from the first bean and associates them with the second. In addition, if the second already had a `Collection` of Phone EJBs in its `phoneNumbers` relationship field, those beans are bumped out of the relationship and disassociated from the bean.

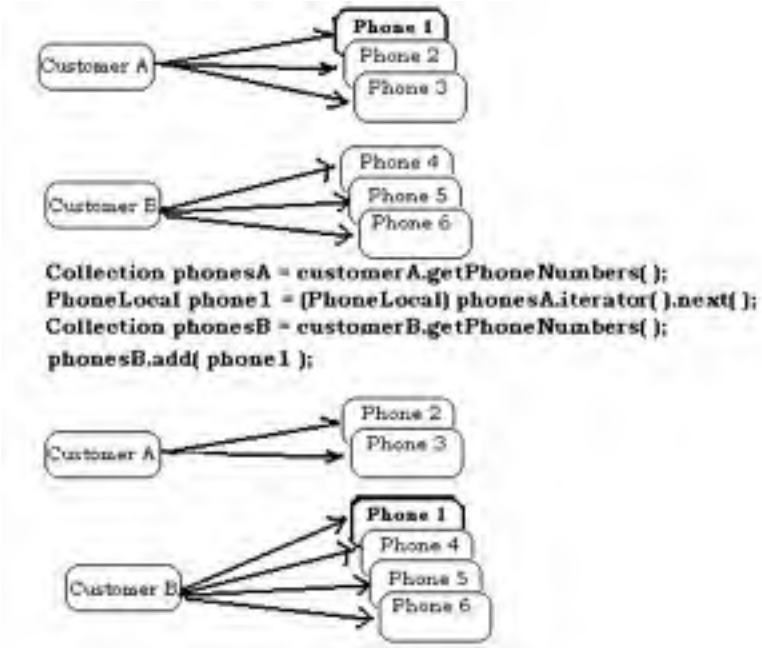


FigureHolder

Figure 7-10: Exchanging a relationship collection in a One-to-One unidirectional Relationship

The result of this exchange may be counterintuitive, but it is necessary to uphold the unidirectional aspect of the relationship, which says that the Phone EJB may only have one Customer EJB. This, at least, explains why Phone EJBs 1,2 and 3 don't reference both Customer A and B, but it doesn't explain why Phone EJBs 4, 5 and 6 disassociated from Customer B. Why isn't Customer B associated with all the Phone EJBs? The reason is purely a matter of semantics, since the relational database schema wouldn't technically prevent this from occurring. The act of replacing one `Collection` with another by calling `setPhoneNumbers(Collection collection)` implies that B's initial `Collection` object is no longer referenced, and is therefore not referenced by any Customer.

In addition to moving whole collection-based relationships between beans, it's also possible to move individual Phone EJBs between Customers, but again they cannot be shared. For example, if a Phone EJB aggregated by Customer A is added to the relationship collection of Customer B, that Phone EJB changes so that it's referenced by Customer B, and not A, as Figure 7-11 illustrates.



FigureHolder

Figure 7-11: Exchanging a bean in a One-to-One unidirectional Relationship

One again, it's the unidirectional aspect of the relationship that prevents Phone 1 from referencing both Customer A and B.

Abstract Persistence Schema

The abstract persistence schema for one-to-many unidirectional relationships has a couple of significant changes when compared to the `<ejb-relation>` elements seen so far, but these changes are easy to understand.

```
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-Phones
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-many-Phone-numbers
```



```
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>phoneNumbers
                </cmr-field-name>
                <cmr-field-type>java.util.Collection
                </cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Phone-belongs-to-Customer
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>PhoneEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
```

In the `<ejb-relation>` element, the multiplicity for the Customer EJB is declared as `One`, while the multiplicity for the Phone EJB's `<ejb-relationship-role>` is `Many`. This obviously establishes the relationship as one-to-many. The fact that the `<ejb-relationship-role>` for the Phone EJB doesn't specify a `<cmr-field>` element indicates that the one-to-many relationship is unidirectional; the Phone EJB doesn't contain a reciprocating reference to the Customer EJB.

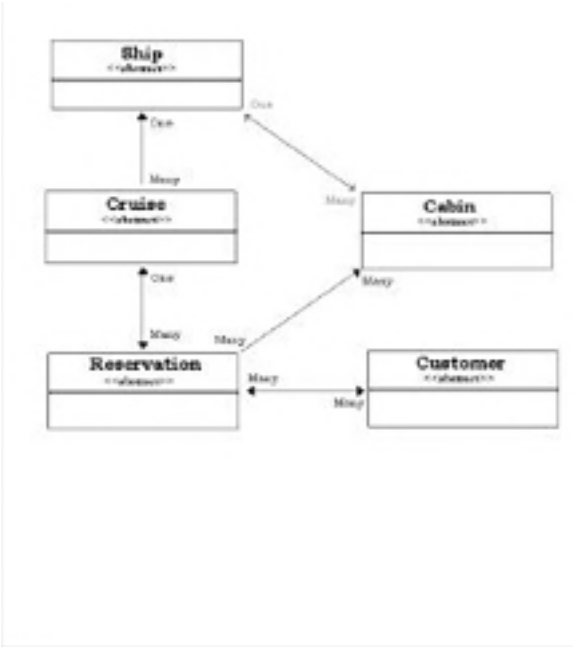
The most interesting change is the addition of the `<cmr-field-type>` element in the Customer EJB's `<cmr-field>` declaration. The `<cmr-field-type>` must be specified for the bean that has a collection-based relationship field (in this case the `phoneNumbers` field maintained by the Customer EJB). The `<cmr-field-type>` can have one of two values, `java.util.Collection` or `java.util.Set`, which are the allowed collection-based relationships types. In a future specification, the allowed types for collection-based relationships may be expanded to include `java.util.List` and `java.util.Map`, but these are not supported yet.

📖 Exercise 7.1, Customer Relationships

The Cruise, Ship, and Reservation EJBs

To make things more interesting, we are going to introduce some more entity beans so that we can model the remaining four relationships: Many-to-one unidirectional, One-to-many bi-directional, and many-to-many unidirectional and finally, many-to-many bi-directional.

In Titan's reservation system every customer (a.k.a. passenger) can be booked on one or more cruises. Each booking requires a reservation. A reservation may be for one, or more passengers (usually 2). Each cruise requires exactly one ship, but each ship may be used for many cruises through out the year. The following diagram illustrates these relationships.



FigureHolder

Figure 7-12: Cruise, Ship & Customer Class Diagram

In the next four sections the relationships investigated will each refer back to the above diagram and show how these relationships are manifested in EJB 2.0 container managed persistence.

Many-to-one Unidirectional Relationships

Many-to-one unidirectional relationships result when many entity beans reference a single entity bean, but the referenced entity bean is unaware of the

relationship. In the Titian Cruise business, for example, the concept of a cruise can be captured by a Cruise EJB. As shown in figure 7-12, each cruise has a many to one relationship with a ship. This relationship is unidirectional; the Cruise EJB will maintain a relationship with Ship EJB, but the Ship EJB is not going to keep track of which Cruises it used for.

Relational Database Schema

The relational database schema for the cruise-to-ship relationship is fairly simple; it requires that the `CRUISE` table maintain a foreign key column for the ship table, where each row in the `CRUISE` table points to a row in the `SHIP` table. The `CRUISE` and `SHIP` tables are defined below; Figure 7-13 shows the relationship between these tables in the database.

An enormous about of data would be required to adequately describe an ocean ship liner, but for the purposes of this book we will keep the definition of the `SHIP` table very simply.

```
CREATE TABLE SHIP
(
 ID INT PRIMARY KEY,
 NAME CHAR(30),
 TONNAGE DECIMAL (8,2)
}
```

The `CRUISE` table maintains data on each cruise's name, ship, and other information that is not germaine to this discussion. (Other tables such as `RESERVATIONS`, `SCHEDULES`, `CREW`, etc. would have relationships with the `CRUISE` table through linking tables.) For our purposes we'll keep it simple and focus on a definition that useful for the examples in this book.

```
CREATE TABLE CRUISE
(
 ID INT PRIMARY KEY,
 NAME CHAR(30),
 SHIP_ID INT
}
```

FigureHolder

Figure 7-13: Many to One Unidirectional Relationship in RDBMS

Abstract Programming Model

In the abstract programming model, the relationship field is of type `ShipLocal` and is maintained by the Cruise EJB. This is not particularly interesting, as the abstract accessor methods are similar to those defined in other examples.

```
public abstract class CruiseBean
implements javax.ejb.EntityBean {
```



```
    public Integer ejbCreate(String name,
                             ShipLocal ship) {
        setName(name);
    }
    public void ejbPostCreate(String name,
                              Ship shipLocal){
        setShip(ship);
    }
    public abstract void setName(String name);
    public abstract String getName( );
    public abstract void setShip(ShipLocal ship);
    public abstract ShipLocal getShip( );

   // EJB callback methods
    …
}
```

Notice that the Cruise EJB requires that a `ShipLocal` reference be passed as an argument when the Cruise is created; this is perfectly natural since a cruise cannot exist without a ship. According to the EJB 2.0 specification, relationship fields cannot be modified or set in the `ejbCreate()` method. They must be modifed in the `ejbPostCreate()`, a constraint that is followed in the `CruiseBean` class.

The reason relationships are set in `ejbPostCreate()` and not `ejbCreate()` is simple: In many cases it's simpler for the EJB container to link two beans together in a relationship after they both exist. Once the `ejbCreate()` method executes, the `CRUISE` record has been inserted to the database so that its relationship with the `SHIP` table can be established. This is especially important when, for example, a link table is used to model relationships. In that case, the link table may have referential integrity constraints that require both records to exist before they are linked[1].

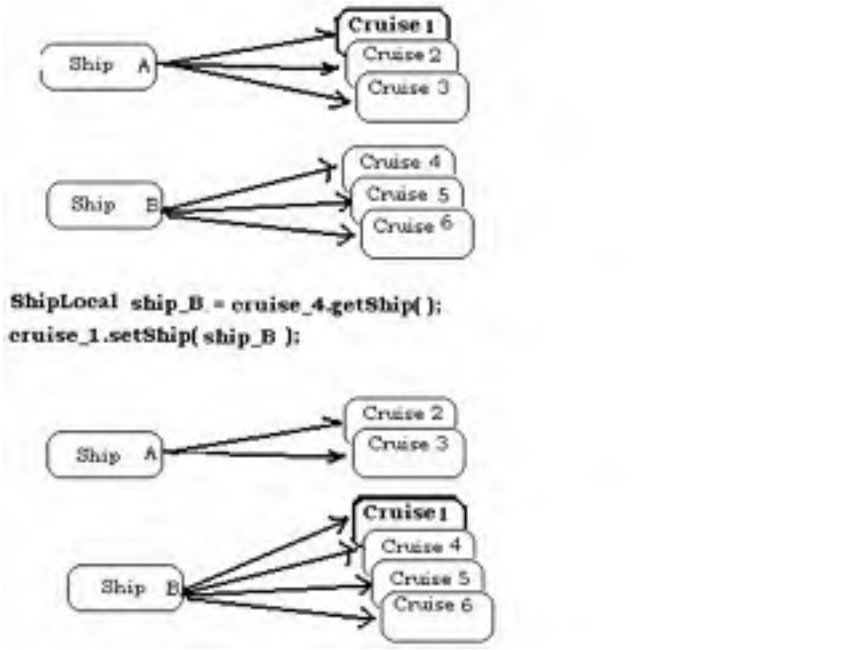
The Ship EJB is even simpler then the Cruise EJB. The relationship between the Cruise and Ship EJB in unidirectional, so the Ship EJB doesn't define any relationship fields, just persistent fields.

```
public abstract class ShipBean
implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer primaryKey,String name,
                    double tonnage) {
        setId(primaryKey);
        setName(name);
        setTonnage(tonnage);
```

[1] The database insert that occurs between the `ejbCreate()` and `ejbPostCreate()` would be done within the same transactional context as updates to the relationship field.


```
    }
    public void ejbPostCreate(Integer primaryKey,String name,
                        double tonnage) {
    }
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public abstract void setName(String name);
    public abstract String getName( );
    public abstract void setTonnage(double tonnage);
    public abstract double getTonnage( );

    // EJB callback methods
    …
}
```

This should all be fairly mundane for you now. The impact of exchanging Ship references between Cruise EJBs is equally obvious. Each Cruise may only reference a single Ship, but each Ship may have many Cruise EJBs. If you take the Ship A, which is referenced by some Cruise EJB, and pass set it to some other Cruise, then both Cruise EJBs will reference the same Ship.



FigureHolder

Figure 7-14: Sharing a bean reference in a many-to-one Unidirectional Relationship

Abstract Persistence Schema

The abstract persistence schema is very simple in a many-to-one unidirectional relationship. It uses everything you have learned up until now, and should not contain any surprises.

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CruiseEJB</ejb-name>
        <local-home>com.titan.cruise.CruiseLocalHome</local-home>
        <local>com.titan.cruise.CruiseLocal</local>
        …
    </entity>
    <entity>
        <ejb-name>ShipEJB</ejb-name>
        <local-home>com.titan.ship.ShipLocalHome</local-home>
        <local>com.titan.ship.ShipLocal</local>
        …
    </entity>
    …
</enterprise-beans>

<relationships>
    <ejb-relation>
        <ejb-relation-name>Cruise-Ship
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Cruise-has-a-Ship
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>CruiseEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>ship
                </cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Ship-has-many-Cruises
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>ShipEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
    <ejb-relation>
```



```
<relationships>
```

The <ejb-relationship-role> of the Cruise EJB defines its multiplicity as Many and declares ship as its relationship field. The <ejb-relationship-role> of the Ship EJB defines its multiplicity as Many and contains no <cmr-field> declaration, because it's a unidirectional relationship.

One-to-many Bi-directional Relationships

One-to-many and many-to-one bi-directional relationships are the same thing, so they are both covered in this section. A one-to-many bi-directional relationship occurs when one entity bean maintains a collection-based relationship field with another entity bean, and each entity bean referenced in the collection maintains a single reference back to its aggregating bean. For example, in the Titan Cruise system, each Cruise EJB maintains a reference to all the passenger reservations made for that Cruise, and each Reservation EJB maintains a single reference to its Cruise. The relationship is a many-to-one bi-directional relationship from the perspective of the Cruise EJB, and a one-to-many bi-directional relationship from the perspective of the Reservation EJB.

Relational Database Schema

The first table we need is the RESERVATION table, which is defined in the following listing. Notice that the RESERVATION table contains, among other things, a column that serves as a foreign key to the CRUISE table.

```
CREATE TABLE RESERVATION
(
 ID INT PRIMARY KEY,
 CRUISE_ID INT,
 AMOUNT_PAID DECIMAL (8,2),
 DATE_RESERVED DATE
}
```

While the RESERVATION table contains a foreign key to the CRUISE table, the CRUISE table doesn't maintain foreign keys back to the RESERVATION table. The EJB container system can realize the relationship between the Cruise and Reservations EJBs by querying the RESERVATION table. Explicit pointers from the CRUISE table to the RESERVATION table are not required. This illustrates once again the separation between the entity bean's view of its persistent relationships and the database's actual implementation of those relationships.

The relationship between the RESERVATION and CRUISE tables is illustrated in Figure 7-15.

FigureHolder

Figure 7-15: One-to-many/ many-to-one Bi-directional Relationship in RDBMS

As an alternative, we could have used a link table that would declare foreign keys to both the `CRUISE` and `RESERVATION` table. This link table would probably impose a unique constraint on the `RESERVATION` foreign key to ensure that each `RESERVATION` record had only one corresponding `CRUISE` record.

Abstract Programming Model

To model the relationship between cruises and reservations, we'll first define the Reservation EJB, which maintains a relationship field to the Cruise EJB.

```
public abstract class ReservationBean
implements javax.ejb.EntityBean {

    public Integer ejbCreate(CruiseLocal cruise){
    }
    public void ejbPostCreate(CruiseLocal cruise){
        setCruise(cruise);
    }

    public abstract void setCruise(CruiseLocal cruise);
    public abstract CruiseLocal getCruise( );

    public abstract void setAmountPaid(float amount);
    public abstract float getAmountPaid( );
    public abstract void setDate(Date date);
    public abstract Date getDate( );

    // EJB callback methods
    …
}
```

When a Reservation EJB is created, a reference to the Cruise for which it is created must be passed to the `create()` method. Notice that the `CruiseLocal` reference is set in the `ejbPostCreate()` and not the `ejbCreate()` method. As in *many-to-one unidirectional* relationships, the `ejbCreate()` method is not allowed to update relationship fields; that is the job of the `ejbPostCreate()` method.

The Cruise EJB needs to have a collection-based relationship field added so that it can reference all the Reservation EJBs that were created for it.

```
public abstract class CruiseBean
implements javax.ejb.EntityBean {
    …

    public abstract void setReservations(Collection res);
```



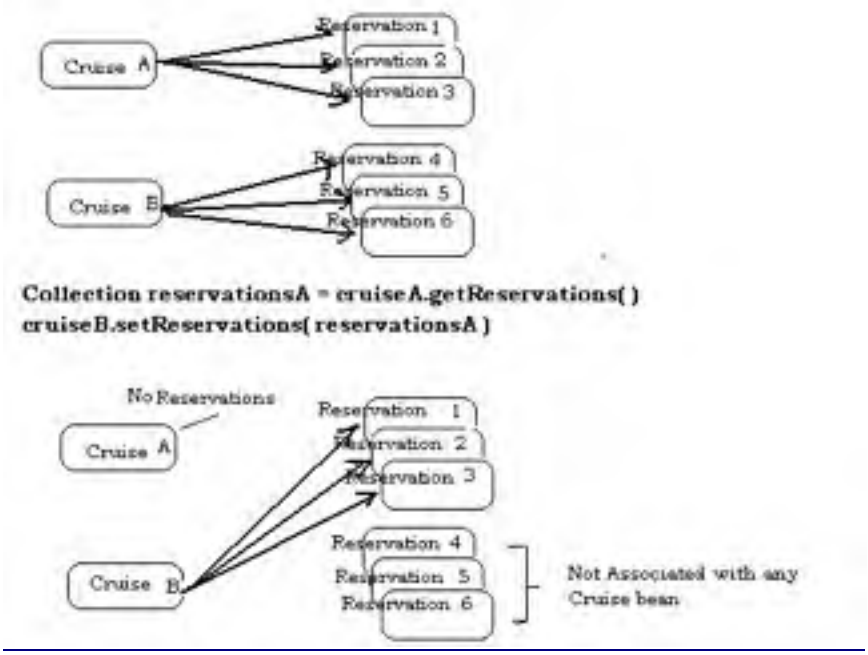
```
    public abstract Collection getReservations( );

    public abstract void setName(String name);
    public abstract String getName( );
    public abstract void setShip(ShipLocal ship);
    public abstract ShipLocal getShip( );

   // EJB callback methods
    …
}
```

The interdependency between the Cruise and Reservation EJBs produces some interesting results when creating a relationship between these beans. For example, the act of creating a Reservation EJB automatically adds that entity bean to the collection-based relationship of the Cruise EJB.

```
CruiseLocal cruise = … get CruiseLocal reference

ReservationLocal reservation = ReservationLocalHome.create( cruise );

Collection collection = cruise.getReservations( );

if(collection.contains(reservation))
    // always returns true
```

This is a side effect of the bi-directional relationship. Any Cruise referenced by a specific reservation has a reciprocal reference back to that reservation. If Reservation X references Cruise A, Cruise A must automatically have a reference to Reservation X. When you create a new Reservation EJB and set the Cruise reference on that bean, the Reservation is automatically added to the Cruise EJB's reservation field.

Sharing references between beans has some of the ugly side affects we learned about earlier. For example, passing a collection of reservations referenced by Cruise A to Cruise B actually moves those relationships to Cruise B, so Cruise A has no more Reservations.



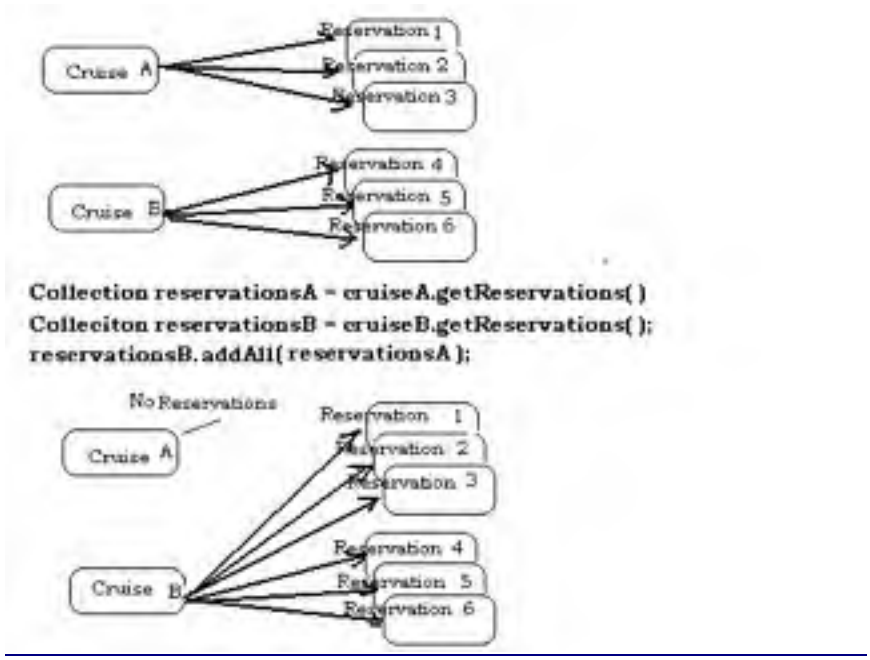
FigureHolder

Figure 7-16: Sharing an entire Collection in a one-to-many bi-directional relationship

As was the case with Customer and Phone (Figure 7-10), this effect is usually undesirable and should be avoided, as it displaces the set of Reservation EJBs formerly associated with Cruise B.

You can move an entire collection from one bean and combine it with the collection of another bean if you use the `Collection.addAll()` method as shown in the following figure[2]. The effect is that Cruise A does not reference any Reservation EJBs, while Cruise B references all of the Reservation EJBs—those it referenced before the exchange as well as Cruise B's Reservation EJBs.

[2] The `addAll()` method must be supported by collection-based relationship fields in EJB 2.0.



FigureHolder

Figure 7-17: Using Collection.addAll() in a one-to-many bi-directional relationship

The impact of moving individual Reservation EJBs from one Cruise to another is similar to what we have seen with other one-to-many relationships: the Reservation EJB is effectively moved from one Cruise to another. The result is the same as was shown in one-to-many unidirectional relationships when a Phone was moved from one Customer to another. See figure 7-11. It's interesting to note that the net affect of using `Collection.addAll()` in this scenario is the same as using `Collection.add()` on the target collection for every element in the source collection. In other words, you move every element from the source collection to the target collection.

Once again, container-managed relationship fields, collection-based or otherwise, must always use the `javax.ejb.EJBLocalObject` (local interface) of a bean and never the `javax.ejb.EJBObject` (remote interface). It would be illegal, for example, to try and add the remote interface of the Reservation EJB (if it has one) to the Cruise EJB's reservation Collection. Any attempt to add a remote interface type to a collection-based relationship field will result in a `java.lang.IllegalArgumentException.`

Abstract Persistence Schema

The abstract persistence schema for the Cruise-Reservation relationship doesn't introduce any new concepts. The Cruise and Reservation `<ejb-`

relationship-role> elements both have <cmr-field> elements. The Cruise specifies One as its multiplicity, while Reservation specifies Many.

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CruiseEJB</ejb-name>
        <local-home>com.titan.cruise.CruiseLocalHome</local-home>
        <local>com.titan.cruise.CruiseLocal</local>
        …
    </entity>
    <entity>
        <ejb-name>ReservationEJB</ejb-name>
        <local-home>
            com.titan.reservations.ReservationLocalHome
        </local-home>
        <local>com.titan.reservation.ReservationLocal</local>
        …
    </entity>
    …
</enterprise-beans>

<relationships>
    <ejb-relation>
        <ejb-relation-name>Cruise-Reservation
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Cruise-has-many-Reservations
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CruiseEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>reservations
                </cmr-field-name>
                <cmr-field-type>
                    java.util.Collection
                </cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Reservation-has-a-Cruise
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>ReservationEJB</ejb-name>
```



```
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>cruise
                </cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
```

Many-to-many Bi-directional Relationship

Many-to-many bi-directional relationships occurs when many beans maintain a collection-based relationship field with another bean, and each bean referenced in the Collection maintains a collection-based relationship fields back to the aggregating beans. For example, in Titan Cruises every Reservation EJB may reference many Customers (a family can make a single reservation) and each Customer may have many reservations (a person may make more than one reservation in a year). This is an example of a many-to-many bi-directional relationship; the customer keeps track of all of its reservations and each reservation may be for many customers.

Relational Database Programming

The `RESERVATION` and `CUSTOMER` tables have already been established. In order to establish a many-to-many bi-directional relationship, the `RESERVATION_CUSTOMER_LINK` table is created. This table maintains two columns: A foreign key column for the `RESERVATION` table and another foreign key column for the `CUSTOMER` table.

```
CREATE TABLE RESERVATION_CUSTOMER_LINK
(
 RESERVATION_ID INT,
 CUSTOMER_ID INT,
}
```

The relationship between the `CUSTOMER`, `RESERVATION` and `CUSTOMER_RESERVATION_LINK` table is illustrated in the following diagram.

FigureHolder

Figure 7-18: Many-to-many Bi-directional Relationship in RDBMS

Many-to-many bi-directional relationships will always require a link in a normalized relational database.

Abstract Programming Model

To model the many-to-many bi-directional relationship between the Customer and Reservation EJBs, we need to modify both bean classes to include collection-based relationship fields.

```
public abstract class ReservationBean
implements javax.ejb.EntityBean {

    public Integer ejbCreate(CruiseLocal cruise
                        Collection customers){
    }
    public void ejbPostCreate(CruiseLocal cruise
                         Collection customers){
        setCruise(cruise);
        Collection myCustomers = this.getCustomers();
        myCustomers.addAll(customers);
    }

    public abstract void setCustomers(Set customers);
    public abstract Set getCustomers( );
    …
}
```

The abstract accessor methods defined for the `customers` relationship field declare the Collection type as `java.util.Set`. The `Set` type should contain only unique Customer EJBs, and no duplicates. Duplicate customers would introduce some interesting but undesirable side effects in Titan's reservation system. To maintain a valid passenger count, and to avoid over-charging customers, Titan requires that a customer only be booked once in the same reservation. The `Set` collection type expresses this restriction. The effectiveness of the `Set` collection type depends largely on referential integrity constraints established in the underling database. Referential integrity of the database and its affect on relationships fields is explored at the end of this chapter.

In addition to adding the `getCustomers()/setCustomers()` abstract accessors, the `ejbCreate()/ejbPostCreate()` methods were modified to take a `Collection` of Customer EJBs. When a Reservation EJB is created, it must be provided with a list of Customer EJBs that it will add to its own Customer EJB collection. As is always the case, container-managed relationships field cannot be modified in the `ejbCreate()` method. It's the job of the `ejbPostCreate()` method to modify container-managed relationships fields when a bean is created.

The Customer EJB is also modified to maintain a collection-based relationship with all of its reservations. While the idea of a Customer having multiple reservations may seem odd, it's possible for someone to book more than one

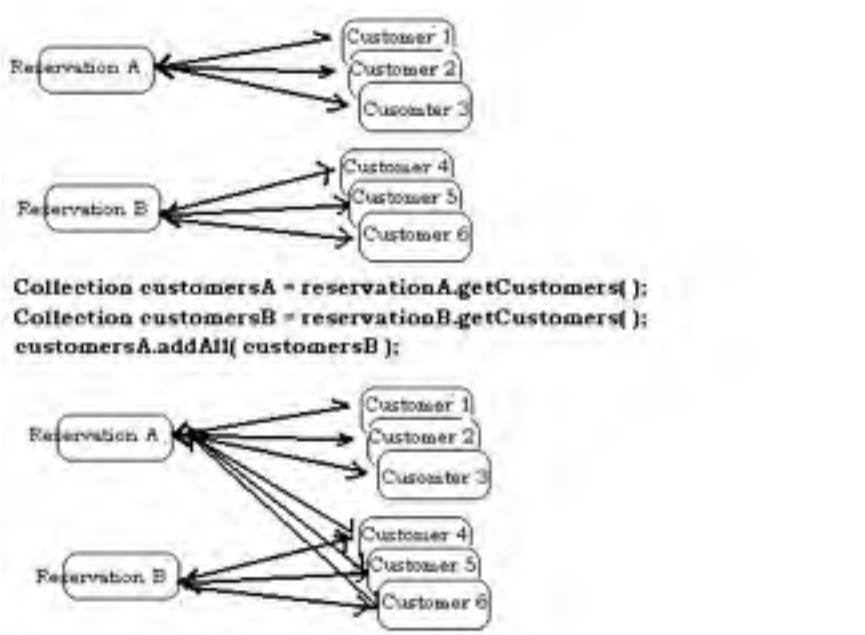
cruise in advance. In order to capture this possibility, the Customer EJB is enhanced to include a `reservations` relationship field:

```
public abstract class CustomerBean
implements javax.ejb.EntityBean {
    …
    // relationship fields
    public abstract
    void setReservations(Collection reservations);

    public abstract Collection getReservations();
    …
```

When a Reservation EJB is created, it is passed references to both its Cruise and a collection of Customers. Because the relationship is defined as bi-directional, the EJB container will automatically add the Reservation EJB to the reservations `relationship` field of the Customer EJB. The following code fragment illustrates this:

```
Collection customers = .. get local Customer EJBs
CruiseLocal cruise = .. get a local Cruise EJB
ReservationLocalHome = .. get local Reservation home

ReservationLocal myReservation =
               resHome.create( cruise,  customers);

Iterator iterator = customers.iterator();
while(iterator.hasNext()){
    CustomerLocal customer = CustomerLocal)iterator.next();
    Collection reservations customer.getReservations();
    if( reservations.contains( myReservation ))
        // this will always be true
}
```

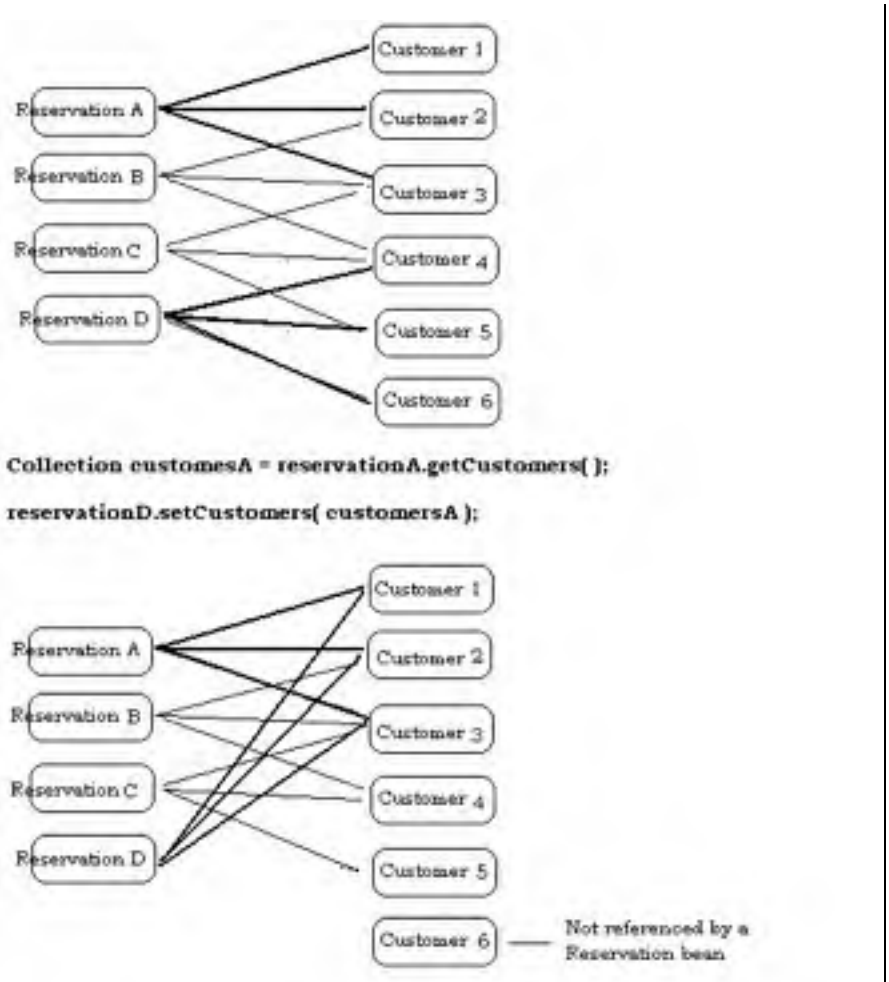
Exchanging bean reference between many-to-many bi-directional relationships results in true sharing, where each relationship maintains a reference to the transferred collection. This is illustrated in figure 7-19.



FigureHolder

Figure 7-19: Using Collection.addAll() in many-to-many bi-directional relationship

Of course, using the `setCustomers()` or `setReservations()` method will end up displacing the references of the target collection, *but* it doesn't impact the original relationship of the source collection. Figure 7-20 illustrates.



FigureHolder

Figure 7-20: Sharing an entire Collection in a many-to-many bi-directional relationship

After the `setCustomers()` method is invoked on Reservation D, Reservation D's customers change to Customer EJBs 1, 2, and 3. Customer EJBs 1, 2, and 3 were also referenced by Reservation A before the sharing operation and remain referenced after it's complete. In fact, only the relationships between Reservation D and Customers 4, 5 and 6 are impacted. The relationship between Customer EJBs 4,5 and 6 and other Reservation EJBs are not affected by the sharing operation. This is a unique property of many-to-many relationships (both bi-directional and unidirectional); operations on the relationship fields only affect those specific relationships, they do not impact either party's relationships with other beans of the same relationship type.

Abstract Persistence Schema

The abstract persistence schema of a many-to-many bi-directional relationship introduces nothing new and so it should have no surprises. Each `ejb-relationship-role` specifies `Many` as its `multiplicity` and declares a `cmr-field` of a specific `Collection` type.

```
<ejb-jar>
...
<enterprise-beans>
    <entity>
        <ejb-name>CustomerEJB</ejb-name>
        <local-home>com.titan.customer.CustomerLocalHome</local-home>
        <local>com.titan.customer.CustomerLocal</local>
        …
    </entity>
    <entity>
        <ejb-name>ReservationEJB</ejb-name>
        <local-home>
            com.titan.reservation.ReservationLocalHome
        </local-home>
        <local>com.titan.reservation.ReservationLocal</local>
        …
    </entity>
    …
</enterprise-beans>

<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-Reservation
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-many-Reservations
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>reservations
                </cmr-field-name>
                <cmr-field-type>
                    java.util.Collection
                </cmr-field-type>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Reservation-has-many-Customers
```



```
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>ReservationEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>customers
                </cmr-field-name>
                    <cmr-field-type>
                    java.util.Set
                </cmr-field-type>
        </cmr-field>
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
```

Many-to-many Unidirectional Relationship

Many-to-many unidirectional relationships occur when many beans maintain a collection based relationship with another bean, but the beans referenced in the `Collection` do not maintain a collection-based relationship back to the aggregating beans. In Titan's reservation system, every reservation is assigned a cabin on the ship. This allows customers to reserve a specific cabin (a deluxe suite or cabin with sentimental significance) on the Ship. In this case, each reservation may be for more then one cabin, since each reservation can be for more then one customer. An example is a family that makes a reservation for five for two adjacent cabins (one for the kids and the other for the parents).

While the reservation will want to keep track of the cabins it reserves, it's not necessary for the cabins to track all the reservations made by all the cruises, so the relationship is unidirectional. The Reservation EJBs reference a collection of Cabin beans, but the Cabin beans do not maintain references back to the Reservations.

Relational Database Schema

Our first order of business is to declare a `CABIN` table.

```
CREATE TABLE CABIN
(
 ID INT PRIMARY KEY,
 SHIP_ID INT,
 NAME CHAR(10),
 DECK_LEVEL INT,
 BED_COUNT INT
}
```

Notice that the `CABIN` table maintains a foreign key for the `SHIP` table. While this relationship is important, it's not explored because the relationship type

(one-to-many bi-directional) is already covered. The relationship is included in Figure 8-12, however, for completeness. Another interesting aspect of the CABIN table is its primary key.

In order to accommodate the many-to-many unidirectional relationship between the RESERVATION and CABIN table, we will need a RESERVATION_CABIN_LINK table.

```
CREATE TABLE RESERVATION_CABIN_LINK
(
 RESERVATION_ID INT,
 CABIN_ID INT,
}
```

The relationship between the CABIN records and the RESERVATION records through the RESERVATION_CABIN_LINK table is illustrated in Figure 7-21.

FigureHolder

Figure 7-21: Many-to-many Unidirectional Relationship in RDBMS

Abstract Programming Model

In order to model this relationship need to add a collection-based relationship field for Cabin beans to the Reservation EJB.

```
public abstract class ReservationBean
implements javax.ejb.EntityBean {

    …

    public abstract void setCabins(Set customers);
    public abstract Set getCabins( );
    …
}
```

In addition, we need to define a Cabin bean. Notice that the Cabin bean doesn't maintain a relationship back to the Reservation EJB. The lack of a container-managed relationship field for the Reservation EJB tells us the relationship is unidirectional.

```
public abstract class CabinBean
implements javax.ejb.EntityBean {

    public Integer ejbCreate(ShipLocal ship,
                        String name){
        this.setName(name);
    }
    public void ejbPostCreate(ShipLocal ship,
                        String name){
        this.setShip(ship);
```

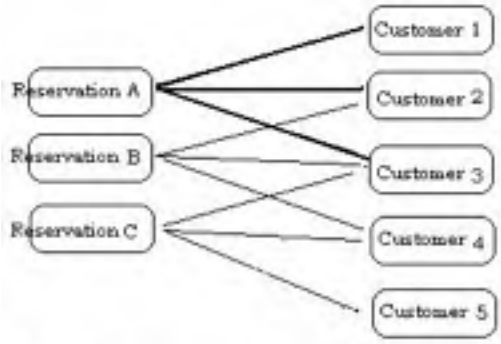


```
    }
    public abstract void setShip(ShipLocal ship);
    public abstract ShipLocal getShip( );
    public abstract void setName(String name);
    public abstract String getName( );
    public abstract void setBedCount(int count);
    public abstract int getBedCount( );
    public abstract void setDeckLevel(int level);
    public abstract int getDeckLevel( );

    // EJB callback methods
}
```

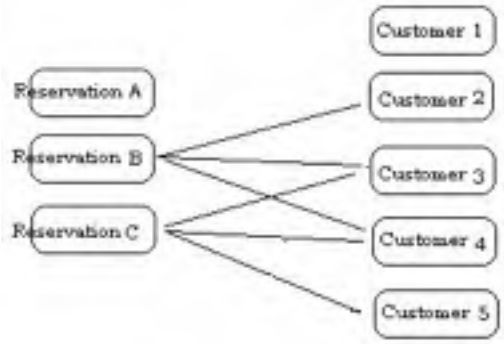
Although the Cabin bean doesn't define a relationship field for the Reservation EJB, it does define a one-to-many bi-directional relationship for the Ship EJB.

The effect of exchanging relationship fields in a many-to-many unidirectional relationship is basically the same as with many-to-many bi-directional relationships. Use of the `Collection.addAll()` operation and sharing entire collections has the same net effect as we noted in the section on many-to-many bi-directional relationships. The only difference is that the arrows only point one way.

If a reservation removes a Cabin bean from its collection-based relationship field, the operation doesn't affect other Reservation EJBs that reference that same Cabin bean. This is illustrated in Figure 7-22.




```
Collection customersA = reservationA.getCustomers( );

Iterator iterator = customersA.iterator( );

while(iterator.hasNext() )
    iterator.remove( );
```



FigureHolder

Figure 7-22: Removing beans in many-to-many unidirectional relationship

If you performed this exact same operation on the many-to-many bi-directional relationship, the result would be the same except the arrows would point both ways.

Abstract Persistence Schema

The abstract persistence schema for the Reservation-Cabin relationship holds no surprises whatsoever. The multiplicity of both `ejb-relationship-role` elements is `Many`, but only the Reservation EJB's `ejb-relationship-role` defines a `cmr-field`.

```
<ejb-jar>
```



```
...
<enterprise-beans>
    <entity>
        <ejb-name>CabinEJB</ejb-name>
        <local-home>com.titan.cabin.CabinLocalHome</local-home>
        <local>com.titan.cabin.CabinLocal</local>
        …
    </entity>
    <entity>
        <ejb-name>ReservationEJB</ejb-name>
        <local-home>
            com.titan.reservation.ReservationLocalHome
        </local-home>
        <local>com.titan.reservation.ReservationLocal</local>
        …
    </entity>
    …
</enterprise-beans>

<relationships>
    <ejb-relation>
        <ejb-relation-name>Cabin-Reservation
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Cabin-has-many-Reservations
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>CabinEJB</ejb-name>
            </relationship-role-source>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Reservation-has-many-Customers
            </ejb-relationship-role-name>
            <multiplicity>Many</multiplicity>
            <relationship-role-source>
                <ejb-name>ReservationEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>cabins
                </cmr-field-name>
                    <cmr-field-type>
                    java.util.Set
                </cmr-field-type>
        </cmr-field>
        </ejb-relationship-role>
    <ejb-relation>
<relationships>
```

📖 Exercise 7.2, Reservation relationships

Collocation and the Deployment Descriptor

Only entity beans that are deployed together with the same deployment descriptor can have relationships with each other. When deployed together, the entity beans are seen as a single deployment unit or application, in which all the entities are using the same database and are co-located in the same Java virtual machine. This restriction makes it possible for the EJB container system to use lazy loading, optimistic concurrency, and other performance optimizations. While it would be technically possible to support relationships across deployments, or even container systems, the difficulty of doing so combined with the expected degradation in performance was reason enough to limit the relationship fields to those entity beans that are deployed together. In the future, entity relationships may be expanded to include remote reference to entities deployed in other containers or other JARs in the same container, but remote references are not allowed as relationship types in Enterprise JavaBeans 2.0.

Cascade Delete and Remove

As you learned in Chapter 5, invoking the `remove()` operation on the EJB home or EJB object of an entity bean deletes that entity bean's data from the database. This, of course, has an impact on the relationships that the entity has with other entity beans.

When an entity bean is deleted, the EJB container first removes it from any relationships it maintains with other entity beans. Consider, for example, the relationship between the entity beans we have created in this chapter as shown in Figure 7-23.

[Figure 7-23 (note this is the same figure as figure 8-1)]

Figure 7-23: Titan Cruises Class Diagram

If an EJB application invokes `remove()` on a CreditCard EJB, then the Customer EJB that referenced it would now have a value of `null` for its `creditCard` relationships field, as the following code fragment illustrates.

```
CustomerLocal customer = … get Customer EJB
CreditCardLocal creditCard = customer.getCreditCard( );
creditCard.remove();
if( custsomer.getCreditCard() == null)
    //This will always be true;
```


The moment the `remove()` operation is invoked on the CreditCard EJB's local reference, the bean is disassociated from the Customer bean and is deleted. The impact of removing a bean is even more interesting when it participates in several relationships. For example, invoking `remove()` on a Customer EJB will impact the relationship fields of Reservation, Address, Phone, and CreditCard EJBs. With single EJB object relationship fields, such as the CreditCard EJB's reference to the Customer EJB, the field is set to `null` for the entity bean that was removed. With collection-based relationship fields, the entity that is removed is no longer a part of the collection. This was shown in Figure 7-9 of the *One-to-many Unidirectional Relationship* section, where a Phone EJB was removed.

In some cases, you want the removal of an entity bean to cause a cascade of deletions. For example, if a Customer EJB is removed, we would want the Address EJBs referenced in its `billingAddress` and `homeAddress` relationships field to be deleted. This would avoid the problem of disconnected Address EJBs in the database. The `<cascade-delete>` element requests cascade deletion; it can be used with one-to-one or one-to many relationships. Here's how to modify the relationship declaration for the Customer and Address EJBs to obtain cascade delete:

```
<relationships>
        <ejb-relation>
            <ejb-relationship-role>
                <multiplicity>One</multiplicity>
                <role-source>
                    <ejb-name>CustomerEJB</ejb-name>
                </role-source>
                <cmr-field>
                    <cmr-field-name>homeAddress</cmr-field-name>
                </cmr-field>
            </ejb-relationship-role>
            <ejb-relationship-role>
                <multiplicity>One</multiplicity>
                <cascade-delete/>
                <role-source>
                    <dependent-name>Address</dependent-name>
                </role-source>
            </ejb-relationship-role>
         </ejb-relation>
    </relationships>
```

Without specifying a cascading delete, the `ADDRESS` record associated with the Address EJB will not be removed when the `CUSTOMER` record is deleted. This can result in a disconnected dependent object class, which means that the data is not linked to anything. In some cases we want to specify a cascading delete to ensure that there are no detached entities following a removal. In other cases, however, we do not want to use a cascading delete. If, for example, the `ADDRESS` record associated with an entity bean is shared by other `CUSTOMER`

records, then we probably do not want it deleted when the CUSTOMER record is deleted. It's easy to imagine two different customers residing at the same residence—sharing address records can be useful.

Cascade delete can only be specified on an entity bean that has a single reference to the entity that is being deleted. For example, the `<ejb-relationship-role>` for the Phone EJB in the Customer-Phone relationship can have a cascade deleted specified if the Customer is deleted, because each Phone EJB is referenced by only one Customer. However, the Customer EJB cannot have a cascade delete specified in the Customer-Phone relationships, because a Customer maybe referenced by many Phone EJBs. The entity bean that causes the cascade delete must have a multiplicity of one in the relationships.

Cascade delete only affects the relationship for which it is specified. So for example, if cascade delete is specified for the Customer-Phone relationships but not the Customer-HomeAddress relationships, then detecting a Customer will cause all the Phone EJBs to be deleted but not the Address EJBs. The Address EJBs must specify their own cascade-delete element if they want to be deleted.

Cascade deletes can propagate through relationships in a chain reaction. For example, if the Ship-Cruise relationships specifies cascade-delete on the Cruise relationships field and the Cruise-Reservation relationships specifies cascade-delete on the Reservation relationship field, then when a Ship is removed all of its Cruises and Reservations for those cruises will be removed.

Cascade delete can be a very powerful tool, but it's also dangerous. It should be handled with care. The effectiveness of a cascade delete depends in large part on the referential integrity of the database. For example, the database may be set up so that a foreign key must point to an existing record, which could result in a transaction rollback if deleting an entity's data would violate that restriction.

📖 Exercise 7.3, Cascade Deletes

8

EJB 2.0 CMP: EJB-QL

Find methods have been a part of EJB since EJB 1.0. These methods are defined on the entity bean's local and remote home interfaces and are used for locating one or more entity beans. All entity beans must have a `findByPrimaryKey()` find method, which takes the primary key of the entity bean as an argument and returns a reference to an entity bean. For example, the Cruise EJB defines the standard primary key find method in its home interface:

```
public CruiseLocalHome extends javax.ejb.EJBLocalHome
{
    public Integer create(String name,ShipLocal ship);

    public CruiseLocal findByPrimaryKey(Integer key);

}
```

In addition to the mandatory `findByPrimaryKey()` methods, entity bean developers may also define as many custom find methods as they like. For example, the Cruise EJB might define a method (e.g., `findByName()`) for locating a Cruise with a specific name.

```
public CruiseLocalHome extends javax.ejb.EJBLocalHome
{
    public Integer create(String name,ShipLocal ship)
    throws CreateException;

    public CruiseLocal findByPrimaryKey(Integer key)
    throws FindException;

    public CruiseLocal findByName(String cruiseName)
    throws FindException;
```

```
}
```

The option of defining custom find methods is nothing new, but until EJB 2.0 there was no standard way of defining how the find methods should work. The behavior of the `findByPrimaryKey()` method is obvious: Find the entity bean with the same primary key. However, the behavior of the custom find methods is not obvious, so addition information is needed to tell the container how these custom find methods should behave. EJB 1.1 didn't provide any standard mechanism for declaring how custom find methods should behave, so vendors came up with their own query languages and methods. This resulted in non-portability and basically guesswork on the part of the deployer in determining how to execute queries of find methods. EJB 2.0 introduces EJB QL, which provides a standard query language for declaring the behavior of custom find methods, and adds new select methods. Select methods are similar to find methods, but they are more flexible and are visible to the bean class only—like private find methods. Find and select methods are collectively referred to as *query* methods in EJB 2.0.

EJB QL is a declarative query language that is similar to the Structured Query Language (SQL) used in relational databases, but it is tailored to work with the abstract persistence schema of entity beans in EJB 2.0.

EJB QL queries are defined in terms of the abstract persistence schema of entity beans and not the underlying data store, so they are portable across databases and data schemas. When an entity bean's abstract bean class is deployed by the container, the EJB QL statements are typically examined and translated into data access code optimized for that container's data store. At run time, query methods defined in EJB QL typically execute in the native language of the underlying data store. For example, a container that uses a relational database for persistence might translate EJB QL statements into standard SQL 92, while an object-database container might translate the same EJB QL statements into an object query language.

EJB QL makes it possible for bean developers to describe the behavior of query methods in an abstract fashion, making queries portable across databases and EJB vendors. The EJB QL language is easy for developers to learn, yet precise enough to be interpreted into native database code. It is a fairly rich and flexible query language that empowers developers at development time, while executing in fast native code at run time. However, EJB QL is not a silver bullet and its not without its problems, as we'll see later in this chapter.

Declaring EJB QL

EJB QL statements are declared in `<query>` elements of entity bean's deployment descriptor. In the following listing, you see that the

`findByName()` method defined in the Customer bean local home interface has its own query element and EJB QL statement.

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CruiseEJB</ejb-name>
            …
            <reentrant>False</reentrant>
            <abstract-schema-name>Cruise</abstract-schema-name>
            <cmp-version>2.x</cmp-version>
            <cmp-field>
                  <field-name>name</field-name>
            </cmp-field>
            <primkey-field>id</primkey-field>
            <query>
                <query-method>
                    <method-name>findByName</method-name>
                    <method-params>java.lang.String</method-params>
                </query-method>
                <ejb-ql>
                   SELECT OBJECT(c) FROM Cruise c WHERE c.name = ?1
                </ejb-ql>
            </query>
        </entity>
    </enterprise-beans>
```

The `<query>` element contains two primary elements. The `<query-method>` element identifies the find method of the remote and/or local home interface, and the `<ejb-ql>` element declares the EJB QL statement. The `<query>` element binds the EJB QL statement to the proper find method. Don't worry too much about the EJB QL statement just yet; we'll cover that in detail starting in the next section.

Every entity bean that will be referenced in an EJB QL statement must have a special designator called the *abstract schema name*, which is declared by the `<abstract-schema-name>` element. The `<abstract-schema-name>` elements must have unique names; no two entity beans may have the same abstract schema name. In the entity element that describes the Cruise EJB, the abstract schema name is declared as `Cruise`. The `<ejb-ql>` element contains an EJB QL statement that uses this identifier in its `FROM` clause.

In Chapter 7 you learned that the abstract persistence schema of an entity bean is defined by its `<cmp-fields>` and `<cmr-field>` elements. The abstract schema name is also an important part of the abstract persistence schema. EJB QL statements are always expressed in terms of the abstract persistence schema of entity beans. It uses the abstract schema names to identify entity bean types, and the container-managed persistence (CMP) fields to identify specific entity

bean data and container-managed relationship (CMR) fields to create paths for navigating from one entity bean to another.

The Query Methods

Find Methods

Find methods are invoked by EJB clients (applications or beans) in order to locate and obtain remote or local EJB object reference of a specific entity bean. For example, you might call the `findByPrimaryKey()` method on the Customer EJB's home interface to obtain a reference to a specific Customer bean.

Find methods are always declared in the local and remote home interfaces of an entity bean. As you have already learned, every home interface must define a `findByPrimaryKey()` method; this is a type of single-entity find method. Specifying a single remote or local return type for a find method indicates that the method only locates one bean. `findByPrimaryKey()` obviously returns one remote reference because there is a one-to-one relationship between a primary key's value and an entity. Other single-entity find methods can also be declared. For example, the Customer EJB could declare several single-entity find methods, each of which supports a different query.

```
public interface CustomerHome extends javax.ejb.EJBHome {
    public Customer findByPrimaryKey( Integer primaryKey )
    throws javax.ejb.FindException;

    public Customer findByName( String lastName, String firstName )
    throws javax.ejb.FindException;

    public Customer findBySSN( String socialSecurityNumber )
    throws javax.ejb.FindException;
}
```

Bean developers can also define multi-entity find methods, which return a collection of EJB objects. The following listing shows a couple of multi-find methods:

```
public interface CustomerLocalHome extends javax.ejb.EJBLocalHome {
    public CustomerLocal findByPrimaryKey( Integer primaryKey )
    throws javax.ejb.FindException;

    public Collection findByCity(String city,String state)
    throws javax.ejb.FindException;

    public Set findByGoodCredit()
    throws javax.ejb.FindException;
}
```

To return several references from a find method, you must use the `java.util.Collection` or `java.util.Set` collection types[1]. A find method that uses a `java.util.Set` return type will not have duplicate values, while a `java.util.Collection` return type *may* have duplicates. Multi-entity finds return an empty `Collection` or `Set` if no matching beans can be found.

Enterprise JavaBeans specifies that all query methods (find or select) must be declared as throwing the `javax.ejb.FindException`. Find methods that return a single remote reference throw a `FindException` if an application error occurs and a `javax.ejb.ObjectNotFoundException` if a matching bean cannot be found. The `ObjectNotFoundException` is a subtype of `FindException` and is only thrown by single-entity find methods.

Every find method declared in the local or remote home interface of a CMP 2.0 entity bean must have a matching query declaration in the bean's deployment descriptor. The following snippet from the Customer EJB's deployment descriptor shows declarations two of find methods, `findByName()` and `findByGoodCredit()`, from the examples above.

```
<query>
    <query-method>
        <method-name>findByName</method-name>
        <method-intf>Home</method-intf>
        <method-params>
            <method-params>java.lang.String</method-params>
            <method-params>java.lang.String</method-params>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.lastName = ?1 AND c.firstName = ?1
    </ejb-ql>
</query>
<query>
    <query-method>
        <method-name>findByGoodCredit</method-name>
        <method-intf>LocalHome</method-intf>
        <method-params></method-params>
    </query-method>
    <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.hasGoodCredit = TRUE
```

[1] As of EJB 2.0, these are the only collection types supported for multi-entity query methods. Others, like `java.util.List` and `java.util.Map`, may be added in future versions.

```
    </ejb-ql>
</query>
```

The query elements in the deployment descriptor allow the bean developer to associate EJB QL query statements with specific find methods. When the bean is deployed, the container attempts to match the find method declared in each of the query elements with find methods in entity bean's local and home interfaces. This is done by matching the values of the `<method-name>` and `<method-params>` elements with method names and parameter types (ordering is important) in the home interfaces.

The `<method-intf>` element specifies which home interface (local or remote) the method is defined in. If the find method is declared in the local home interface, then the value `LocalHome` is used. If the find method is declared in the remote home interface, then the value `Home` is used. This element is only needed when two find methods collide, i.e., two find methods in the local and remote home interfaces have the same method name and parameters. Using the `method-intf` element allows the bean developer to specify different EJB QL statements for each method. If `<method-intf>` not specified, and there is a collision, the query declaration will apply to both of the colliding methods. The container will take care of returning the proper type for each colliding query method. The remote home will return a one or more remote EJB objects, and the local home will return one or more local EJB objects. This allows you to define the behavior of colliding local and remote home find methods using a single `query` element, which is convenient if you want local clients to have access to the same find methods as remote clients.

The `<ejb-ql>` element specifies the EJB QL statement for a specific find method. You may have noticed that the EJB QL statement can use input parameters (`?1,?2,…?n`), which are mapped to the `<method-params>` of the find method, as well as literals (e.g. `TRUE`). The use of input parameters and literals will be discussed in more detail through out this chapter.

All single-entity and multi-entity find methods must be declared in `<query>` elements in the deployment descriptor, except for `findByPrimaryKey()` methods. Query declarations for `findByPrimaryKey()` methods are not necessary, and in fact, are forbidden. It's obvious what this method should do, and you may not try to change its behavior.

Select Methods

Select methods are very similar to find methods, but they are more versatile and can only be used internally by the bean class. In other words, select methods are private query methods; they are not exposed to entity bean's clients through the home interfaces.

Select methods are declared as abstract methods using the naming convention `ejbSelect<METHOD-NAME>`. The following code shows four select methods declared in the `AddressBean` class.

```
public class AddressBean implements javax.ejb.EntityBean {
    …
    public abstract String ejbSelectMostPopularCity()
    throws FindException;

    public abstract Set ejbSelectZipCodes(String state)
    throws FindException;

    public abstract Collection ejbSelectAll()
    throws FindException;

    public abstract CustomerLocal ejbSelectCustomer(AddressLocal addr)
    throws FindException;
    …
```

Select methods can return the value of CMP fields. The `ejbSelectMostPopularCity()` select, for example, returns a single `String` value, the name of the city referenced by the most Address EJBs. The `ejbSelectZipCodes()` method returns a `java.util.Set` of `String` values, which is a unique collection of all the zip codes declared for Address EJB's for a specific state.

Select methods can also return EJB objects, just like find methods. The `ejbSelectAll()` method, for example, returns a `java.util.Collection` of EJB objects representing all the Address EJBs in the system. However, unlike find methods, select methods can return any type of EJB object, and are not limited to the type of bean they are declared in. The `ejbSelectCustomer()` method, for example, returns the remote EJB object representing the Customer bean assigned to the specified Address EJB. Notice that the bean type returned is `CustomerLocal`, not `AddressLocal`.

Like find methods, select methods can declare zero or more arguments, which are used to limit the scope of the query. The `ejbSelectZipCodes()` and the `ejbSelectCustomer()` methods both declare arguments used to limit the scope of the results. These arguments will be used as input parameters in the EJB QL statements assigned to the select methods.

Select methods can return local or remote EJB objects. For single-entity select methods, the type is determined by the return type of the `ejbSelect` method. The `ejbSelectCustomer()` method, for example, returns a local EJB object, the `CustomerLocal`. This method could have easily been defined to return a remote EJB object by changing the return type to the Customer bean's remote interface (`CustomerRemote`). Multi-entity select methods, which return a collection of EJB objects, return a collection of local EJB objects by default.

However, the bean provider can override this default behavior using a special element, the `<result-type-mapping>` element, in select method's `<query>` element.

The following portion of an XML deployment descriptor declares two of the select methods from the above example. Notice that they are exactly the same as the find method declarations. Find and select methods are declared in the same part of the deployment descriptor, within an `<entity>` bean element, within the same `<query>` element.

```
<query>
    <query-method>
        <method-name>ejbSelectZipCodes</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
        </method-params>
    </query-method>
    <ejb-ql>
        SELECT a.homeAddress.zip FROM Address AS a
        WHERE a.homeAddress.state = ?1
    </ejb-ql>
</query>
<query>
    <query-method>
        <method-name>ejbSelectAll</method-name>
        <method-params></method-params>
    </query-method>
    <result-type-mapping>Remote</result-type-mapping>
    <ejb-ql>
        SELECT OBJECT(a) FROM Address AS a
    </ejb-ql>
</query>
```

The name given in each `<method-name>` element must match one of the `ejbSelect<METHOD-NAME>()` methods defined in the bean class. This is different from find methods of CMP 2.0 beans, which do not have a corresponding `ejbFind` method in the bean class. For find methods we use the method name in the local or remote home interface. Select methods, on the other hand, are not declared in the local or remote home interface so we use the `ejbSelect` method name in the bean class.

If a select method returns a collection of EJB objects, then the `<result-type-mapping>` can be used to declare if it should return local or remote EJB objects. The value `Local` indicates that a method should return local EJB objects; `Remote` indicates remote EJB objects. If the `<result-type-mapping>` element is not declared, the default is `Local`. In the query element for the `ejbSelectAll` method, the `<result-type-mapping>` is declared

as `Remote`, which means the query should return remote EJB object types; remote references to the Address EJB.

Select methods are not limited to the context of any specific entity bean. They can be used to query across all the entity beans declared in the same deployment descriptor. Select methods may be used by the bean class from its `ejbHome` methods or any business methods or the `ejbLoad` and `ejbStore` methods. The `ejbHome`, `ejbLoad` and `ejbStore` methods are covered in more detail in Chapter 11.

The most important thing to remember about select methods is that they can do anything find methods can and more, but they can only be used by the entity bean class that declares them, not by the entity bean's clients.

EJB QL Examples

EJB QL is expressed in terms of the abstract persistence schema of an entity bean; its abstract schema name, container-managed persistence fields, and container-managed relationship fields. EJB QL uses the abstract schema names to identify beans, the container-managed persistence fields to specify values and container-managed relationship field names to navigate across relationships.

To discuss EJB QL, we will make use of the relationships among the Customer, Address, CreditCard, Cruise, Ship, Reservation, and Cabin defined in Chapter 7. Figure 8-1 is a class diagram that shows the direction and cardinality (multiplicity) of the relationships among these beans.

[Figure 8-1(note this is the same figure as figure 7-23)]

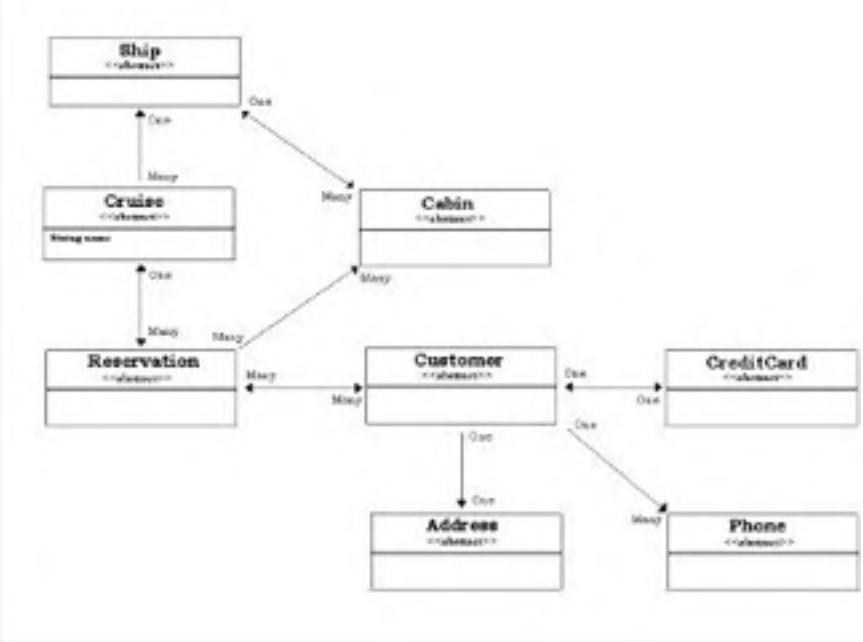


Figure 8-1: Titan Cruises Class Diagram

Simple Queries

The simplest EJB QL statement has no `WHERE` clause and only one abstract schema type. For example, a query method might be defined to select all Customer beans.

```
SELECT OBJECT( c ) FROM Customer AS c
```

The `FROM` clause determines which entity bean types will be included in the select statement. It provides the scope of the select. In this case the `FROM` clause declares the type to be `Customer`, which is the abstract schema name of the Customer EJB. The "`AS c`" part of the clause assigns `c` as the identifier of the Customer EJB. This is similar to SQL, which allows an identifier to be associated with a table. Identifiers can be any length and follow the same rules that are applied to field names in the Java programming language. The following is also perfectly legal.

```
SELECT OBJECT( customer ) FROM Customer AS customer
```

The `AS` operator is optional, but its used in this book to help make the EJB QL statements more clear. The following statement is equivalent:

```
SELECT OBJECT( customer ) FROM Customer customer
```

The `SELECT` clause determines the type of values returned. In this case, it's the Customer entity bean as indicated by the `customer` identifier.

The `OBJECT()` operator is required when the `SELECT` type is an abstract schema identifier (entity bean identifier). The reason for this requirement is pretty vague (and in the author's opinion, the specification would have been better off without it), but it's required whenever the `SELECT` type is an entity bean identifier.

Simple Queries with Paths

EJB QL allows `SELECT` clauses to return any container-managed persistence (CMP) or *single* container-managed relationship (CMR) field. For example, a simple select statement can be defined to return all the last names of all the customers as follows.

```
SELECT c.lastName FROM Customer AS c
```

The `SELECT` clause uses a simple path to select the Customer bean's `lastName` CMP field as the return type. EJB QL uses the CMP and CMR field names declared in `<cmp-field>` and `<cmr-field>` elements of the deployment descriptor. This navigation leverages the same syntax as the Java programming language, specifically the dot ("`.`") navigation operator. For example, compare the above EJB QL statement with the following snippet from the Customer EJB's deployment descriptor:

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            <home> CustomerHomeRemote</ejb-home>
            <remote>CustomerRemote</ejb-remote>
            <ejb-class>CustomerBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Integer</prim-key-class>
            <reentrant>False</reentrant>
            <abstract-schema-name>Customer</abstract-schema-name>
            <cmp-version>2.x</cmp-version>
            <cmp-field><field-name>id</field-name></cmp-field>
            <cmp-field><field-name>lastName</field-name></cmp-field>
            <cmp-field><field-name>firstName</field-name></cmp-field>
```

CMR field types may also be used in simple select statements. For example, the following EJB QL statement selects all the CreditCard EJBs from all the Customer EJBs.

```
SELECT c.creditCard FROM Customer c
```

In this case, the EJB QL statement uses a path to navigate from the Customer EJBs to their `creditCard` relationship fields. The `creditCard` identifier is obtained from the `<cmr-field>` name used in the relationship element that describes the Customer-CreditCard relationship.

```
<enterprise-beans>
```

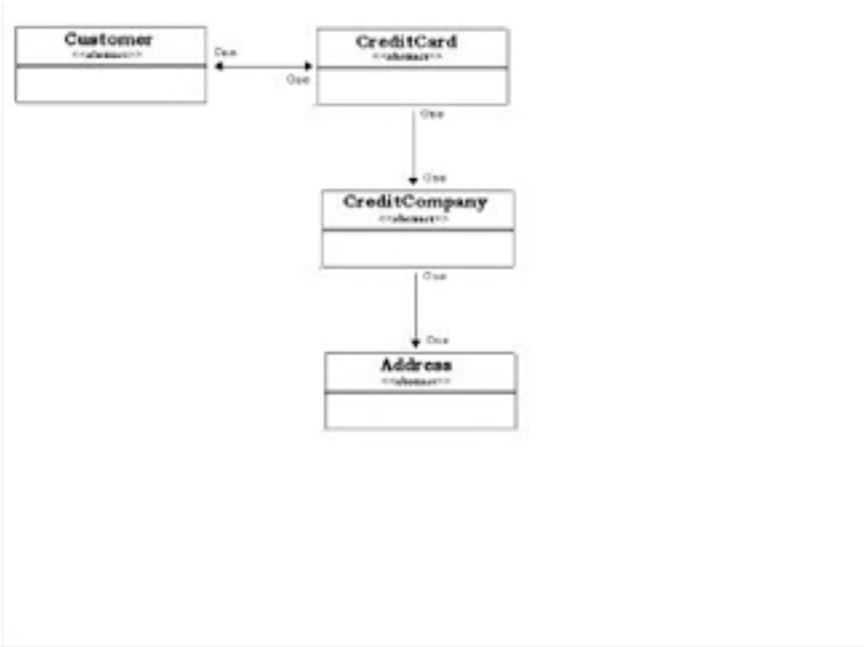
```
        <entity>
            <ejb-name>CustomerEJB</ejb-name>
            …
            <abstract-schema-name>Customer</abstract-schema-name>
        </entity>
</enterprise-beans>
…
<relationships>
    <ejb-relation>
        <ejb-relation-name>Customer-CreditCard
        </ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                Customer-has-a-CreditCard
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <relationship-role-source>
                <ejb-name>CustomerEJB</ejb-name>
            </relationship-role-source>
            <cmr-field>
                <cmr-field-name>creditCard</cmr-field-name>
            </cmr-field>
        </ejb-relationship-role>
        <ejb-relationship-role>
        ...
```

Paths can be as long as required. It's common to use paths that navigate over one or more CMR fields to end at either a CMR or CMP field. For example, the following EJB QL statement selects all the `city` CMP fields of all the Address EJBs of every Customer EJB.

```
SELECT c.homeAddress.city FROM Customer c
```

In this case, the path uses the abstract schema name of the Customer EJB, the Customer EJB's `homeAddress` CMR field and finally the Address EJB's `city` CMP field. Using paths in EJB QL is similar to navigating through object references in the Java language.

To illustrate more complex paths, we'll need to expand the class diagram. Figure 8-2 shows that CreditCard EJB is related to a CreditCompany EJB that has its own Address EJB.



Figureholder

Figure 8-2: Expanded Class Diagram for CreditCard

Using these relationships, a more complex path could be specified that navigates from the Customer EJB to the CreditCompany EJB's Address EJB. The following EJB QL selects all the addresses of all the credit companies.

```
SELECT c.creditCard.creditCompany.address FROM Customer AS c
```

The EJB QL statement could also navigate all the way to the Address bean's CMP fields. For example, the following EJB QL selects all the cities for all the credit card companies for those credit cards used by Titan's customers.

```
SELECT c.creditCard.creditCompany.address.city FROM Customer AS c
```

It's interesting to note that these EJB QL statements would only return `address` CMR fields or Address `city` CMP fields for credit companies of cards owned by Titan's customers. If there are any credit companies whose cards are not currently used by Titan's customers, their address information won't be included in the result.

Paths cannot navigate beyond CMP fields. For example, imagine that the Address EJB uses a `ZipCode` class as its zip CMP field.

```
public class ZipCode implements java.io.Serializable{
    public int mainCode;
    public int codeSuffix;
    …
}
```

It would be illegal to attempt to navigate to one of the `ZipCode` class' instance fields.

```
// this is illegal
SELECT c.homeAddress.zip.mainCode FROM Customer AS c
```

CMP fields cannot be further decomposed and navigated by paths. All CMP fields are considered opaque.

The paths used in a SELECT clause of an EJB QL must always end with a single type. They may not end in a collection-based relationship field. For example, the following is *not* legal because the CMR field `reservations` is a collection-based relationship field.

```
// this is illegal
SELECT c.reservations FROM Customer AS c
```

In fact, it's illegal to navigate across a collection-based relationship field. The following EJB QL statement is also illegal, even though the path ends in a single relationships field.

```
SELECT c.reservations.cruise FROM Customer AS c
```

If you think about it, this limitation makes sense. You cannot use a navigation operator ("`.`") in Java to access elements of a `java.util.Collection` object either. For example, you can't do the following (assume `getReservations()` returns a `java.util.Collection` type).

```
// this is illegal in the Java programming language.
customer.getReservations().getCruise()
```

Referencing the elements of a collection-based relationship field is possible in EJB QL, but it require the use of an `IN` operator and an identification assignment in the `FROM` clause, which are discussed next.

Simple Queries the IN operation

Many relationships between entity beans are collection-based relationships; being able to access and select from these relationships is important. We've seen that it is illegal to select elements directly from a collection-based relationship. To overcome this limitation, EJB QL introduces the `IN` operation, which allows an identifier to represent individual elements in a collection-based relationship field.

The following query uses the `IN` operation to select the elements from a collection-based relationship. It returns all the reservations of all the customers.

```
SELECT OBJECT( r )
FROM Customer AS c,  IN( c.reservations ) AS r
```

The IN operation assigns the individual elements in the reservations CMR field to the identifier r. Once we have an identifier to represent the individual elements of the collection, we can reference them directly and even select them in the EJB QL statement. The element identifier can also be used in path expressions. For example, the following EJB QL statement will select every cruise for which Titan's customers have made reservations.

```
SELECT r.cruise
FROM Customer AS c, IN( c.reservations ) AS r
```

The identifiers assigned in the FROM clause of EJB QL are evaluated from left to right. Once an identifier has been declared it can be used is subsequent declarations in the FROM clause. Notice that the identifier c, which was declared first, was subsequently used in the IN operation to define the identifier r.

The OBJECT() operation is used for single identifiers in the select statement and not for path expressions. While this convention makes little sense, it is none-the-less required by the EJB 2.0 specification. A rule of thumb: If the select type is a solitary identifier, then it must be wrapped in an OBJECT() operation. If the select type is a path expression then it is not.

Identification chains, in which subsequent identifications depend on previous identifications, can become very long. The following EJB QL statement uses two IN operations to navigate two collection-based relationships and a single CMR relationship. While not necessarily useful, this statement demonstrates how a query can use IN operations across many relationships.

```
SELECT cabin.ship
FROM Customer AS c, IN ( c.reservations ) AS r,
IN( r.cabins ) AS cabin
```

📖 Exercise 8.1, Simple EJB QL Statements

The WHERE clause and Literals

Literal values can also be used in the EJB QL to narrow the scope of the elements selected. This is accomplished through the WHERE clause, which behaves in much the same way as the WHERE clause in SQL.

For example, an EJB QL statement can be defined to select all the Customer EJBs that use a specific brand of credit card. The literal in this case is a string literal. Literal strings are enclosed by single quotes. Literal values that include a single quote, like the restaurant name "Wendy's", use two single quotes to escape the quote: 'Wendy''s'. The following statement returns customers that use the American Express credit card:

```
SELECT OBJECT( c ) FROM Customer AS c
```

```
WHERE c.creditCard.organization = 'American Express'
```

Path expressions are always used in the `WHERE` clause in the same way that they're used in the `SELECT` clause. When making comparisons with a literal, the path expression must evaluate to a CMP field; you can't compare a CMR field with a literal.

In addition to literal strings, literal can also be exact numeric values (long types) and approximate numerical values (double types). Exact numerical literal values are expressed using the Java integer literal syntax (`321`, `-8932`, `+22`). Approximate literal values are expressed using Java floating point literal syntax in scientific (`5E3`, `-8.932E5`) or decimal (`5.234`, `38282.2`) notation.

For example, the following EJB QL statement selects all the ships that weigh 100,000.00 metric tons.

```
SELECT OBJECT( s )
FROM Ship AS s
WHERE s.tonnage = 100000.00
```

Boolean literal values use `TRUE` and `FALSE`. Here's an EJB QL statement selects all the customers who have good credit.

```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.hasGoodCredit = TRUE
```

The WHERE clause and Input Parameters

Query methods (find and select methods) that use EJB QL statements may specify method arguments. Input parameters allow those method arguments to be mapped to EJB QL statements and are used to narrow the scope of the query. For example, the `ejbSelectByCity()` method is designed to select all the customers that reside in a particular city and state.

```
public abstract class CustomerBean
implements javax.ejb.EntityBean {
    …
    public abstract Collection ejbSelectByCity(String city,String state)
    throws FindException;
    …
}
```

The EJB QL statement for this method would use the city and state arguments as input parameters.

```
SELECT OBJECT( c ) FROM Customer AS c
WHERE c.homeAddress.state = ?2
AND c.homeAddress.city = ?1
```

Input parameters use a `?` prefix followed by the argument's position, in order of the query method's parameters. In this case, `state` is the second argument and

`city` is the first argument listed in the `ejbSelectByCity()` method. When a query method declares one or more arguments, the associated EJB QL statement may use some or all of the arguments as input parameters.

Input parameters are not limited to simple CMP field types; they can also be EJB object references. For example, the following find method `findByShip()` is declared in the Cruise bean's local interface.

```
public interface CruiseLocal extends javax.ejb.EJBLocalObject {
    public Collection findByShip( ShipLocal customer )
    throws FindException;
}
```

The EJB QL statement associated with this method would use the `ship` argument to locate all the cruises scheduled for the specified Ship bean.

```
SELECT OBJECT( cruise ) FROM Cruise AS cruise
WHERE cruise.ship = ?1
```

When an EJB object is used as an input parameter, the container bases the comparison on the primary key of the EJB object. In this case, it searches through all the Cruise EJBs looking for references to a Ship EJB with same primary key value that the Ship EJB passed to the query method.

The WHERE clause and Operator Precedence

The `WHERE` clause is composed of conditional expressions that reduce the scope of the query and limit the number of items selected. A number of conditional and logical operators can be used in expressions; they are listed below in the order of precedence. The operators at the top of the list have the highest precedence; they are evaluated first.

- Navigation operator (.)
- Arithmetic operators:

  +, - unary

  *, / multiplication and division

  +, - addition and subtraction
- Comparison operators :

  =, >, >=, <, <=, <> (not equal),

  `LIKE`, `BETWEEN`, `IN`, `IS NULL`, `IS EMPTY`, `MEMBER OF`
- Logical operators:

  `NOT`, `AND`, `OR`

If you've been working as a programmer for longer than a month, most of these operators will be familiar to you.

EJB QL statements are declared in XML deployment descriptors. XML uses the greater than ('>') and less than ('<') characters as delimiters for tags, so using these symbols in the EJB QL statements will cause parsing errors unless CDATA sections are used. For example, the following EJB QL statement causes a parsing error, because the XML parser cannot distinguish the use of the '>' symbol from a delimiter to a XML tag:

```
<query>
   <query-method>
      <method-name>findWithPaymentGreaterThan</method-name>
      <method-params>java.lang.Double</method-params>
   </query-method>
   <ejb-ql>
      SELECT OBJECT( r ) FROM Reservation r
      WHERE r.amountPaid  > ?1
   </ejb-ql>
</query>
```

To avoid this problem, the EJB QL statement should be placed in a CDATA section:

```
<query>
   <query-method>
      <method-name>findWithPaymentGreaterThan</method-name>
      <method-params>java.lang.Double</method-params>
   </query-method>
   <ejb-ql>
      <![CDATA[
      SELECT OBJECT( r ) FROM Reservation r
      WHERE r.amountPaid  > 300.00
      ]]>
   </ejb-ql>
</query>
```

The CDATA section takes the form `<![CDATA[ literal-text ]]>`. When an XML processor encounters a CDATA section it doesn't attempt to parse the contents enclosed by the CDATA section, instead the parser treats it as literal text[2].

[2] To learn more about XML and the use of CDATA Sections, see *XML in a Nutshell* by Elliotte Rusty Harold and W. Scott Means published by O'Reilly & Associates 2001.

The WHERE clause and Arithmetic Operators

The arithmetic operators allow a query to perform arithmetic in the process of doing a comparison. In EJB QL, arithmetic operators can only be used in the `WHERE` clause and not in the `SELECT` clause. The following EJB QL statement returns references to all the Reservation EJBs that will be charged a port tax of more than $300.00.

```
SELECT OBJECT( r ) FROM Reservation r
WHERE (r.amountPaid * .01)  > 300.00
```

The rules applied to arithmetic operations are the same as those used in the Java programming language, where numbers are widened or promoted in the process of performing a calculation. For example, multiplying a `double` and an `int` value requires that the `int` first be promoted to a `double` value. The result will always be that of the widest type used in the calculation, so multiplying an `int` and a `double` results in a `double` value.

`String`, `boolean`, and EJB object types cannot be used in arithmetic operations. For example, using the addition operator with two `String` values is considered an illegal operation. There is a special function for concatenating `String` values, which is covered in *The WHERE clause and FUNTIONS section.*

The WHERE clause and Logical Operators

Logical operators such as `AND`, `OR`, and `NOT` operate the same as their corresponding logical operators in SQL.

Logical operators evaluate only boolean expressions, so each operand (each side of the expression) must evaluate to `true` or `false`. This is why the logical operators have the lowest precedence: so that all the expressions can be evaluated before they are applied.

The `AND` and `OR` operations *may not*, however, behave like their Java language counterparts `&&` and `||`. Specifically, EJB QL does not specify whether the right-hand operands are evaluated conditionally. For example, the `&&` operator in Java evaluates its right-hand operand only if the left hand operand is `true`. Similarly, the `||` logical operator evaluates the right-hand operand only if the left-hand operand is `false`. We can't make the same assumption for the `AND` and `OR` operators in EJB QL. Whether these operators evaluate right-hand operands depends on the native query language into which it's translated. It's best to assume that both operands are evaluated on all logical operators.

`NOT` simply reverses the boolean result of its operand; expressions that evaluate to the boolean value of `true` become `false`, and visa versa.

The WHERE clause and Comparison Symbols

Comparison operators, which use the symbols =, >, >=, <, <=, and <>, should be familiar to you. The following statement selects all the Ship EJBs whose tonnage CMP field is greater than or equal to 80,000 tons but less than or equal to 130,000 tons.

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage >= 80000.00 AND s.tonnage <= 130000.00
```

Only the = and <> (not equal) operators may be used on `String`, `boolean`, and EJB object references. The greater-than and less-than symbols (>, >=, <, <=) can only be used on numerical values. It would be illegal, for example, to use the greater-than, or less-than symbols to compare two `String`s. There is no mechanism to compare `String`s in this way in EJB QL.

The WHERE clause and Equality semantics

While it's legal to compare an exact numerical value (`short`, `int`, `long`) to an approximate numerical value (`double`, `float`) all other equality comparisons must compare the exact same types. You cannot, for example, compare a `String` value of '123' to the integer literal 123.

EJB objects can also be compared for equality, but they too must be of the same exact type. To be more specific, they must both be EJB object references to beans of the same deployment. As an example, the following method finds all the Reservation EJBs made by a specific Customer EJB:

```
public interface ReservationHomeLocal extends EJBLocalObject{
    public Collection findByCustomer(CustomerLocal customer)
    throws FindException;
...
}
```

The matching EJB QL statement uses the customer argument as an input parameter.

```
SELECT OBJECT( r )
FROM Reservation r, IN ( r.customers ) customer
WHERE  customer = ?1
```

It's not enough for the EJB object that's used in the comparison to implement the `CustomerLocal` interface; it must be the same bean type as the Customer EJB used in the Reservation's customers CMR Field. In other words, they must be from the same deployment. Once it's determined that the bean is the correct type, the actual comparison is performed on the bean's primary keys. If they have the same primary keys, they are considered equal.

`java.util.Date` objects cannot be used in equality comparisons. In order to compare dates, the long millisecond value of the date must be used, which means that the date must be persisted in a long CMP field and not a `java.util.Date` CMP. The input value or literal must also be a `long` value.

The WHERE clause and BETWEEN

The `BETWEEN` clause is an inclusive operation specifying a range of values. It can be used to select all ships between 80,000 and 130,000 tons.

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage BETWEEN 80000.00 AND 130000.00
```

The `BETWEEN` clause may only be used on numeric primitives (`byte`, `short`, `int`, `long`, `double`, `float`) and their corresponding `java.lang.Number` types (`Byte`, `Short`, `Integer`, etc.). It may not be used on `String`, `boolean`, or EJB object references.

Using the `NOT` logical operator in conjunction with `BETWEEN` excludes the range specified. For example, the following EJB QL statement selects all the Ship EJBs that are less than 80,000 tons or greater then 130,000 tons but excludes everything in-between.

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage NOT BETWEEN 80000.00 AND 130000.00
```

The net effect of this query is the same as if it had been executed with comparative symbols:

```
SELECT OBJECT( s ) FROM Ship s
WHERE s.tonnage < 80000.00 OR s.tonnage > 130000.00
```

The WHERE clause and IN

The `IN` conditional operator used in the `WHERE` clause is not the same as the `IN` operator used in the `FROM` clause. In the `WHERE` clause, `IN` tests for membership in a list of literal string values, and can only be used with operands that evaluate to string values. For example, the following EJB QL statement uses the `IN` operator to select all the customers who reside in a specific set of states:

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress.state IN ('FL', 'TX', 'MI', 'WI', 'MN')
```

Applying the `NOT` operator to this expression reverses the selection, excluding all customers who reside in the list of states:

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress.city
      NOT IN ('FL', 'TX', 'MI', 'WI', 'MN')
```

If the field tested is `null`, the value of the expression is "unknown", which means it cannot be predicted.

The WHERE clause and IS NULL

The `IS NULL` comparison operator allows you to test whether a path expression is `null`. For example, the following EJB QL statement selects all the customers who *do not* have a home address.

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress IS NULL
```

Using the `NOT` logical operator, we can reverse the results of this query, selecting all the customers that *do* have a home address.

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.homeAddress IS NOT NULL
```

When `null` fields appear in comparison operations such as `IN` and `BETWEEN`, they can have pretty serious side affects. In most cases, evaluating a `null` field in a comparison operation (other than `IS NULL`) produces in an `UNKNOWN` result. Unknown evaluations throw the entire EJB QL results set into question; since we cannot predict the outcome the EJB QL statement, it is unreliable. One way to avoid this situation is to require that fields used in the expressions have values. This requires careful programming. To ensure an entity bean field is never `null`, you must initialize the field when the entity is created. For primitive values this not a problem, since they cannot be `null`; they have default values. For other fields, such as single CMR fields and object based CMP fields, like `String`, the fields must be initialized in the `ejbCreate()` and `ejbPostCreate()` methods.

The WHERE clause and IS EMPTY

The `IS EMPTY` operator allows the query to test if a collection-based relationship is empty. Remember from Chapter 7 that a collection-based relationship will never be `null`. If a collection-based relationship field has no elements, it will return an empty `Collection` or `Set`.

Testing whether a collection-based relationship is empty has the same purpose as testing whether single CMR field or CMP field is `null`: it can be used to limit the scope of the query and items selected. For example, the following query selects all the cruises that have *not* booked any reservations:

```
SELECT OBJECT( cruise ) FROM Cruise cruise
WHERE cruise.reservations IS EMPTY
```

The `NOT` operator reverses the result of `IS EMPTY`. The following query selects all the cruises that have at least one reservation.

```
SELECT OBJECT( cruise ) FROM Cruise c
WHERE cruise.reservations IS NOT EMPTY
```

Interestingly, it's illegal to use `IS EMPTY` against collection-based relationships that have been assigned an identifier in the `FROM` clause.

```
// illegal query
SELECT OBJECT( r )
FROM Reservation r, IN( r.customers ) c
WHERE
r.customers IS NOT EMPTY AND
c.address.city = 'Boston'
```

While this query appears to be good insurance against unknown results, it's not. In fact, it's an illegal EJB QL statement, because the `IS EMPTY` operator cannot be used on a collection-based relationship identified in an `IN` operation in the `FROM` clause. Because the relationship is specified in the IN clause, only those Reservation EJBs that have a non-empty `customers` field will be included in the query; any Reservation EJB that has an empty CMR field will be excluded because its customers elements cannot be assigned the `c` identifier.

The WHERE clause and MEMBER OF

The `MEMBER OF` operator is a powerful tool for determining whether an EJB object is a member of a specific collection-based relationship. The following query determines whether a particular Customer (specified by the input parameter) is a member of any of the Reservation-Customer relationships.

```
SELECT OBJECT( cruise )
FROM Cruise cruise, Customer c
WHERE
  c = ?1
  AND
  c MEMBER OF cruise.reservations
```

Applying the `NOT` operator to `MEMBER OF` will have the reverse effect, select all the cruises on which the specified customer doesn't have a reservation.

```
SELECT OBJECT( cruise )
FROM Cruise cruise, Customer c
WHERE
c = ?1
  AND
c NOT MEMBER OF cruise.reservations
```

Checking whether an EJB object is a member of an empty collection always returns `false`.

The WHERE clause and LIKE

The `LIKE` comparison operator allows the query to select `String` type CMP fields that match a specified pattern. For example, the following EJB QL statement selects all the customers with hyphenated names, like "`Monson-Haefel`" and "`Berners-Lee`".

```
SELECT OBJECT( c ) FROM Customer c
WHERE c.lastName LIKE '%-%'
```

Two special characters can be used when establishing a comparison pattern: `'%'` (percent) stands for any sequence of characters, and `'_'` (underscore) stands for any single character. `%` and `_` characters can be used at any location within a string pattern. The escape character `\` can be used if a `%` or `_` actually occurs in the string. The `NOT` logical operator reverses the evaluation so that matching patterns are excluded.

The following examples show how the LIKE clause would evaluate `String` type CMP fields.

- `phone.number LIKE '617%'`

  true for '617-322-4151'

  false for '415-222-3523'

- `cabin.name LIKE 'Suite _100'`

  true for 'Suite A100'

  false for 'Suite A233'

- `phone.number NOT LIKE '608%'`

  true for '415-222-3523'

  false for '608-233-8484'

- `someField.underscored LIKE '\_%'`

  true for '_xyz'

  false for 'abc'

- `someField.percentage LIKE '\%%'`

  true for '% XYZ'

  false for 'ABC'

The WHERE clause and Functional Expressions

EJB QL has six functional expressions that allow for simple `String` manipulation and a couple of basic numerical operations. The `String` functions are listed below:

`CONCAT(String1, String2)`
: returns the `String` that results from concatenating `String1` and `String2`.

`SUBSTRING(String1, start, length)`
: returns the `String` consisting of `length` characters taken from `String1`, starting at the position given by `start`.

`LOCATE(String1, String2 [, start])`
: returns an `int` indicating the position at which `String1` is found within `String2`. If it's present, `start` indicates the character position in `String2` at which the search should start.

`LENGTH(String)`
: returns an `int` indicating the `length` of the string.

The `start` and `length` parameters indicate positions in a `String` as integer values. These expressions can be used in the `WHERE` clause to help refine the scope of the items selected. Here is an example of how the `LOCATE` and `LENGTH` functions might be used:

```
SELECT OBJECT( c )
FROM Customer c
WHERE
LENGTH(c.lastName) > 6
  AND
LOCATE( c.lastName, 'Monson') > -1
```

This EJB QL statement selects all the customers with '`Monson`' somewhere in their last name, but the name must be longer than 6 characters. Therefore, '`Monson-Haefel`' and '`Monson-Ares`' evaluate to `true`, but '`Monson`' returns `false` because it has only 6 characters.

The arithmetic functions are `ABS` and `SQRT`.

`ABS(number)`
: returns the absolute value of a number (`int`, `float`, or `double`)

`SQRT(double)`
: returns the square root of a `double`

📖 Exercise 8.2, Complex EJB QL statements

Problems with EJB QL

EJB QL is a powerful new tool that promises to improve performance, flexibility, and portability of the entity beans in container-managed persistence, but it has some design flaws and omissions.

The OBJECT() operation

The use of the `OBJECT()` operation is unnecessary, cumbersome, and provides little or no value to the bean developer. It's trivial for EJB vendors to determine when an abstract schema type is the return value, so the `OBJECT()` operation provides little real value during query translation. In addition, the `OBJECT()` operation is applied haphazardly. It's required when the return type is an abstract schema identifier, but not when a path expression of the `SELECT` clause ends in a CMR field. Both return an EJB object reference, so the use of `OBJECT()` in one scenario and not the other is illogical and confusing.

When questioned about this, Sun replied that several vendors had requested the use of the `OBJECT()` operations because it will be included in the next major release of the SQL programming language. EJB QL was designed to be similar to SQL because it's the query language that is most familiar to developers, but this doesn't mean it should include functions and operations that have no real meaning in Enterprise JavaBeans.

The missing ORDER BY clause

Soon after you begin using EJB QL you will quickly realize that it's missing a major component, the `ORDER BY` clause. Requesting ordered lists is extremely important in any query language; most major query languages including SQL and object query languages support this concept.

The `ORDER BY` clause has a couple of big advantages: it clearly communicates the bean developer's intentions; and it gives the application server vendors the option of delegating ordering to the database:

- The `ORDER BY` clause would provide a very clear mechanism for the bean developer to communicate his intentions to the EJB QL interpreter. The `ORDER BY` clause is unambiguous; it states exactly how a collection should be ordered (the attributes to order by, ascending, decending, etc.). Given that it's the purpose of EJB QL to clearly describe the behavior of the find and select operations in a portable fashion, `ORDER BY` is clearly a significant omission.
- With an `ORDER BY` clause, EJB QL interpreters used by EJB vendors could, in most cases, choose an ordering mechanism that is optimized for a

particular database. Allowing the resource to perform the ordering is more efficient than having the container do it after the data is retrieved. It was suggested that EJB vendors could provide ordering mechanically, by having the collection sorted after it's obtained. This is a rather ridiculous expectation, since it would require collections to be fully manifested after the query completes, eliminating the advantages of lazy loading.

However, even if the application server vendor chooses to have the container do the ordering, the `ORDER BY` clause still provides the EJB vendor with a clear indication of how to order the collection. It's up to the vendor to choose how to support the `ORDER BY` clause. For databases and other resources that support it, ordering could be delegated to the resource. For those resources that don't support ordering, it can be performed by container. Without an `ORDER BY` clause, the deployer will have to manipulate collections manually or force the container's collection implementations to do the ordering. These two options are untenable in real world applications where performance is critical.

When pressed, Sun explained that the `ORDER BY` clause was not included in this version of the specification because of problems dealing with the mismatch in ordering behavior between the Java language and databases. The example give was string values. The semantics of ordering strings in a database may be different than that of the Java language. For example, Java orders `String` types according to character sequence and case (upper case vs. lower case). Different databases may or may not consider case while ordering or discount leading or trailing white space. In light of these possible differences, it seams like Sun has a reasonable argument, but only for limiting the portability of `ORDER BY`, not for eliminating its use all together. EJB developers can live with less than prefect portability of the `ORDER BY` clause, but they cannot live without the `ORDER BY` clause.

Finally, contrary to popular belief, the `ORDER BY` clause would *not* necessitate the use of the `java.util.List` as a return type. Although the `List` type is supposed to be used for ordered lists, it also allows developers to place items in a specific location of the list, which in EJB would mean a specific location of the database. This is nearly impossible to support, and so appears to be a reasonable argument against using the `ORDER BY` clause. However, this reasoning is flawed, because there is nothing preventing EJB from using the simple `Collection` type for ordered queries. The understanding would be that the items are ordered, but only as long as the collection is not modified after it is obtained. In other words, elements are not added or removed. Another option is to require that EJB QL statements that use the `ORDER BY` clause return a `java.util.Enumeration` type. This seems perfectly reasonable, since the `Collection` received by a select or find operation shouldn't be manipulated anyway.

Lack of support for Date

EJB QL doesn't provide native support for the `java.util.Date` class. This is not acceptable. The `java.util.Date` class should be supported as a natural type in EJB QL. It should be possible, for example, to do comparisons with `Date` CMP fields and literal and input parameters. It should be possible to use comparison symbols (=, >, >=, <, <=, <>) with `Date` CMP fields. It should also be possible to introduce common date functions so that comparisons can be done at different levels, like comparing the day of the week `DOW()` or month (`MONTH()`), etc. Of course, including the `Date` as a supported type in EJB QL is not trivial and problems with interpretation of dates and locals would need to be considered, but the failure to address `Date` as a supported type is a significant omission.

Limited Functional Expressions

While the functional expressions provided by EJB QL will be valuable to developers there are many other functions that should have been included. For example, `COUNT()` is used a lot in real world applications. Other functions that would be useful include (but are not limited to): `CAST()` useful for comparing different types; `MAX()` and `MIN()`; `SUM()`; `UPPER()` and perhaps others. In addition, if support for `java.util.Date` was included in EJB QL, other date functions could be added, like `DOW()`, `MONTH()`, etc.

9

EJB 1.1: Container-Managed Persistence

A Note for EJB 2.0 Readers

Container-managed persistence has undergone a dramatic change in EJB 2.0, which is not backward compatible with EJB 1.1. For that reason, EJB 2.0 vendors must support both EJB 2.0's container-managed persistence model and EJB 1.1's container-managed persistence model. The EJB 1.1 model is supported purely for backward compatibility, so that application developers can migrate their existing applications to the new EJB 2.0 platform as painlessly as possible. It's expected that all new entity beans and new applications will use the EJB 2.0 container-managed persistence, not the EJB 1.1 version. Although EJB 1.1 container-managed persistence is covered in this book, avoid it unless you maintain a legacy EJB 1.1 system. EJB 2.0 container-managed persistence is covered in Chapters 6 thru 8.

In EJB 2.0, EJB 1.1 container-managed persistence is limited in other ways. For example, EJB 1.1 CMP beans can only have remote component interfaces; they are not allowed to have local or local home interfaces. Other subtle differences also make EJB 1.1 CMP more limiting the EJB 2.0. For example, the `ejbCreate()` and `ejbPostCreate()` methods in EJB 1.1 do not support the `<METHOD-NAME>` suffix allowed in EJB 2.0, which makes method overloading more difficult.

Overview for EJB 1.1 Readers

The following overview of EJB 1.1 container-managed persistence is pretty much duplicated in Chapter 6, but for EJB 1.1 readers who have not read Chapter 6, the overview is important to understanding the context of entity beans and container-managed persistence.

In Chapter 4, we started developing some simple enterprise beans, skipping over a lot of the details about developing enterprise beans. In this chapter, we'll take a thorough look at the process of developing entity beans. On the surface, some of this material may look familiar, but it is much more detailed and specific to entity beans.

Entity beans model business concepts that can be expressed as nouns. This is a rule of thumb rather than a requirement, but it helps in determining when a business concept is a candidate for implementation as an entity bean. In grammar school you learned that nouns are words that describe a person, place, or thing. The concepts of person and place are fairly obvious: a person EJB might represent a customer or a passenger, and a place EJB might represent a city or a port-of-call. Similarly, entity beans often represent things: real-world objects like ships, credit cards, and so on. An EJB can even represent a fairly abstract thing, such as a ticket or a reservation. Entity beans describe both the state and behavior of real-world objects and allow developers to encapsulate the data and business rules associated with specific concepts; a Ship EJB encapsulates the data and business rules associated with a ship, and so on. This makes it possible for data associated with a concept to be manipulated consistently and safely.

In Titan's cruise ship business, we can identify hundreds of business concepts that are nouns and therefore could conceivably be modeled by entity beans. We've already seen a simple Cabin EJB in Chapter 4, and we'll develop Ship EJB in this chapter. Titan could clearly make use of a Customer EJB, Cruise EJB, a Reservation EJB, and many others. Each of these business concepts represents data that needs to be tracked and possibly manipulated. Entities really represent data in the database, so changes to an entity bean result in changes to the database.

There are many advantages to using entity beans instead of accessing the database directly. Utilizing entity beans to objectify data provides programmers with a simpler mechanism for accessing and changing data. It is much easier, for example, to change a customer's name by calling `ShipRemote.setName()` than to execute an SQL command against the database. In addition, objectifying the data using entity beans also provides for more software reuse. Once an entity bean has been defined, its definition can be used throughout Titan's system in a consistent manner. The concept of customer, for example, is used in many areas of Titan's business, including booking, scheduling, and marketing. A Ship EJB provides Titan with one complete way of accessing ship information, and thus it

ensures that access to the information is consistent and simple. Representing data as entity beans makes development easier and more cost effective.

When a new EJB is created, a new record must be inserted into the database and a bean instance must be associated with that data. As the EJB is used and its state changes, these changes must be synchronized with the data in the database: entries must be inserted, updated, and removed. The process of coordinating the data represented by a bean instance with the database is called *persistence*.

There are two basic types of entity beans, and they are distinguished by how they manage persistence. *Container-managed persistence* beans have their persistence automatically managed by the EJB container. The container knows how a bean instance's persistent fields and relationships map to the database and automatically takes care of inserting, updating, and deleting the data associated with entities in the database. Entity beans using *bean-managed persistence* do all this work explicitly: the bean developer must write the code to manipulate the database. The EJB container tells the bean instance when it is safe to insert, update, and delete its data from the database, but it provides no other help. The bean instance does all the persistence work itself. Bean-managed persistence is covered in Chapter 10.

Container-Managed Persistence

When you deploy an EJB 1.1 CMP entity bean, you identify which fields in the entity are managed by the container and how they map to the database. Once you have defined the fields that will be automatically managed and how they map to the database, the container generates the logic necessary to save the bean instance's state automatically.

Fields that are mapped to the database are called container-managed fields—EJB 1.1 doesn't support relationship fields, as does EJB 2.0. Container- managed fields can be any Java primitive type or serializable objects. Most beans will use Java primitive types when persisting to a relational database, since it's easier to map Java primitives to relational data types.

EJB 1.1 also allows references to other beans to be container-managed fields. The EJB vendor must support converting bean references (remote or home interface types) from remote references to something that can be persisted in the database and converted back to a remote reference automatically. Vendors will normally convert remote references to primary keys, `Handle` or `HomeHandle` objects, or some other proprietary pointer type, which can be used to preserve the bean reference in the database. The container will manage this conversion from remote reference to persistent pointer and back automatically. This feature was abandoned in EJB 2.0 CMP in favor of container-managed relationship fields.

The advantage of container-managed persistence is that the bean can be defined independently of the database used to store its state. Container-managed beans can take advantage of a relational database or an object-oriented database. The bean state is defined independently, which makes the bean more reusable and flexible.

The disadvantage of container-managed beans is that they require sophisticated mapping tools to define how the bean's fields map to the database. In some cases, this may be a simple matter of mapping each field in the bean instance to a column in the database, or of serializing the bean to a file. In other cases, it may be more difficult. The state of some beans, for example, may be defined in terms of a complex relational database join or mapped to some kind of legacy system such as CICS or IMS.

In this chapter, we will create a new container-managed entity bean, the Ship EJB, which we will examine in detail. A Ship EJB is also used in both Chapter 7, when discussing complex relationships in EJB 2.0, and Chapter 10, when discussing bean-managed persistence. When you are done with this chapter you may want compare the Ship EJB developed here with the ones created in Chapter 7 and 10.

Let's start by thinking about what we're trying to do. An enormous amount of data would go into a complete description of a ship, but for our purposes we will limit the scope of the data to a small set of information. For now, we can say that a ship has the following characteristics or attributes: its name, passenger capacity, and tonnage (i.e., size). The Ship EJB will encapsulate this data; we'll need to create a `SHIP` table in our database to hold this data. Here is the definition for the `SHIP` table expressed in standard SQL:

```
CREATE TABLE SHIP (ID INT PRIMARY KEY, NAME CHAR(30), CAPACITY INT,
TONNAGE DECIMAL(8,2))
```

When defining any bean, we start by coding the remote interfaces. This focuses our attention on the most important aspect of any bean: its business purpose. Once we have defined the interfaces, we can start working on the actual bean definition.

The Remote Interface

For the Ship EJB we will need a remote interface. This interface defines the business methods that clients will use to interact with the bean. When defining the remote interface, we will take into account all the different areas in Titan's system that may want to use the ship concept. Here is the remote interface, `ShipRemote`, for the Ship EJB:

```
package com.titan.ship;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;
```



```
public interface ShipRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setCapacity(int cap) throws RemoteException;
    public int getCapacity() throws RemoteException;
    public double getTonnage() throws RemoteException;
    public void setTonnage(double tons) throws RemoteException;
}
```

The Remote Home Interface

The remote home interface of any entity bean is used to create, locate, and remove objects from EJB systems. Each entity bean type has its own home interface. The home interface defines two basic kinds of methods: zero or more create methods and one or more find methods.[1] The create methods act like remote constructors and define how new Ship EJBs are created. (In our home interface, we only provide a single `create()` method.) The find method is used to locate a specific ship or ships.

The following code contains the complete definition of the `ShipHomeRemote` interface:

```
package com.titan.ship;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Enumeration;

public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote create(Integer id, String name,
                             int capacity, double tonnage)
        throws RemoteException,CreateException;
    public ShipRemote create(Integer id, String name)
        throws RemoteException,CreateException;
    public ShipRemote findByPrimaryKey(Integer primaryKey)
        throws FinderException, RemoteException;
    public Enumeration findByCapacity(int capacity)
        throws FinderException, RemoteException;
}
```

Enterprise JavaBeans specifies that create methods in the home interface must throw the `javax.ejb.CreateException`. In the case of container-

1 Chapter XX explains when you should not define any create methods in the home interface.

managed persistence, the container needs a common exception for communicating problems experienced during the create process.

The find methods

EJB 1.1 CMP only supports find methods, not EJB 2.0's select methods. In addition, find methods are supported by only the remote home interface; local component interfaces are not supported by EJB 1.1 entity beans.

With EJB 1.1 container-managed persistence, implementations of the find methods are generated automatically at deployment time. Different EJB container vendors employ different strategies for defining how the find methods work. Regardless of the implementation, when you deploy the bean, you'll need to do some work to define the rules of the find method. `findByPrimaryKey()` is a standard method that all home interfaces for entity beans must support. This method locates beans based on the attributes of the primary key. In the case of the Ship EJB, the primary key is the `Integer` class, which maps to the `id` field of the `ShipBean`. With relational databases, the primary key attributes usually map to a primary key in a table. In the `ShipBean` class, for example, the `id` attribute maps to the `ID` primary key column in the `SHIP` table. In an object-oriented database, the primary key's attributes might point to some other unique identifier.

EJB 1.1 allows you to specify other find methods in the home interface, in addition to `findByPrimaryKey()`. All find methods must have names that match the pattern `find<SUFFIX>()`. So, for example, if we were to include a find method based on the Ship EJB's capacity, it might be called `findByCapacity(int capacity)`. In container-managed persistence, any find method included in the home interface must be explained to the container. In other words, the deployer needs to define how the find method should work in terms that the container understands. This is done at deployment time, using the vendor's deployment tools and syntax specific to the vendor.

Find methods return either the remote-interface type appropriate for that bean, or an instance of `java.util.Enumeration` or `java.util.Collection` type. Unlike EJB 2.0 CMP, EJB 1.1 CMP doesn't support the `java.util.Set` as a return type from finder methods.

Specifying a remote-interface type indicates that the method only locates one bean. The `findByPrimaryKey()` method obviously returns one remote reference because there is a one-to-one relationship between a primary key's value and an entity. The `findByCapacity(int capacity)` method, however, could return several remote references, one for every ship that has a capacity equal to the parameter `capacity`. The possibility of returning several remote references requires the use of the `Enumeration` type or a `Collection` type. Enterprise JavaBeans specifies that any find method used in a home interface must throw the `javax.ejb.FinderException`. Find

methods that return a single remote reference throw a `FinderException` if an application error occurs, and a `javax.ejb.ObjectNotFoundException` if a matching bean cannot be found. The `ObjectNotFoundException` is a subtype of `FinderException` and is only thrown by find methods that return single remote references.

Find methods that return an `Enumeration` or `Collection` type (multi-entity finders) return an empty collection (not a null reference) if no matching beans can be found, or throw a `FinderException` if an application error occurs.

How find methods are mapped to the database for container-managed persistence is not defined in the EJB 1.1 specification—it is vendor-specific. Consult the documentation provided by your EJB vendor to determine how find methods are defined at deployment time. Unlike EJB 2.0 CMP, there is no standard query language for expressing the behavior of find methods at runtime.

The Primary Key

A primary key is an object that uniquely identifies an entity bean according to the bean type, home interface, and container context from which it is used.

In container-managed persistence, a primary key can be a serializable object defined specifically for the bean by the bean developer, or its definition can be deferred until deployment. The primary key defines attributes that can be used to locate a specific bean in the database. In this case, we need only one attribute, `id`, but in other cases, a primary key may have several attributes, all of which uniquely identify a bean's data. We will examine primary keys in detail in Chapter 11; for now, we specify that the Ship EJB use a simple single-value primary key of type `java.lang.Integer`.

The ShipBean Class

No bean is complete without its implementation class. Now that we have defined the Ship EJB's remote interfaces and primary key, we are ready to define the `ShipBean` itself. The `ShipBean` will reside on the EJB server. When a client application or bean invokes a business method on the Ship EJB's remote interface, that method invocation is received by the EJB object, which then delegates it to the `ShipBean` instance.

When developing any bean, we have to use the bean's remote interfaces as a guide. Business methods defined in the remote interface must be duplicated in the bean class. In container-managed beans, the create methods of the home interface must also have matching methods in the bean class according to the EJB 1.1 specification. Finally, callback methods defined by the `javax.ejb.EntityBean` interface must be implemented. Here is the code for the `ShipBean` class.


```
package com.titan.ship;

import javax.ejb.EntityContext;

public class ShipBean implements javax.ejb.EntityBean {
    public Integer id;
    public String name;
    public int capacity;
    public double tonnage;

    public EntityContext context;

public Integer ejbCreate(Integer id, String name,
                 int capacity, double tonnage) {
        this.id = id;
        this.name = name;
        this.capacity = capacity;
        this.tonnage = tonnage;
        return null;
}
public Integer ejbCreate(Integer id, String name) {
        this.id = id;
        this.name = name;
        capacity = 0;
        tonnage = 0;
        return null;
}

    public void ejbPostCreate(Integer id, String name, int capacity,
                              double tonnage){
        Integer pk = (Integer)context.getPrimaryKey();
        // Do something useful with the primary key.
    }

    public void ejbPostCreate(int id, String name) {
        ShipRemote myself = (ShipRemote)context.getEJBObject();
        // Do something useful with the EJBObject reference.
    }
    public void setEntityContext(EntityContext ctx) {
        context = ctx;
    }
    public void unsetEntityContext() {
        context = null;
    }
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}

    public String getName() {
```



```
        return name;
    }
    public void setName(String n) {
        name = n;
    }
    public void setCapacity(int cap) {
        capacity = cap;
    }
    public int getCapacity() {
        return capacity;
    }
    public double getTonnage() {
        return tonnage;
    }
    public void setTonnage(double tons) {
        tonnage = tons;
    }
}
```

The Ship EJB defines four persistent fields: `id`, `name`, `capacity`, and `tonnage`. No mystery here: these fields represent the persistent state of the Ship EJB; they are the state that defines a unique ship entity in the database. The Ship EJB also defines another field, `context`, which holds the bean's `EntityContext`. We'll have more to say about this later.

The set and get methods are the business methods we defined for the Ship EJB; both the remote interface and the bean class must support them. This means that the signatures of these methods must be exactly the same, except for the `javax.ejb.RemoteException`. The bean class's business methods aren't required to throw the `RemoteException`. This makes sense because these methods aren't actually invoked remotely— they're invoked by the EJB object. If a communication problem occurs, the container will throw the `RemoteException` for the bean automatically.

Implementing the javax.ejb.EntityBean Interface

To make the `ShipBean` an entity bean, it must implement the `javax.ejb.EntityBean` interface. The `EntityBean` interface contains a number of callback methods that the container uses to alert the bean instance of various runtime events:

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean {
    public abstract void ejbActivate() throws RemoteException;
    public abstract void ejbPassivate() throws RemoteException;
    public abstract void ejbLoad() throws RemoteException;
    public abstract void ejbStore() throws RemoteException;
    public abstract void ejbRemove() throws RemoteException;
    public abstract void setEntityContext(EntityContext ctx)
        throws RemoteException;
```



```
    public abstract void unsetEntityContext() throws RemoteException;
}
```

Each callback method is called at a specific time during the life cycle of a `ShipBean`. In many cases, container-managed beans (like the `ShipBean`) don't need to do anything when a callback method is invoked. Container-managed beans have persistence managed automatically, so many of the resources and logic that might be managed by these methods are already handled by the container.

This version of the Ship EJB has empty implementations for its callback methods. It is important to note, however, that even a container-managed bean can take advantage of these callback methods if needed; we just don't need them in our `ShipBean` at this time. The callback methods are examined in detail in Chapter 11. You should read the chapter to learn more about the callback methods and when they are invoked.

The Create Methods

When a create method is invoked on the home interface, the EJB home delegates it to the bean instance in the same way that business methods on the remote interface are handled. This means that we need an `ejbCreate()` method in the bean class that corresponds to each `create()` method in the home interface.

The `ejbCreate()` method returns a `null` value of type `Integer` for the bean's primary key. The return value of the `ejbCreate()` method for a container-managed bean is actually ignored by the container.

> EJB 1.1 changed its return value from `void`, which was the return type in EJB 1.0, to the primary key type to facilitate subclassing; the change was made so that it's easier for a bean-managed bean to extend a container-managed bean. In EJB 1.0, this is not possible because Java won't allow you to overload methods with different return values. By changing this definition so that a bean-managed bean can extend a container-managed bean, the EJB 1.1 specification allows vendors to support container-managed persistence by extending the container-managed bean with a generated bean-managed bean—a fairly simple solution to a difficult problem. Bean developers can also take advantage of inheritance to change an existing CMP bean into a BMP bean, which may be needed to overcome difficult persistence problems.

For every `create()` method defined in the entity bean's home interface, there must be a corresponding `ejbPostCreate()` method in the bean instance class. In other words, `ejbCreate()` and `ejbPostCreate()` methods

occur in pairs with matching signatures; there must be one pair for each `create()` method defined in the home interface.

ejbCreate() and ejbPostCreate

In a container-managed bean, the `ejbCreate()` method is called just prior to writing the bean's container-managed fields to the database. Values passed in to the `ejbCreate()` method should be used to initialize the fields of the bean instance. Once the `ejbCreate()` method completes, a new record, based on the container-managed fields, is written to the database.

The bean developer must ensure that the `ejbCreate()` method sets the persistent fields that correspond to the fields of the primary key. When a primary key is defined for a container-managed bean, it must define fields that match one or more of the container- managed (persistent) fields in the bean class. The fields must match with regard to type and name exactly. At runtime, the container will assume that fields in the primary key match some or all of the fields in the bean class. When a new bean is created, the container will use those container-managed fields in the bean class to instantiated and populate a primary key for the bean automatically.

Once the bean's state has been populated and its `EntityContext` established, an `ejbPostCreate()` method is invoked. This method gives the bean an opportunity to perform any post-processing prior to servicing client requests.

The bean identity isn't available to the bean during the call to `ejbCreate()`, but is available in the `ejbPostCreate()` method. This means that the bean can access its own primary key and EJB object, which can be useful for initializing the bean instance prior to servicing business method invocations. You can use the `ejbPostCreate()` method to perform any additional initialization. Each `ejbPostCreate()` method must have the same parameters as its corresponding `ejbCreate()` method. The `ejbPostCreate()` method returns `void`.

Chapter 11 provides more details about the `ejbCreate()` and `ejbPostCreate()` method and how they relate to the life cycle of entity beans. Consult that chapter for more details about these methods.

Using ejbLoad() and ejbStore() in container-managed beans

The process of ensuring that the database record and the entity bean instance are equivalent is called *synchronization*. In container-managed persistence, the bean's container- managed fields are automatically synchronized with the database. In most cases, we will not need the `ejbLoad()` and `ejbStore()` methods because persistence in container- managed beans is uncomplicated.

Deployment Descriptor

Whether you are using an EJB 2.0 or EJB 1.1 platform, EJB 1.1 CMP entity beans must use the EJB 1.1 deployment descriptor format. You do not use the EJB 2.0 deployment descriptor for deploying EJB 1.1 container-managed persistence entities in a 2.0 platform.

With a complete definition of the Ship EJB, including the remote interface and the home interface, we are ready to create a deployment descriptor. The following listing shows the bean's XML deployment descriptor. The `<cmp-field>` element is particularly important. These elements list the fields that are managed by the container; they have the same meaning as they do in EJB 2.0 container-managed persistence.

```
<?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>
 <enterprise-beans>
   <entity>
      <description>
            This bean represents a cruise ship.
      </description>
      <ejb-name>ShipEJB</ejb-name>
      <home>com.titan.ship.ShipHomeRemote</home>
      <remote>com.titan.ship.ShipRemote</remote>
      <ejb-class>com.titan.ship.ShipBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>1.x</cmp-version>
      <cmp-field><field-name>id</field-name></cmp-field>
      <cmp-field><field-name>name</field-name></cmp-field>
      <cmp-field><field-name>capacity</field-name></cmp-field>
      <cmp-field><field-name>tonnage</field-name></cmp-field>
   </entity>
 </enterprise-beans>

 <assembly-descriptor>
   <security-role>
      <description>
         This role represents everyone who is allowed full access
         to the Ship EJB.
      </description>
     <role-name>everyone</role-name>
   </security-role>

   <method-permission>
```



```
    <role-name>everyone</role-name>
    <method>
        <ejb-name>ShipEJB</ejb-name>
        <method-name>*</method-name>
    </method>
  </method-permission>

  <container-transaction>
    <method>
       <ejb-name>ShipEJB</ejb-name>
       <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

The `<cmp-field>` elements list all the container-managed fields in the entity bean class. These are the fields that will be persisted in the database and are managed by the container at runtime.

📖 Exercise 9.1, CMP 1.1 Entity Bean

10

Bean-Managed Persistence

Bean-Managed Persistence

Bean-managed persistence is more complicated than container-managed persistence because you must explicitly write the persistence logic into the bean class. In order to write the persistence handling code into the bean class, you must know what type of database is being used and the how the bean class's fields map to that database.

Given that container-managed persistence saves a lot of work, why would anyone bother with bean-managed persistence? The advantage of bean-managed persistence is that it gives you more flexibility in how state is managed between the bean instance and the database. Entity beans that use data from a combination of different databases or other resources such as legacy systems can benefit from bean-managed persistence. Essentially, bean-managed persistence is the alternative to container-managed persistence when the container tools are inadequate for mapping the bean instance's state to the backend databases or resource.

The disadvantage of bean-managed persistence is obvious: more work is required to define the bean. You have to understand the structure of the database or resource, the APIs that access them, and develop the logic to create, update, and remove data associated with an entity. This requires diligence in using the EJB callback methods such as `ejbLoad()` and `ejbStore()` appropriately. In addition, you must explicitly develop the find methods defined in the bean's home interfaces.

The select methods used in EJB 2.0 container-managed persistence are not supported in bean-managed persistence.

Another disadvantage of bean-managed persistence is that it ties the bean to a specific database type and structure. Any changes in the database or in the structure of data require changes to the bean instance's definition; these changes may not be trivial. A bean-managed entity is not as database-independent as a container-managed entity, but it can better accommodate a complex or unusual set of data.[1]

To understand how bean-managed persistence works, we will create a new Ship EJB that is similar to the one used in Chapters 7 and 11. For bean-managed persistence, we need to implement the `ejbCreate()`, `ejbLoad()`, `ejbStore()`, and `ejbRemove()` methods to handle synchronizing the bean's state with the database.

The Remote Interface

We will need a remote interface for the Ship EJB. This interface is basically the same as any other remote or local interface. It defines the business methods used by clients to interact with the bean:

```
package com.titan.ship;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface ShipRemote extends javax.ejb.EJBObject {
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setCapacity(int cap) throws RemoteException;
    public int getCapacity() throws RemoteException;
    public double getTonnage() throws RemoteException;
    public void setTonnage(double tons) throws RemoteException;
}
```

In this chapter, we will not develop a local interface for the bean-managed Ship bean; however, in EJB 2.0, bean-managed entity beans can have either local or remote component interfaces, just like CMP.

Set and get methods

The `ShipRemote` definition uses a series of accessor methods whose names begin with `set` and `get`. This is not a required signature pattern, but it is the

1 Containers that use object-to-relational mapping tools in bean-managed persistence can mitigate this disadvantage.

naming convention used by most Java developers when obtaining and changing the values of object attributes or fields. These methods are often referred to as *setters* and *getters* (a.k.a. mutators and accessors) and the attributes that they manipulate can be called *properties*.[2] These properties should be defined independently of the anticipated storage structure of the data. In other words, you should design the remote interface to model the business concepts, not the underlying data. Just because there's a `capacity` property doesn't mean that there has to be a capacity field in the bean or the database; the `getCapacity()` method could conceivably compute the capacity from a list of cabins, by looking up the ship's model and configuration, or with some other algorithm.

Defining entity properties according to the business concept and not the underlying data is not always possible, but you should try to employ this strategy whenever you can. The reason is two-fold. First, the underlying data doesn't always clearly define the business purpose or concept being modeled by the entity bean. Remote interfaces will be used by developers who know the business, not the database configuration. It is important to them that the entity bean reflect the business concept. Second, defining the properties of the entity bean independent of the data allows the bean and data to evolve separately. This is important because it allows a database implementation to change over time; it also allows for new behavior to be added to the entity bean as needed. If the bean's definition is independent of the data source, the impact of these evolutions is limited.

The Remote Home Interface

The home interfaces (local and remote) of any entity bean are used to create, locate, and remove objects from EJB systems. Each entity bean has its own remote or local home interface. The home interface defines two basic kinds of methods: zero or more create methods, and one or more find methods.[3] The create methods act like remote constructors and define how new Ship EJBs are created. (In our home interface, we only provide a single `create()` method.) The find method is used to locate a specific ship or ships.

The following code contains the complete definition of the `ShipHomeRemote` interface:

```
package com.titan.ship;
```

2 Although EJB is different from its GUI counterpart, JavaBeans, the concepts of accessors and properties are similar. You can learn about this idiom by reading *Developing Java Beans™* by Rob Englander (O'Reilly).

3 Chapter XX explains when you should not define any create methods in the home interface.


```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Collection;

public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote create(Integer id, String name,
                             int capacity, double tonnage)
        throws RemoteException,CreateException;
    public ShipRemote create(Integer id, String name)
        throws RemoteException,CreateException;
    public ShipRemote findByPrimaryKey(Integer primaryKey)
        throws FinderException, RemoteException;
    public Collection findByCapacity(int capacity)
        throws FinderException, RemoteException;
}
```

Enterprise JavaBeans specifies that create methods in the home interface must throw the `javax.ejb.CreateException`. This provides the EJB container with a common exception for communicating problems experienced during the create process.

The `RemoteException` is thrown by all remote interfaces and is used to communicate network problems that occurred while processing invocations between a remote client and the EJB container system.

The Primary Key

In bean-managed persistence, a primary key can be a serializable object defined specifically for the bean by the bean developer. The primary key defines attributes that can be used to locate a specific bean in the database. For the `ShipBean`, we need only one attribute, `id`, but in other cases, a primary key may have several attributes, which taken together uniquely identify a bean's data.

We will examine primary keys in detail in Chapter 11; for now, we specify that the Ship EJB uses a simple single-value primary key of type `java.lang.Integer`. The actual persistence field in the bean class is an `Integer` named `id`.

The ShipBean

The `ShipBean` defined for this chapter uses JDBC to synchronize the bean's state to the database. In reality, an entity bean that is this simple could easily be deployed as a container-managed persistence bean. The purpose of this chapter,

however, is to illustrate exactly where the resource access code goes for bean-managed persistence and how to implement it. The fact that we are synchronizing the bean state against a relational database is not important to the example. The bean could be persisted to some legacy system, or an ERP application, or some other resource that is not supported by your vendor's EJB container-managed persistence, like LDAP or some hierarchical database.

So when learning about bean-managed persistence you should focus on when and where the resource is accessed to synchronize the bean with the database, and not be overly concerned with the fact that this example use JDBC and a relational database.

Here is the complete definition of the `ShipBean`:

```
package com.titan.ship;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.ejb.EntityContext;
import java.rmi.RemoteException;
import java.sql.SQLException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import javax.sql.DataSource;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.FinderException;
import javax.ejb.ObjectNotFoundException;
import java.util.Enumeration;
import java.util.Properties;
import java.util.Vector;
import java.util.Collection;

public class ShipBean implements javax.ejb.EntityBean {
    public Integer id;
    public String name;
    public int capacity;
    public double tonnage;

    public EntityContext context;

    public Integer ejbCreate(Integer id, String name,
        int capacity, double tonnage)
        throws CreateException {
        if ((id.intValue() < 1) || (name == null))
           throw new CreateException("Invalid Parameters");
        this.id = id;
```

```
        this.name = name;
        this.capacity = capacity;
        this.tonnage = tonnage;

        Connection con = null;
        PreparedStatement ps = null;
        try {
          con = this.getConnection();
          ps = con.prepareStatement(
           "insert into Ship (id, name, capacity, tonnage) " +
           "values (?,?,?,?)");
          ps.setInt(1, id.intValue());
          ps.setString(2, name);
          ps.setInt(3, capacity);
          ps.setDouble(4, tonnage);
          if (ps.executeUpdate() != 1) {
            throw new CreateException ("Failed to add Ship to database");
          }
           return id;
        }
        catch (SQLException se) {
            throw new EJBException (se);
        }
        finally {
          try {
            if (ps != null) ps.close();
            if (con!= null) con.close();
          } catch(SQLException se) {
            se.printStackTrace();
          }
        }
    }
    public void ejbPostCreate(Integer id, String name,
        int capacity, double tonnage) {
        // Do something useful with the primary key.
    }
    public Integer ejbCreate(Integer id, String name )
        throws CreateException {
        return ejbCreate(id,name,0,0);
    }
    public void ejbPostCreate(int id, String name) {
        // Do something useful with the EJBObject reference.
    }
    public Integer ejbFindByPrimaryKey(Integer primaryKey)
        throws FinderException {
        Connection con = null;
        PreparedStatement ps = null;
        ResultSet result = null;
        try {
          con = this.getConnection();
          ps = con.prepareStatement(
```

```
            "select id from Ship where id = ?");
        ps.setInt(1, primaryKey.intValue());
        result = ps.executeQuery();
        // Does ship id exist in database?
        if (!result.next()) {
          throw new ObjectNotFoundException(
              "Cannot find Ship with id = "+id);
        }
      } catch (SQLException se) {
          throw new EJBException(se);
      }
      finally {
        try {
          if (result != null) result.close();
          if (ps != null) ps.close();
          if (con!= null) con.close();
        } catch(SQLException se){
          se.printStackTrace();
        }
      }
      return primaryKey;
  }
  public Collection ejbFindByCapacity(int capacity)
      throws FinderException {
      Connection con = null;
      PreparedStatement ps = null;
      ResultSet result = null;
      try {
        con = this.getConnection();
        ps = con.prepareStatement(
            "select id from Ship where capacity = ?");
        ps.setInt(1,capacity);
        result = ps.executeQuery();
        Vector keys = new Vector();
        while(result.next()) {
           keys.addElement(result.getObject("id"));
        }
        return keys;

      }
      catch (SQLException se) {
        throw new EJBException (se);
      }
      finally {
        try {
          if (result != null) result.close();
          if (ps != null) ps.close();
          if (con!= null) con.close();
        } catch(SQLException se) {
          se.printStackTrace();
        }
```



```
    }
}
public void setEntityContext(EntityContext ctx) {
    context = ctx;
}
public void unsetEntityContext() {
    context = null;
}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbLoad() {

    Integer primaryKey = (Integer)context.getPrimaryKey();
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet result = null;
    try {
      con = this.getConnection();
      ps = con.prepareStatement(
          "select name, capacity, tonnage from Ship where id = ?");
      ps.setInt(1, primaryKey.intValue());
      result = ps.executeQuery();
      if (result.next()){
        id = pk.intValue();
        name = result.getString("name");
        capacity = result.getInt("capacity");
        tonnage = result.getDouble("tonnage");
      } else {
        throw new EJBException();
      }
    } catch (SQLException se) {
        throw new EJBException(se);
    }
    finally {
      try {
        if (result != null) result.close();
        if (ps != null) ps.close();
        if (con!= null) con.close();
      } catch(SQLException se) {
        se.printStackTrace();
      }
    }
}
public void ejbStore() {
    Connection con = null;
    PreparedStatement ps = null;
    try {
      con = this.getConnection();
      ps = con.prepareStatement(
          "update Ship set name = ?, capacity = ?, " +
          "tonnage = ? where id = ?");
```

```
            ps.setString(1,name);
            ps.setInt(2,capacity);
            ps.setDouble(3,tonnage);
            ps.setInt(4,id.intValue());
            if (ps.executeUpdate() != 1) {
               throw new EJBException("ejbStore");
            }
          }
          catch (SQLException se) {
               throw new EJBException (se);
          }
          finally {
            try {
              if (ps != null) ps.close();
              if (con!= null) con.close();
            } catch(SQLException se) {
              se.printStackTrace();
            }
          }
      }
      public void ejbRemove() {
          Connection con = null;
          PreparedStatement ps = null;
          try {
            con = this.getConnection();
            ps = con.prepareStatement("delete from Ship where id = ?");
            ps.setInt(1, id.intValue());
            if (ps.executeUpdate() != 1) {
                throw new EJBException("ejbRemove");
            }
          }
          catch (SQLException se) {
            throw new EJBException (se);
          }
          finally {
            try {
              if (ps != null) ps.close();
              if (con!= null) con.close();
            } catch(SQLException se) {
              se.printStackTrace();
            }
          }
      }
      public String getName() {
          return name;
      }
      public void setName(String n) {
          name = n;
      }
      public void setCapacity(int cap) {
          capacity = cap;
```

```
    }
    public int getCapacity() {
        return capacity;
    }
    public double getTonnage() {
        return tonnage;
    }
    public void setTonnage(double tons) {
        tonnage = tons;
    }
    private Connection getConnection() throws SQLException {
         // Implementations for EJB 1.0 and EJB 1.1 shown below
   }
}
```

Obtaining a Resource Connection

In order for a BMP entity bean to work, it must have access to the database or resource that it will persist itself to. To get access to the database, the bean usually obtains a resource factory from the JNDI ENC. The JNDI ENC is covered in detail in chapter 12, *Session beans*, but an overview here will be helpful since this is the first time its actually used. To get access to the database we simply request a connection from a `DataSource`, which we obtain from the JNDI environment naming context:

```
private Connection getConnection() throws SQLException {
    try {
      Context jndiCntx = new InitialContext();
      DataSource ds =
         (DataSource)jndiCntx.lookup("java:comp/env/jdbc/titanDB");
      return ds.getConnection();
    }
    catch (NamingException ne) {
      throw new EJBException(ne);
    }
}
```

In EJB, every enterprise bean has access to its JNDI environment naming context (ENC), which is part of the bean-container contract. The bean's deployment descriptor maps resources such as the JDBC DataSource, JavaMail, and Java Message Service to a context (name) in the ENC. This provides a portable model for accessing these types of resources. Here's the relevant portion of the deployment descriptor that describes the JDBC resource:

```
<enterprise-beans>
   <entity>
     <ejb-name>ShipEJB</ejb-name>
     ...
     <resource-ref>
          <description>DataSource for the Titan database</description>
          <res-ref-name>jdbc/titanDB</res-ref-name>
```



```
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        <resource-ref>
          ...
      <entity>
...
<enterprise-beans>
```

The `<resource-ref>` tag is used for any resource (JDBC, JMS, JavaMail) that is accessed from the ENC. It describes the JNDI name of the resource (`<res-ref-name>`), the factory type (`<res-type>`), and whether authentication is performed explicitly by the bean or automatically by the container (`<res-auth>`). In this example, we are declaring that the JNDI name `"jdbc/titanDB"` refers to a `javax.sql.DataSource` resource manager, and that authentication to the database is handle automatically by the container. The JNDI name specified in the `<res-ref-name>` tag is always relative to the standard JNDI ENC context name, `"java:comp/env"`.

When the bean is deployed, the deployer maps the information in the `<resource-ref>` tag to a live database. This is done in a vendor-specific manner, but the end result is the same. When a database connection is requested using the JNDI name `"java:comp/jdbc/titanDB"`, a `DataSource` for the Titan database is returned. Consult your vendor's documentation for details on how to map the `DataSource` to the database at deployment time.

The `getConnection()` method provides us with a simple and consistent mechanism for obtaining a database connection for our `ShipBean` class. Now that we have a mechanism for obtaining a database connection, we can use it to insert, update, delete, and find Ship EJBs in the database.

Exception Handling

Exception handling is particularly relevant in our discussion of bean-managed persistence because, unlike container-managed persistence, the bean developer is responsible for throwing the correct exceptions at the right moments. For this reason we'll take a moment to discuss different types of exceptions in bean-managed persistence. This discussion will be useful when we get into the details of database access and implementing the callback methods.

There are three types of exceptions thrown from a bean: application exceptions, which indicate business logic errors, runtime exceptions, and checked subsystem exceptions, which are thrown from subsystems like JDBC or JNDI.

Application exceptions

Application exceptions include standard EJB application exceptions and custom application exceptions. The standard EJB application exceptions are `CreateException`, `FinderException`,

`ObjectNotFoundException`, `DuplicateKeyException`, and `RemoveException`. These exceptions are thrown from the appropriate methods to indicate that a business logic error has occurred. Custom exceptions are exceptions you develop for specific business problems. You will develop custom exceptions in Chapter 12, *Session beans.*

Runtime exceptions

`RuntimeException` types are thrown from the virtual machine itself and indicate that a fairly serious programming error has occurred. Examples include `NullPointerException` and `IndexOutOfBoundsException`. These exceptions are handled by the container automatically and should not be handled inside a bean method.

You will notice that all the callback methods (`ejbLoad`, `ejbStore`, `ejbActivate`, `ejbPassivate`, and `ejbRemove`) throw an `EJBException` when a serious problem occurs. All EJB callback methods declare the `EJBException` and `RemoteException` in their `throws` clause. If you need to throw an exception from one of the callback methods, it must be an `EJBException` or a subclass. The `RemoteException` type is included in the method signature to support backward compatibility with EJB 1.0 beans. Its use has been deprecated since EJB 1.1. `RemoteExceptions` should never be thrown by callback methods of EJB 1.1 or EJB 2.0 beans.

Subsystem exceptions

Checked exceptions thrown by other subsystems should be wrapped in an `EJBException` or application exception and re-thrown from the method. Several examples of this can be found in the previous example, in which an `SQLException` that was thrown from JDBC was caught and rethrown as an `EJBException`. Checked exceptions from other subsystems, such as those thrown from JNDI, JavaMail, JMS, etc., should be handled in the same fashion. The `EJBException` is a subtype of the `RuntimeException`, so it doesn't need to be declared in the method's `throws` clause. If the exception thrown by the subsystem is not serious, you can opt to throw an application exception, but this is not recommended unless you are sure of the cause and affect of the exception on the subsystem. In the majority of cases, throwing an `EJBException` is prefered.

Exceptions have an impact on transactions and are fundamental to transaction processing. Exceptions are examined in greater detail in Chapter 14, *Transactions.*

The ejbCreate() Method

The `ejbCreate()` methods are called by the container when a client invokes the corresponding `create()` method on the bean's home. With bean-managed persistence, the `ejbCreate()` methods are responsible for adding the new entity to the database. This means that the new version of `ejbCreate()` will be much more complicated than the equivalent methods in container-managed

entities; with container-managed beans, `ejbCreate()` doesn't have to do much more than initialize a few fields. The EJB specification also states that `ejbCreate()` methods in bean-managed persistence must return the primary key of the newly created entity. This is another difference between bean-managed and container-managed persistence; in our container-managed beans, `ejbCreate()` is required to return `void`.

The following code contains the `ejbCreate()` method of the `ShipBean`. Its return type is the Ship EJB's primary key, `Integer`. Furthermore, the method uses the JDBC API to insert a new record into the database based on the information passed as parameters.

```
public Integer ejbCreate(Integer id, String name,
    int capacity, double tonnage)
    throws CreateException {
    if ((id.intValue() < 1) || (name == null))
        throw new CreateException("Invalid Parameters");
    this.id = id;
    this.name = name;
    this.capacity = capacity;
    this.tonnage = tonnage;

    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement(
        "insert into Ship (id, name, capacity, tonnage) " +
        "values (?,?,?,?)");
        ps.setInt(1, id.intValue());
        ps.setString(2, name);
        ps.setInt(3, capacity);
        ps.setDouble(4, tonnage);
        if (ps.executeUpdate() != 1) {
        throw new CreateException ("Failed to add Ship to database");
        }
        return id;
    }
    catch (SQLException se) {
        throw new EJBException (se);
    }
    finally {
        try {
        if (ps != null) ps.close();
        if (con!= null) con.close();
        } catch(SQLException se) {
        se.printStackTrace();
        }
    }
}
```

At the beginning of the method, we verify that the parameters are correct, and throw a `CreateException` if the `id` is less than 1, or the `name` is `null`. This shows how you would typically use a `CreateException` to report an application logic error.

The `ShipBean` instance fields are still initialized using the parameters passed to `ejbCreate()` by setting the instance fields of the ShipBean. These values will be used to manually insert the data into the `SHIP` table in our database.

To perform the database insert, we use a JDBC `PreparedStatement` for SQL requests because it makes it easier to see the parameters being used. Alternatively, we could have used a stored procedure through a JDBC `CallableStatement` or a simple JDBC `Statement` object. We insert the new bean into the database using a SQL `INSERT` statement and the values passed into `ejbCreate()` parameters. If the insert is successful (no exceptions thrown), we create a primary key and return it to the container.

If the insert operation is unsuccessful, we throw a new `CreateException`, which illustrates its use in more ambiguous situation. Failure to insert the record could be construed as an application error or a system failure. In this situation, the JDBC subsystem hasn't thrown an exception, so we shouldn't interpret the inability to insert a record as a failure of the subsystem. Therefore, we throw a `CreateException` instead of an `EJBException`. Throwing a `CreateException` provides the application the opportunity to recover from the error, a transactional concept that is covered in more detail in Chapter 14, *Transactions.*

After the insert operation is successful, the primary key is returned to the EJB container from the `ejbCreate()` method. In this case we simply return the same `Integer` object passed into the method, but in many cases a new key might be derived from the method arguments. This is especially true when using compound primary keys, which are discussed in Chapter 11. Behind the scenes, the container uses the primary key and the `ShipBean` instance that returned it to provide the client with a reference to the new Ship entity. Conceptually, this means that the `ShipBean` instance and primary key are assigned to a newly constructed EJB object, and the EJB object stub is returned to the client.

Our home interface requires us to provide a second `ejbCreate()` method with different parameters. We can save work and write more bulletproof code by making the second method call the first:

```
public Integer ejbCreate(Integer id, String name )
throws CreateException {
    return ejbCreate(id,name,0,0);
}
```


The ejbLoad() and ejbStore() Methods

Throughout the life of an entity, its data will be changed by client applications. In the `ShipBean`, we provide accessor methods to change the `name`, `capacity`, and `tonnage` of the Ship EJB after it has been created. Invoking any of these accessor methods changes the state of the `ShipBean` instance, which must be reflected in the database.

In container-managed persistence, synchronization between the entity bean and the database takes place automatically; the container handles it for you. With bean-managed persistence, you are responsible for synchronization: the entity bean must read and write to the database directly. The container works closely with the bean-managed persistence entities by advising them when to synchronize their state through the use of two callback methods: `ejbStore()` and `ejbLoad()`.

The `ejbStore()` method is called when the container decides that it is a good time to write the entity bean's data to the database. The container makes these decisions based on all the activities it is managing, including transactions, concurrency, and resource management. Vendor implementations may differ slightly as to when the `ejbStore()` method is called, but this is not the bean developer's concern. In most cases, the `ejbStore()` method will be called after a business method has been invoked or at the end of a transaction. Here is the `ejbStore()` method for the `ShipBean`:

```
public void ejbStore() {
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement(
            "update Ship set name = ?, capacity = ?, " +
            "tonnage = ? where id = ?");
        ps.setString(1,name);
        ps.setInt(2,capacity);
        ps.setDouble(3,tonnage);
        ps.setInt(4,id.intValue());
        if (ps.executeUpdate() != 1) {
            throw new EJBException("ejbStore");
        }
    }
    catch (SQLException se) {
            throw new EJBException (se);
    }
    finally {
        try {
        if (ps != null) ps.close();
        if (con!= null) con.close();
        } catch(SQLException se) {
```



```
        se.printStackTrace();
        }
    }
}
```

Except for the fact that we are doing an update instead of an insert, this method is similar to the `ejbCreate()` method we examined earlier. A JDBC `PreparedStatement` is employed to execute the SQL `UPDATE` command, and the entity bean's persistent fields are used as parameters to the request. This method synchronizes the database with the state of the bean.

EJB also provides an `ejbLoad()` method that synchronizes the state of the entity with the database. This method is usually called prior to a new transaction or business method invocation. The idea is to make sure that the bean always represents the most current data in the database, which could be changed by other beans or other non-EJB applications. Here is the `ejbLoad()` method for a bean-managed `ShipBean` class:

```
public void ejbLoad() {

    Integer primaryKey = (Integer)context.getPrimaryKey();
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet result = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement(
            "select name, capacity, tonnage from Ship where id = ?");
        ps.setInt(1, primaryKey.intValue());
        result = ps.executeQuery();
        if (result.next()){
        id = primaryKey;
        name = result.getString("name");
        capacity = result.getInt("capacity");
        tonnage = result.getDouble("tonnage");
        } else {
        throw new EJBException();
        }
    } catch (SQLException se) {
        throw new EJBException(se);
    }
    finally {
        try {
        if (result != null) result.close();
        if (ps != null) ps.close();
        if (con!= null) con.close();
        } catch(SQLException se) {
        se.printStackTrace();
        }
    }
}
```

To execute the `ejbLoad()` method we need a primary key. To get a primary key, we query the bean's `EntityContext`. Note that we don't get the primary key directly from the `ShipBean`'s `id` field because we cannot guarantee that this field is always valid—the `ejbLoad()` method might be populating the bean instance's state for the first time, in which case the fields would all be set to their default values. This situation would occur following bean activation. We can guarantee that the `EntityContext` for the `ShipBean` is valid because the EJB specification requires that the bean instance `EntityContext` reference is valid before the `ejbLoad()` method can be invoked. The `EntityContext` will be discussed in detail in Chapter 11.

You may want to jump to Chapter 11 and read the section titled *EntityContext* so that you have a better understanding of its purpose and usefulness in entity beans.

The ejbRemove() Method

In addition to handling their own inserts and updates, bean-managed entities must also handle their own deletions. When a client application invokes the remove method on the EJB home or EJB object, that method invocation is delegated to the bean-managed entity by calling `ejbRemove()`. It is the bean developer's responsibility to implement an `ejbRemove()` method that deletes the entity's data from the database. Here's the `ejbRemove()` method for our bean-managed `ShipBean`:

```
public void ejbRemove() {
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement("delete from Ship where id = ?");
        ps.setInt(1, id.intValue());
        if (ps.executeUpdate() != 1) {
            throw new EJBException("ejbRemove");
        }
    }
    catch (SQLException se) {
        throw new EJBException (se);
    }
    finally {
        try {
        if (ps != null) ps.close();
        if (con!= null) con.close();
        } catch(SQLException se) {
        se.printStackTrace();
        }
    }
}
```

ejbFind() Methods

In bean-managed persistence, the find methods in the remote or local home interface must match the `ejbFind` methods in the actual bean class. In other words, for each method named `find<SUFFIX>()` in a home interface, there must be a corresponding `ejbFind<SUFFIX>()` method in the entity bean class with the same arguments and exceptions. When a find method is invoked on an EJB home, the container delegates the `find` method to a corresponding `ejbFind` method on the bean instance. The bean-managed entity is responsible for locating records that match the find requests. In `ShipHomeRemote`, there are two find methods:

```
public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote findByPrimaryKey(Integer primaryKey)
      throws FinderException, RemoteException;
    public Enumeration findByCapacity(int capacity)
      throws FinderException, RemoteException;
}
```

And here are the signatures of the corresponding `ejbFind` methods in the `ShipBean`:

```
public class ShipBean extends javax.ejb.EntityBean {

    public Integer ejbFindByPrimaryKey(Integer primaryKey)
      throws FinderException, RemoteException {}
    public Collection ejbFindByCapacity(int capacity)
      throws FinderException, RemoteException {}
}
```

Aside from the names, there's one difference between these two groups of methods. The find methods in the home interface return either an EJB object implementing the bean's remote interface—in this case, `ShipRemote`—or a collection of EJB objects in the form of a `java.util.Enumeration` or `java.util.Collection`. The `ejbFind` methods in the bean class, on the other hand, return either a primary key for the appropriate bean—in this case, `Integer`—or a collection of primary keys. The methods that return a single value (whether a remote/local interface or a primary key) are used whenever you need to look up a single reference to a bean. If you are looking up a group of references (for example, all ships with a certain capacity), you have to use the method that returns either the `Collection` or `Enumeration` type. In either case, the container intercepts the primary keys and converts them into remote references for the client.

The EJB 2.0 specification recommends that EJB 2.0 bean-managed persistence beans use the `Collection` type instead of the `Enumeration` type. This recommendation is probably made so that bean-managed persistence beans are

more consistent with EJB 2.0 container-managed persistence beans, which use the `Collection` type. However, unlike EJB 2.0 container-managed persistence beans, bean-managed persistence beans do not support `java.util.Set` as a return type.

It shouldn't come as a surprise that the type returned—whether it's a primary key or a remote (or local in EJB 2.0) interface—must be appropriate for the type of bean you're defining. For example, you shouldn't put find methods in a Ship EJB to look up and return Cabin EJB objects. If you need to return collections of a different bean type, use a business method in the remote interface, not a find method from one of the home interfaces.

In EJB 2.0, the EJB container takes care of returning the proper (local or remote) interface to the client. For example, the Ship EJB may define both a local and remote home interface both of which have a `findByPrimaryKey()` method. When `findByPrimary()`is invoked on the local or remote interface, it will be delegated to the same `ejbFindByPrimary()` key method. After the `ejbFindByPrimaryKey()` method executes and returns the primary key, the EJB container takes care of returning a `ShipRemote` or `ShipLocal` reference to the client, depending on which home interface (local or remote) was used. The EJB container also handles this for multi-entity find methods, returning a collection of remote references for remote home interfaces and local references for local home interfaces.

Both find methods defined in the `ShipBean` class throw a `FinderException` if a failure in the request occurs when an SQL exception condition is encountered. The `findByPrimaryKey()` throws the `ObjectNotFoundException` if there are no records in the database that match the `id` argument. This is exception should always be thrown by single-entity find methods if no entity is found.

The `findByCapacity()` method returns an empty collection if no `SHIP` records were found with a matching capacity; multi-entity find methods do *not* throw an `ObjectNotFoundException` if no entities are found. Find methods also throw `FinderException` and `EJBException`, in addition to any application-specific exceptions that the bean developer considers appropriate.

It is mandatory that all entity remote and local home interfaces include the method `findByPrimaryKey()`. This method returns the remote interface type, `Ship`. The method declares one parameter, the primary key for that bean type. With local home interfaces, the return type of any single-entity finder method is always the bean's local interface. With remote home interfaces, the return type of any single-entity find method is always the remote interface. You

cannot deploy an entity bean that doesn't include a `findByPrimaryKey()` method in its home interfaces.

Following the rules outlined earlier, we can define two `ejbFind` methods in `ShipBean` that match the two find methods defined in the `ShipHome`:

```
public Integer ejbFindByPrimaryKey(Integer primaryKey)
    throws FinderException, {
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet result = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement(
            "select id from Ship where id = ?");
        ps.setInt(1, primaryKey.intValue());
        result = ps.executeQuery();
        // Does ship id exist in database?
        if (!result.next()) {
        throw new ObjectNotFoundException(
            "Cannot find Ship with id = "+id);
        }
    } catch (SQLException se) {
        throw new EJBException(se);
    }
    finally {
        try {
        if (result != null) result.close();
        if (ps != null) ps.close();
        if (con!= null) con.close();
        } catch(SQLException se){
        se.printStackTrace();
        }
    }
    return primaryKey;
}
public Collection ejbFindByCapacity(int capacity)
    throws FinderException {
    Connection con = null;
    PreparedStatement ps = null;
    ResultSet result = null;
    try {
        con = this.getConnection();
        ps = con.prepareStatement(
            "select id from Ship where capacity = ?");
        ps.setInt(1,capacity);
        result = ps.executeQuery();
        Vector keys = new Vector();
        while(result.next()) {
            keys.addElement(result.getObject("id"));
        }
```

```
            return keys

        }
        catch (SQLException se) {
            throw new EJBException (se);
        }
        finally {
            try {
            if (result != null) result.close();
            if (ps != null) ps.close();
            if (con!= null) con.close();
            } catch(SQLException se) {
            se.printStackTrace();
            }
        }
}
```

The mandatory `findByPrimaryKey()` method uses the primary key to locate the corresponding database record. Once it has verified that the record exists, it simply returns the primary key to the container, which then uses the key to activate a new instance and associate it with that primary key at the appropriate time. If there is no record associated with the primary key, the method throws an `ObjectNotFoundException`.

The `ejbFindByCapacity()` method returns a `Collection` of primary keys that match the criteria passed into the method. Again, we construct a prepared statement that we use to execute our SQL query. This time, however, we expect multiple results so we use the `java.sql.ResultSet` to iterate through the results, creating a vector of primary keys for each `SHIP_ID` returned.

Find methods are not executed on bean instances that are currently supporting a client application. Only bean instances that are not assigned to an EJB object (instances in the instance pool) are supposed to service find requests, which means that the `ejbFind()` methods in the bean instance have somewhat limited use of the `EntityContext`. The `EntityContext` methods `getPrimaryKey()` and `getEJBObject()` will throw exceptions because the bean instance is in the pool and is not associated with a primary key or EJB object when the `ejbFind` method is called.

Where do the objects returned by a finder method come from? This seems like a simple enough question, but the answer is surprisingly complex. Remember that a finder method isn't executed by a bean instance that is actually supporting the client; the container selects an idle bean instance from the instance pool to execute the method. The container is responsible for creating the EJB objects and local or remote references for the primary keys returned by the `ejbFind` method in the bean class. As the client accesses these remote references, bean instances

are swapped into the appropriate EJB objects, loaded with data, and made ready to service the client's requests.

Deployment Descriptor

With a complete definition of the Ship EJB, including the remote interface, home interface, and primary key, we are ready to create a deployment descriptor. Here are the XML deployment descriptors for EJB 1.1 and 2.0. These deployment descriptors are a little different from the descriptors we created for the container-managed entity beans in Chapters 6, 7, and 10. In this deployment descriptor, the `persistence-type` is `Bean` and there are no container-managed or relationship field declarations. We also must declare the `DataSource` resource factory that we use to query and update the database.

Here is the deployment descriptor for EJB 2.0:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
 <enterprise-beans>
   <entity>
      <description>
            This bean represents a cruise ship.
      </description>
      <ejb-name>ShipEJB</ejb-name>
      <home>com.titan.ship.ShipHomeRemote</home>
      <remote>com.titan.ship.ShipRemote</remote>
      <ejb-class>com.titan.ship.ShipBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <security-identity><use-callers-identity/><security-identity>
      <resource-ref>
           <description>DataSource for the Titan database</description>
           <res-ref-name>jdbc/titanDB</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
      </resource-ref>

  </entity>
 </enterprise-beans>

 <assembly-descriptor>
   <security-role>
      <description>
         This role represents everyone who is allowed full access
```



```
        to the Ship EJB.
      </description>
    <role-name>everyone</role-name>
  </security-role>

  <method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>ShipEJB</ejb-name>
        <method-name>*</method-name>
    </method>
  </method-permission>

  <container-transaction>
    <method>
       <ejb-name>ShipEJB</ejb-name>
       <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>

</ejb-jar>
```

The EJB 1.1 deployment descriptor is exactly the same except for two things: the `<!DOCTYPE>` element references EJB 1.1 instead of 2.0:

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

And the `<security-identity>` element is specific to EJB 2.0 and would not be in the EJB 1.1 deployment descriptor.

```
<security-identity><use-callers-identity/><security-identity>
```

📖 Exercise 10.1, Bean-Managed Persistence

11

Entity-Container Contract

Although each of the three entity type components (EJB 2.0 CMP, EJB 1.1 CMP, and BMP) are programmed differently, their relationships to the container system at runtime are very similar. This chapter covers the relationship between EJBs and their containers, which includes areas like primary keys, callback methods, and the entity bean lifecycle. When differences between the bean types are important, they will be noted.

The Primary Key

A primary key is an object that uniquely identifies a specific type of entity bean. A primary key can be any serializable type including primitive wrappers (`Integer`, etc.) or custom classes defined by the bean developer. In the Ship EJB (Chapters 7, 9, and 10) we used the `Integer` type as a primary key. Primary keys can be declared by the bean developer, or the primary key type can be deferred until deployment. We will talk about deferred primary keys later.

Because the primary key may be used in remote invocations, it must adhere to the restrictions imposed by Java RMI-IIOP. These are addressed in Chapter 5, but for most cases, you just need to make the primary key serializable. In addition, the primary key must be a valid Java RMI-IIOP value type; and it must implement `equals()` and `hashCode()` appropriately.

EJB allows two types of primary keys: compound and single-field keys. Single-field primary keys map to a single persistent field defined in the bean class. The Customer and Ship EJBs, for example, use a `java.lang.Integer` primary key that maps to the container-managed persistence (CMP) field named `id`. A

compound primary key is a custom defined object that contains several instance variables that map to more than one persistent field in the bean class.

Single-field key

The `String` class and the standard wrapper classes for the primitive data types (`java.lang.Integer`, `java.lang.Double`, etc.) can be used as primary keys. These are referred to as single-field primary keys because the primary key is atomic; it maps only to one of the bean's persistent fields. A compound primary key, discussed next, maps a primary key to two or more persistent fields. In the case of the Ship EJB, we specified an `Integer` type as the primary key in the finder methods

```
public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public Ship findByPrimaryKey(java.lang.Integer primarykey)
        throws FinderException, RemoteException;
    ...
}
```

In this case, there must be a single persistent field in the bean class with the same matching type as the primary key. For the `ShipBean`, the `id` CMP field is of type `java.lang.Integer`, so it maps well to the `Integer` primary key type.

In EJB 2.0 container-managed persistence, the primary key type must map to one of the CMP fields. The abstract accessor methods for the `id` field in the `ShipBean` class fit this description.

```
public class ShipBean implements javax.ejb.EntityBean {
    public abstract Integer getId( );
    public abstract void setId(Integer id);
    …
}
```

In bean-managed persistence (Chapter 10) and EJB 1.1 container-managed persistence (Chapter 9) the single-field primary key maps to a container-managed persistent field. For the `ShipBean` defined in Chapters 9 and 10, the `Integer` primary key would map to the `id` instance field.

```
public class ShipBean implements javax.ejb.EntityBean {
    public Integer id;
    public String name;
    …
}
```

With single-field types, you identify the matching persistent field in the bean class using the `primkey-field` element in the deployment descriptor to specify one of the bean's CMP fields as the primary key. The `prim-key-class` element specifies the type of object used for the primary key class. The

Ship EJB uses both of these elements when defining the `id` persistent field as the primary key.

```
<entity>
   <ejb-name>ShipEJB</ejb-name>
   <home>com.titan.ShipHomeRemote</ejb-home>
   <remote>com.titan.ShipRemote</ejb-remote>
   <ejb-class>com.titan.ShipBean</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>java.lang.Integer</prim-key-class>
   <reentrant>False</reentrant>
   <cmp-field><field-name>id</field-name></cmp-field>
   <cmp-field><field-name>name</field-name></cmp-field>
   <cmp-field><field-name>tonnage</field-name></cmp-field>
   <primkey-field>id</primkey-field>
</entity>
```

Although primary keys can be primitive wrappers (`Integer`, `Double`, `Long`, etc.), primary keys cannot be primitive types (`int`, `double`, `long`, etc.); some of the semantics of EJB interfaces prohibit the use of primitives. For example, the `EJBObject.getPrimaryKey()` method returns an `Object` type, thus forcing primary keys to be `Object`s. Primitives also cannot be primary keys because primary keys must implement the `equals()` and `hashcode()` methods, so they can be managed in collections. Primitives are not objects and do not have `equals()` or `hashcode()` methods.

Compound primary keys

A compound primary key is a class that implements `java.io.Serializable` and contains one or more public fields whose names and types match a subset of persistent fields in the bean class. These types of primary keys are classes defined by the bean developer for a specific entity bean.

For example, if a Ship EJB didn't have an `id` field, we might uniquely identify ships by their name and registration number. (We are adding the `registration` CMP to the Ship EJB for this example.) In this case the `name` and `registration` CMP fields would become our primary key fields. To accommodate multiple fields as a primary key we need to define a primary key class.

In this book, it's a convention to define all compound primary keys as serializable classes with names that match the pattern `BeanNamePK`. In this case we can construct a new class called `ShipPK`, which serves as the compound primary key for our Ship EJB.

```
public class ShipPK implements java.io.Serializable {
    public String name;
```



```
    public String registration;

    public ShipPK(){
    }
    public ShipPK(String name, String registration){
        this.name = name;
        this.registration = registration;
    }
    public String getName() {
        return name;
    }
    public String getRegistration() {
        return registration;
    }
    public boolean equals(Object obj){
        if (obj == null || !(obj instanceof ShipPK))
            return false;

        ShipPK other = (ShipPK)obj;
        if(this.name.equals(other.name) andand
           this.registration.equals(other.registration))
           return true;
        else
           return false;

    }
    public int hashCode(){
      return name.hashCode()^registration.hashCode();
    }

    public String toString(){
      return name+" "+registration;
    }

}
```

To make the `ShipPK` class work as a compound primary key we must make its fields public. This allows the container system to use reflection when synchronizing the values in the primary key class with the persistent fields in the bean class. In addition, we must define an `equals()` and `hashCode()` method so that the primary key can be easily manipulated within collections, which is often needed by container systems and application developers alike.

It's important to make sure that the variables declared in the primary key have corresponding CMP fields in the entity bean with matching identifiers (names) and data types. This is required so that the container, using reflection, can match the variables declared in the compound key to the correct CMP fields in the bean class. In this case, the `name` and `registration` instance variables declared in the `ShipPK` class correspond to `name` and `registration` CMP fields in the Ship EJB, so it's a good match.

We have also overridden the `toString()` method to return a meaningful value. The default implementation defined in `Object` returns the class name of the object appended to the object identity for that name space.

The `ShipPK` class defines two constructors: a no-argument constructor and an overloaded constructor that sets the `name` and `registration` variables. The overloaded constructor is a convenience method for developers that reduces the number of steps required to create a primary key. The no-argument constructor is *required* for container-managed persistence. When a new bean is created, the container automatically instantiated the primary key using the `Class.newInstance()` method, and populates it from the bean class's container-managed fields. A no-argument constructor must exist in order for that to work.

To accommodate the `ShipPK` we change the `ejbCreate()`/`ejbPostCreate()` methods so that they have name and registration arguments to set the primary key fields in the bean. Here is how the `ShipPK` primary key class would be used in EJB 2.0 using the Ship EJB's bean class that was developed in Chapter 7:

```
import javax.ejb.EntityContext;
import javax.ejb.CreateException;

public abstract class ShipBean implements javax.ejb.EntityBean {

    public ShipPK ejbCreate(String name, String registration){
        setName(name);
        setRegistration(registration);
        return null;
    }
    public void ejbPostCreate(String name, String registration){
    }
    ...
```

In EJB 1.1 container-managed persistence, the container-managed fields are set directly. Here is an example of how this would be done with the Ship EJB in CMP 1.1:

```
public class ShipBean implements javax.ejb.EntityBean {
    public String name;
    public String registration;

    public ShipPK ejbCreate(String name, String registration){
        this.name = name;
        this.registration = registration;
        return null;
    }
```

In bean-managed persistence, the Ship EJB sets its instance fields, instantiate the primary key, and return it to the container.


```
public class ShipBean implements javax.ejb.EntityBean {
    public String name;
    public String registration;

    public ShipPK ejbCreate(String name, String registration){
        this.name = name;
        this.registration = registration;
        ...
        // database insert logic goes here
        ...
        return new ShipPK(name, registration);
    }
```

The `ejbCreate()` method now returns the `ShipPK` as the primary key type. The `ejbCreate()` method of entity beans must be defined with a return type matching the primary key type if it's defined—it returns the type `java.lang.Object` if it is undefined.

In EJB 2.0 container-managed persistence, if the primary key fields are defined—if they are accessible through abstract accessor methods—then they *must* be set in the `ejbCreate()` method. Undefined or deferred primary keys are explained in the next section. While the return type of the `ejbCreate()` method is always the primary key type, the value returned must always be `null`. The EJB container itself takes care of extracting the proper primary key directly. In bean-managed persistence, the bean class is responsible for constructing the primary key and returning it to the container.

The `ShipHomeRemote` interface must be modified so that it uses the name and registration arguments in the `create()` method and the `ShipPK` in the `findByPrimaryKey()` method—EJB requires that we use the primary key type in that method.

```
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.FinderException;

public interface ShipHomeRemote extends javax.ejb.EJBHome {

    public ShipRemote create(String name, String registration)
    throws CreateException, RemoteException;

    public ShipRemote findByPrimaryKey(ShipPK primaryKey)
    throws FinderException, RemoteException;

}
```

`setName()` and `setRegistration()`, which modify the `name` and `registration` of the Ship EJB, should not be declared in the remote or local interfaces of the bean. The primary key of an entity bean must *not* be changed

once the bean is created. However, methods that simply read the primary key fields may be exposed because they don't change the key's values.

EJB 2.0 specifies that the primary key may only be set once in the `ejbCreate()` method or, if it's undefined, automatically by the container when the bean is created. Once the bean is created the primary key fields *must never* be modified by the bean or one of its clients. This is a reasonable requirement that should also be applied to EJB 1.1 CMP and bean-managed persistence beans, because the primary key is the unique identifier of the bean. Changing it could violate referential integrity in the database, resulting in two beans mapped to the same identifier or breaking relationships with other beans based on value of the primary key.

Undefined primary keys

Undefined primary keys for container-managed persistence were introduced in EJB 1.1, but they didn't realize their full potential until EJB 2.0. Basically, undefined primary keys allow the bean developer to defer declaring the primary key to the deployer, which makes it possible to utilize auto-generated primary keys and to create more portable entity beans.

Undefined Primary Keys and Auto-Generated Values

An advantage of the undefined primary key is that the primary key's value can be automatically generated by the database or resources. The most popular relational databases, for example, allow fields to be defined so that they auto-increment or otherwise auto-generate values when a new record is inserted. This is convenient as it allows new records to be allotted a unique primary key; the application code doesn't have to invent unique identifiers every time a new record is added.

To facilitate an undefined primary key, the bean class and its interfaces use the `Object` type to identify the primary key. In Chapter 6 the Address EJB was introduced, which uses an undefined primary key. The following shows the `ejbCreate()` method as returning an Object type.

```
public abstract class AddressBean extends javax.ejb.EntityBean {

    public Object ejbCreateAddress
                        (String street, String city,
                         String state,  String zip )
    {
         setStreet(street);
         setCity(city);
         setState(state);
         setZip(zip);
         return null;
    }
```

The `findByPrimaryKey()` method defined in the local and remote home interface must also use an `Object` type.

```
public interface AddressLocalHome extends javax.ejb.EJBLocalHome {

    public AddressLocal findByPrimaryKey(Object primaryKey)
                    throws javax.ejb.FinderException;

}
```

Finally, the deployment descriptor of the Address EJB defines its primary key type as `java.lang.Object`, and does not define any `prim-key-field` elements.

```
<ejb-jar>
    <enterprise-beans>
        <entity>
            <ejb-name>AddressEJB</ejb-name>
            <local-home>AddressLocalHome</local-home>
            <local>AddressLocal</local>
            <ejb-class>AddressBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>java.lang.Object</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field><field-name>street</field-name></cmp-field>
            <cmp-field><field-name>city</field-name></cmp-field>
            <cmp-field><field-name>state</field-name></cmp-field>
            <cmp-field><field-name>zip</field-name></cmp-field>
        </entity>
```

The use of an undefined primary key means that the bean developer and application developer (client code) must work with a `java.lang.Object` type and not a specific primary key type, which can be limiting. For example, it's not possible to construct an undefined primary key to use in a finder method if you don't know its type. This limitation can be quite daunting if you need to locate an entity bean by its primary key. However, entity beans with undefined primary keys can be easily located using other query methods that do not depend on the primary key value, so this limitation is not a serious handicap.

In the case of the Address EJB we derived a great deal of value from using an undefined primary key. It allowed us to create new Address beans without having to worry about what the value of the primary key should be. If using auto-generated primary keys makes your life easier, feel free to use undefined primary keys throughout your system. Just be aware of the limitations mentioned above.

Undefined Primary Keys and Portability

Another advantage of undefined primary keys is that they can improve the portability of entity beans across different databases and resources. One

problem with container-managed persistence in EJB 1.0 was that the entity bean developer had to define the primary key before the entity bean was deployed. In turn, this requirement forced the developer to make assumptions about the environment in which the entity bean would be used, which limited the entity bean's portability across databases. For example, a relational database uses a set of columns in a table as the primary key, to which entity bean's fields map nicely. An object database, however, uses a completely different mechanism for indexing objects, to which a primary key may not map very well. The same is true for legacy systems and Enterprise Resource Planing (ERP) systems.

An undefined primary key allows the deployer to choose a system-specific key at deployment time. An object database may generate an Object ID, while an ERP system may generate some other primary key. These keys are generated by the database or backend system automatically. This may require that the CMP bean be altered or extended to support the key, but this is immaterial to the bean developer; she concentrates on the business logic of the bean and leaves the indexing to the container.

In bean-managed persistence an undefined primary key can be declared by simply making the primary key type `java.lang.Object`. However, this is pure semantics; the primary key value will not be auto-generated by the container because the bean developer has total control over persistence. In this case the bean developer would still need to use a valid primary key, but its type would be hidden from the bean clients. This could be useful if the primary key type is expected to change over time.

The Callback Methods

All entity beans (container- or bean-managed) must implement the `javax.ejb.EntityBean` interface. The `EntityBean` interface contains a number of callback methods that the container uses to alert the bean instance of various runtime events:

```
public interface javax.ejb.EntityBean extends javax.ejb.EnterpriseBean {
    public abstract void ejbActivate()
        throws EJBException, RemoteException;
    public abstract void ejbPassivate()
        throws EJBException, RemoteException;
    public abstract void ejbLoad() throws EJBException, RemoteException;
    public abstract void ejbStore() throws EJBException, RemoteException;
    public abstract void ejbRemove()
        throws EJBException, RemoteException;
    public abstract void setEntityContext(EntityContext ctx)
    throws EJBException, RemoteException;
    public abstract void unsetEntityContext() throws EJBException,
        RemoteException;
}
```


Each callback method is invoked on an entity bean instance at a specific time during its life cycle.

As described in Chapter 10, BMP beans must implement most of these callback methods to synchronize the bean's state with the database. The `ejbLoad()` method tells the BMP bean when to read its state from the database; `ejbStore()` tells it when to write to the database; and `ejbRemove()` tells the bean when to delete itself from the database.

While bean-managed persistence beans take full advantage of callback methods, CMP entity beans may not use them at all. CMP entity beans have persistence managed automatically, so many of the resources and logic that might be managed by these methods are already handled by the container. However, even a CMP entity bean can take advantage of these callback methods if needed; we just don't need them in any of the container-managed entity beans defined in this book.

You will have noticed that each method in the `EntityBean` interface throws both `javax.ejb.EJBException` and `java.rmi.RemoteException`. EJB 1.0 required that a `RemoteException` be thrown if a system exception occurred while bean executed a callback method. However, since EJB 1.1, the use of `RemoteException` in these methods has been deprecated in favor of the `javax.ejb.EJBException`. EJB 1.1 and EJB 2.0 require that the `EJBException` be thrown if a system error, like a `SQLException`, is encountered while executing a method. The `EJBException` is a subclass of `RuntimeException`, so you don't have to declare it in the method signature. At any rate, you don't have to declare the `RemoteException` when implementing the callback methods, and it's recommended that you don't.

setEntityContext() and unsetEntityContext()

The first method called after a bean instance is instantiated is `setEntityContext()`. As the method signature indicates, this method passes the bean instance a reference to a `javax.ejb.EntityContext`, which is really the bean instance's interface to the container. The purpose and functionality of the `EntityContext` is covered in detail later in this chapter.

The `setEntityContext()` method is called prior to the bean instance's entry into the instance pool. In Chapter 3, we discussed the instance pool that EJB containers maintain, where instances of entity and stateless session beans are kept ready to use. `EntityBean` instances in the instance pool are not associated with any data in the database; their state is not unique. When a client requests a specific entity, an instance from the pool is chosen, populated with data from the database, and assigned to service the client. It is recommended that any non-managed resources needed for the life of the instance be obtained

when this method is called. This ensures that such resources are only obtained once in the life of a bean instance. A non-managed resource is one that is not automatically managed by the container (e.g., references to CORBA objects). Only resources that are not specific to the entity bean's identity should be obtained in the `setEntityContext()`. Other managed resources (e.g. Java Message Service factories) and entity bean references are obtained as needed from the JNDI ENC. Bean references and managed resources obtained through the JNDI ENC are not available from the `setEntityContext()`. The JNDI ENC is covered in detail later in this chapter.

At the end of the entity bean instance's life, after the entity bean instance is removed permanently from the instance pool and before it's garbage collected, the `unsetEntityContext()` method is called, indicating that the bean instance's `EntityContext` is no longer implemented by the container. This is a good time to free up any resources obtained in the `setEntityContext()` method.

ejbCreate()

In a CMP bean, the `ejbCreate()` method is called just prior to writing the bean's state to the database. Values passed in to the `ejbCreate()` method should be used to initialize the CMP fields of the bean instance. Once the `ejbCreate()` method completes, a new record, based on the persistent fields, is written to the database.

In bean-managed persistence, the `ejbCreate()` method is called when it's time for the bean to add itself to the database. Inside the `ejbCreate()` method, a BMP bean must use some kind of API to insert its data into the database.

Each `ejbCreate()` method must have parameters that match a `create()` method in the home interface. If you look at the `ShipBean` class definition and compare it to the Ship EJB's home interface (Chapters 7, 9, and 10), you can see how the parameters for the create methods match exactly in type and sequence. This enables the container to delegate the `create()` method on the home interface to the proper `ejbCreate()` method in the bean instance.

In EJB 2.0, the `ejbCreate()` method can take the form `ejbCreate<SUFFIX>()`, which allows for easier method overloading when parameters are the same but the methods act differently. For example, `ejbCreateByName(String name)` and `ejbCreateByRegistration(String registration)` would have corresponding create methods defined in the local or home interface of the form `createByName(String name)` and `createByRegistration(String registration)`.

EJB 1.1 CMP does not allow the use of suffixes on `ejbCreate()` names. The `ejbCreate()` and `create()` methods may only differ by the number and type of parameters defined.

The `EntityContext` maintained by the bean instance does not provide an entity bean with the proper identity until the `ejbCreate()` method has completed. This means that during the course of the `ejbCreate()` method, the bean instance doesn't have access to its primary key or EJB object. The `EntityContext` does, however, provide the bean with information about the caller's identity, access to its EJB home object (local and remote), and properties. The bean can also use the JNDI naming context to access other beans and resource managers like `javax.sql.DataSource`.

The CMP entity bean developer must, however, ensure that the `ejbCreate()` method sets the persistent fields that correspond to the fields of the primary key. When a new CMP entity bean is created, the container will use the CMP fields in the bean class to instantiate and populate a primary key automatically. In the case of an undefined primary key, the container and database will work together to generate the primary key for the entity bean.

Once the bean's state has been populated and its `EntityContext` established, the `ejbPostCreate()` method is invoked. This method gives the bean an opportunity to perform any post-processing prior to servicing client requests. In EJB 2.0 CMP entity beans, the `ejbPostCreate()` method is used to manipulate container-managed relationship (CMR) fields. In EJB 2.0 container-managed persistence, CMR fields must not be modified in the `ejbCreate()` method. The reason for this restriction has to do with referential integrity; in order for two beans to have a relationship, both must exist. In the case of a relational database, for example, relationships between data may not be possible unless both parties have records in the database. There could be a referential integrity constraint that says a foreign key value cannot be used if the corresponding record doesn't exist. Requiring that the `ejbCreate` method complete before CMR fields are modified ensures that the entity bean's record is inserted into the database before attempting to link it to other records.

ejbPostCreate()

The bean identity isn't available during the call to `ejbCreate()`, but is available in the `ejbPostCreate()` method. This means that the bean can access its own primary key and EJB object (local or remote) inside of `ejbPostCreate()`. This can be useful for performing post processing prior to servicing business method invocations—in CMP 2.0 it can be used for initializing CMR fields of the entity bean.

Each `ejbPostCreate()` method must have the same parameters as its corresponding `ejbCreate()` method as well as the same method name. For example, if the `ShipBean` class defines an `ejbCreateByName(String name)` method, it must also define a matching `ejbPostCreateByName(String name)` method. The `ejbPostCreate()` method returns `void`. In EJB 1.1 CMP, suffixes are not allowed on create methods. Only the parameter lists may differ between `ejbPostCreate()` methods, but the method names must be `ejbPostCreate`.

Matching the parameter lists of `ejbCreate()` and `ejbPostCreate()` methods is important for a couple of reasons. First, it indicates which `ejbPostCreate()` method is associated with which `ejbCreate()` method. This ensures that the container calls the correct `ejbPostCreate()` method after `ejbCreate()` is done. Second, it is possible that one of the parameters passed is not assigned to a persistent field. In this case, you would need to duplicate the parameters of the `ejbCreate()` method to have that information available in the `ejbPostCreate()` method.

In EJB 2.0 container-managed persistence, relationship fields are the primary reason for utilizing the `ejbPostCreate()` method, because of referential integrity (discussed in the previous section on `ejbCreate()`).

ejbCreate() and ejbPostCreate() sequence of events

To understand how an entity bean instance gets up and running, we have to think of a entity bean in the context of its life cycle. Figure 11-1 shows the sequence of events during a portion of a container-managed persistence bean's life cycle, as defined by the EJB specification. Every EJB vendor must support this sequence of events.

[FIGURE a modified version of Figure 6-1 from the 2nd edition]

Figure 11-1: Event sequence for bean instance creation

The process begins when the client invokes one of the `create()` methods on the bean's EJB home. A `create()` method is invoked on the EJB home stub (step 1), which communicates the method to the EJB home across the network (step 2). The EJB home plucks a `ShipBean` instance from the pool and invokes its corresponding `ejbCreate()` method (step 3).

The `create()` and `ejbCreate()` methods are responsible for initializing the bean instance so that the container can insert a record into the database. In the case of the `ShipBean`, the minimal information required to add a new customer to the system is the customer's unique `id`. This CMP field is initialized during the `ejbCreate()` method invocation (step 4).

In container-managed persistence (EJB 2.0 and 1.1), the container uses the bean bean's CMP fields (`id`, `name`, `tonnage`) to insert a record in the database, which it reads from the bean (step 5). Only the fields described as CMP fields in the deployment descriptor are accessed. Once the container has read the CMP fields from the bean instance, it will automatically insert a new record into the database using those fields (step 6). How the data is written to the database is defined when the bean's fields are mapped at deployment time. In our example, a new record is inserted into the `CUSTOMER` table.

> In bean-managed persistence, the bean class itself reads the fields and performs a database insert to add the bean's data to the database. This would take place in steps 5 and 6.

Once the record has been inserted into the database, the bean instance is ready to be assigned to an EJB object (step 7). Once the bean is assigned to an EJB object, the bean's identity is available. This is when the `ejbPostCreate()` method is invoked (step 8).

In EJB 2.0 CMP entity beans the `ejbPostCreate()` method is used to manage the entity beans container-managed relationship fields. This might involve setting the Cruise, in the Ship EJB's cruise CMP field or some other relationship (step 9).

Finally, when the `ejbPostCreate()` processing is complete, the bean is ready to service client requests. The EJB object stub is created and returned to client application, which will use it to invoke business methods on the bean (step 10).

Using ejbLoad() and ejbStore() in container-managed persistence

The process of ensuring that the database record and the entity bean instance are equivalent is called *synchronization*. In container-managed persistence, the bean's CMP fields are automatically synchronized with the database. In most cases, we will not need the `ejbLoad()` and `ejbStore()` methods because persistence in container-managed beans is fairly straightforward.

Leveraging the `ejbLoad()` and `ejbStore()` callback methods in container-managed beans, however, can be useful if custom logic is needed when synchronizing CMP fields. Data intended for the database can be reformatted or compressed to conserve space; data just retrieved from the database can be used to calculate derived values for non-persistent fields.

Imagine a hypothetical bean class that includes some binary value that you want to store in the database. The binary value may be very large (an image for example), so you need to compress it before storing it away. Using the

`ejbLoad()` and `ejbStore()` methods in a container-managed bean allows the bean instance to reformat the data as appropriate for the state of the bean and the structure of the database. Here's how this might work:

```
import java.util.zip.Inflater;
import java.util.zip.Deflater;

public abstract class HypotheticalBean implements javax.ejb.EntityBean {
    // instance variable
    public byte [] inflatedImage;

    // CMP field methods
    public abstract void setImage(byte [] image);
    public abstract byte [] getImage( );

    // business methods. Used by client.
    public byte [] getImageFile( ){
        if(inflatedImage == null){
            Inflater unzipper = new Inflater();
            byte [] temp = getImage();
            unzipper.setInput(temp);
            unzipper.inflate(inflatedImage);
        }
        return inflatedImage;
    }
    public void setImageFile(byte [] image){
         inflatedImage = image;
    }

    // callback methods
    public void ejbLoad(){
        inflatedImage = null;
    }
    public void ejbStore(){
        if(inflatedImage != null){
            Deflater zipper = new Deflater();
            zipper.setInput(inflatedImage);
            byte [] temp = new byte[inflatedImage.length];
            int size = zipper.deflate(temp);
            byte [] temp2 = new byte[size];
            System.arraycopy(temp, 0, temp2, 0, size);
            setImage(temp2);
        }
    }
}
```

Just before the container synchronizes the state of entity bean with the database, it calls the `ejbStore()` method. This method uses the `java.util.zip` package to compress the image file, if it has been modified, before writing it to the database.

Just after the container updates the fields of the `HypotheticalBean` with fresh data from the database, it calls the `ejbLoad()` method, which re-initializes the `inflatedImage` instance variable to `null`. Decompression is preformed lazily so it's only done when it is needed. Compression is performed by the `ejbStore()` method only if the image was accessed, otherwise the image field is not modified.

Using ejbLoad() and ejbStore() in bean-managed persistence

In bean-managed persistence the `ejbLoad()` and `ejbStore()` methods are called by the container when it's time to read or write the database. The `ejbLoad()` method will be invoked after the start of a transaction, but before the entity bean can service a method call. The `ejbStore()` is usually called after the business method is called, but it must be called before the end of the transaction.

While the entity bean is responsible for reading and writing its state to the database, the container is responsible for managing the scope of the transaction. This means that the entity bean developer need not worry about committing operations on database access APIs, provided the resource is managed by the container. The container will take care of committing the transaction and making persistent the changes at the appropriate times.

If a bean-managed persistence entity bean uses a resource that is not managed by the container system, the entity bean must manage the scope of the transaction manually, using operations specific to the API. Examples of how to use the `ejbLoad()` and `ejbStore()` methods in bean-managed persistence are shown in detail in Chapter 10.

ejbPassivate() and ejbActivate()

The `ejbPassivate()` method notifies the bean developer that the entity bean instance is about to be pooled or otherwise disassociated from the entity bean identity. This gives the entity bean developer an opportunity to do some last minute clean up before the bean is placed in the pool—where it will be reused by some other EJB object.

The `ejbActivate()` method notifies the bean developer that the entity bean instance has just returned from the pool and is now associated with an EJB object and has been assigned an identity. This gives the entity bean developer an opportunity to prepare the entity bean for service, which might include obtaining some kind of resource connection.

However, as with the `ejbPassivate()` method, it's difficult to see why this method would be used in practice. It is best to secure resources lazily (i.e., as needed). The `ejbActivate()` method suggests that some kind of eager preparation can be accomplished, but this is rarely used in practice.

Even in EJB containers that do not pool entity bean instances, the value of `ejbActivate()` and `ejbPassivate()` is questionable. It's possible that an EJB container may choose to evict instances from memory between client invocations, and create a new instance for each new transaction. While this may appear to hurt performance, it's a reasonable design provided the container system's Java virtual machine has an extremely efficient garbage collection and memory allocation strategy. Hotspot is an example of a VM that has made some important advances in this area. Even in this case, `ejbActivate()` and `ejbPassivate()` provide little value because the `setEntityContext()` and `unsetEntityContext()` can accomplish the same thing.

One of the few practical reasons for using `ejbActivate()` is to re-initialize non-persistent instance fields of the bean class that may have become dirty while the instance serviced another client.

Regardless of their general usefulness, these callback methods are at your disposal if you need them. In most cases, you are better off using `setEntityContext()` and `unsetEntityContext()` for the same purpose, since these methods will only execute once in the life cycle of a bean instance.

ejbRemove()

The component interfaces (remote, local, home, and local home) define `remove()` methods that can be used to delete an entity from the system. When a client invokes one of the `remove()` methods, as shown in the following code, the container must delete the entity's data from the database.

```
CustomerHomeRemote customerHome;
CustomerRemote customer;

customer.remove( )
// or
customerHome.remove(customer);
```

The data deleted from the database includes all CMP fields and, in the case of CMP 2.0, the CMR fields. So, for example, when invoking remove on a Ship EJB, the corresponding record in the `SHIP` table is deleted.

In CMP 2.0, the remove method also removes the link between the SHIP record and the CRUISE record. However, the CRUISE record associated with the SHIP record will not be automatically deleted. The address data will be deleted along with the customer data only if *cascading delete* is specified. A cascading delete must be declared explicitly in the XML deployment descriptor, as explained in Chapter 7.

The ejbRemove() method in container-managed persistence notifies the entity bean that it's about to be removed, and its data deleted. This notification occurs after the client invokes one of the remove() methods defined in a component interface, but before the container actually deletes the data. It gives the bean developer an opportunity to do some last minute clean up before the entity is removed. Any clean-up operations that might ordinarily be done in the ejbPassivate() method should also be done in the ejbRemove() method, because the bean will be pooled after the ejbRemove() method without having its ejbPassivate() method invoked.

In bean-managed persistence, the bean developer is responsible for implementing the logic that removes the entity bean's data from the database.

EJB 2.0: ejbHome

In EJB 2.0, CMP and BMP entity beans can declare *home methods* that perform operations related to the EJB component, but that are not specific to an entity bean instance. A home method must have a matching implementation in the bean class with the signature ejbHome<METHOD-NAME>().

For example, the Cruise EJB might define a home method that calculates the total revenue in bookings for a specific Cruise.

```
public interface CruiseHomeLocal extends javax.ejb.EJBLocalHome {

    public CruiseLocal create(String name, ShipLocal ship);
    public void setName(String name);
    public String getName( );
    public void setShip(ShipLocal ship);
    public ShipLocal getShip( );

    public double totalReservationRevenue(CruiseLocal cruise);
}
```

Every method in the home interfaces must have a corresponding ejbHome<METHOD-NAME>() in the bean class. For example, the CruiseBean class would have an ejbHomeTotalReservationRevenue() method, as shown in the following code.


```
public abstract class CruiseBean
implements javax.ejb.EntityBean {
    public Integer ejbCreate(String name,
                        ShipLocal ship) {
        setName(name);
    }
    ...
    public double ejbHomeTotalReservationRevenue(CruiseLocal cruise){

        Set reservations = ejbSelectReservations(cruise);
        Iterator enum = set.iterator();
        double total = 0;
        while(enum.hasNext()){
            ReservationLocal res = (ReservationLocal)enum.next();
            Total += res.getAmount();
        }
        return total;

    }

   public abstract ejbSelectReservations(CruiseLocal cruise);
    …
}
```

Like the `ejbFind` methods in bean-managed persistence, the `ejbHome` methods execute without an identity within the instance pool. This is why the `ejbHomeTotalReservationRevenue()` required that a `CruiseLocal` EJB object reference be passed in to the method. This makes sense once you realize that the caller is invoking the home method on the entity bean's EJB home object, and not an entity bean reference directly. The EJB home (local or remote) is not specific to any one entity instance.

The bean developer may implement home methods in both EJB 2.0 bean-managed persistence and container-managed persistence. Container-managed persistence implementations typically rely on select methods, while BMP implementations frequently use direct database access and the finder methods of beans to query data and apply changes.

EntityContext

The first method called by the container after a bean instance is created is `setEntityContext()`. This method passes the bean instance a reference to its `javax.ejb.EntityContext`, which is really the instance's interface to the container.

The `setEntityContext()` method is called prior to the bean instance's entry into the instance pool. In Chapter 3, we discussed the instance pool that

EJB containers maintain, where instances of entity and stateless session beans are kept ready to use. `EntityBean` instances in the instance pool are not associated with any data in the database; their state is not unique. When a request for a specific entity is made by a client, an instance from the pool is chosen, populated with data from the database, and assigned to service the client.

At the end of the entity bean instance's life, after it is removed permanently from the instance pool and before it is garbage collected, the `unsetEntityContext()` method is called, indicating that the bean instance's `EntityContext` is no longer implemented by the container.

The `setEntityContext()` method should be implemented by the entity bean developer so that it places the `EntityContext` reference in an instance field of the bean where it will be kept for the life of the instance. The definition of `EntityContext` in EJB 2.0 is as follows:

```
public interface javax.ejb.EntityContext extends javax.ejb.EJBContext {
    public EJBLocalObject getEJBLocalObject()
        throws IllegalStateException
    public abstract EJBObject getEJBObject()
        throws IllegalStateException;
    public abstract Object getPrimaryKey() throws IllegalStateException;
}
```

`EJBLocalObject` is new to EJB 2.0 and is not supported by EJB 1.1. EJB 1.1 uses the Enterprise JavaBeans 1.1 `EntityContext`, which doesn't define a `getEJBLocalObject()` method.

The definition of the `EntityContext` in EJB 1.1 is as follows:

```
public interface javax.ejb.EntityContext extends javax.ejb.EJBContext {
    public abstract EJBObject getEJBObject()
        throws IllegalStateException;
    public abstract Object getPrimaryKey() throws IllegalStateException;
}
```

As the bean instance is swapped from one EJB object to the next, the information obtainable from the `EntityContext` reference changes to reflect the EJB object that the instance is assigned to. This is possible because the `EntityContext` is an interface, not a static class definition. This means that the container can implement the `EntityContext` with a concrete class that it controls. As the entity bean instance is swapped from one EJB object to another, some of the information made available through the `EntityContext` will also change.

The `getEJBObject()` method returns a remote reference to the bean instance's EJB object. The `getEJBLocalObject()` method (EJB 2.0), on the other hand, returns a local reference to the bean instance's EJB object.

The EJB objects obtained from the `EntityContext` are the same kinds of references that might be used by an application client, in the case of the remote reference, or another co-located bean, in the case of a local reference. The purpose of this method is to provide the bean instance with a reference to itself when it needs to perform a loopback operation, or to provide a reference to another bean for a relationship field.

A loopback occurs when a bean invokes a method on another bean, passing itself as one of the parameters. Here is an example:

```
public class A_Bean extends EntityBean {
   public EntityContext context;
   public void someMethod() {
       B_Bean  b = ... // Get a remote reference to B_Bean.
       EJBObject obj = context.getEJBObject();
       A_Bean mySelf =  (A_Bean)
           PortableRemoteObject.narrow(obj,A_Bean.class);
       b.aMethod( mySelf );
   }
   ...
}
```

It is illegal for a bean instance to pass a `this` reference to another bean; instead, it passes its remote or local EJB object reference, which the bean instance gets from its `EntityContext`. As discussed in Chapter 3, loopbacks or reentrant behavior are problematic in EJB and should be avoided by new EJB developers.

> Session beans also define the `getEJBObject()` and `getEJBLocalObject()` method (EJB 2.0) in the `SessionContext` interface; its behavior is exactly the same.

In EJB 2.0, the ability to obtain an EJB object reference to itself is also useful when establishing relationships with other beans in container-managed persistence. For example, the Customer EJB might implement a business method that allows it to assign itself a Reservation.

```
public abstract class CustomerBean implements javax.ejb.EntityBean {
    public EntityContext context;

    public void assignToReservation(ReservationLocal reservation){
        EJBLocalObject localRef = context.getEJBLocalObject();
        Collection customers = reservation.getCustomers();
        customers.add(localRef);
    }
    …
}
```

The `getPrimaryKey()` method allows a bean instance to get a copy of the primary key to which it is currently assigned. Using this method outside of the `ejbLoad()` and `ejbStore()` methods of BMP entity beans is probably rare,

but the EntityContext makes the primary key available for those unusual circumstances when it is needed.

As the context in which the bean instance operates changes, some of the information made available through the EntityContext reference will be changed by the container. This is why the methods in the EntityContext throw the java.lang.IllegalStateException. The EntityContext is always available to the bean instance, but the instance is not always assigned to an EJB object. When the bean is between EJB objects, when it's in the pool, it has no EJB object or primary key to return. If the getEJBObject(), getEJBLocalObject(), or getPrimaryKey() methods are invoked when the bean is not assigned to an EJB object (when it's in the pool), these methods will throw an IllegalStateException. Appendix B provides tables for each bean type describing which EJBContext methods can be invoked at what times.

EJBContext

The EntityContext extends the javax.ejb.EJBContext class, which is also the base class for the SessionContext used by session beans. The EJBContext defines several methods that provide useful information to a bean at runtime. Here is the definition of the EJBContext interface:

```
package javax.ejb;
public interface EJBContext {

   // EJB home methods
   public EJBHome getEJBHome();
   // EJB 2.0 only
   public EJBLocalHome getEJBLocalHome();

   // security methods
   public java.security.Principal getCallerPrincipal();
   public boolean isCallerInRole(java.lang.String roleName);

   // transaction methods
   public javax.transaction.UserTransaction getUserTransaction()
   throws java.lang.IllegalStateException;
   public boolean getRollbackOnly()
       throws java.lang.IllegalStateException;
   public void setRollbackOnly()
       throws java.lang.IllegalStateException;

   // deprecated methods
   public java.security.Identity getCallerIdentity();
   public boolean isCallerInRole(java.security.Identity role);
   public java.util.Properties getEnvironment();
```



```
}
```

The `getEJBHome()` and `getEJBLocalHome()` methods (EJB 2.0) returns a reference to the bean's EJB home. This is useful if the bean needs to create or find entity beans of its own type. Access to the EJB home proves more useful in bean-managed entity beans or CMP 1.1 entity beans then it does in CMP 2.0 entity beans, which have select methods and CMR fields.

As an example, if all employees in Titan's system (including managers) are represented by CMP 1.1 Employee beans, then a manager employee who needs access to subordinate employees can use the `getEJBHome()` method to get beans representing the appropriate employees:

```
public class EmployeeBean implements EntityBean {
     int id;
     String firstName;
     ...
     public Enumeration getSubordinates() {
         Object ref = ejbContext.getEJBHome();
         EmployeeHome home = (EmployeeHome)
             PortableRemoteObject.narrow(ref, EmployeeHome.class);
         Integer primKey = (Integer)context.getPrimaryKey();
         Enumeration subordinates = home.findByManagerID(primKey);
         return subordinates;
     }
     ...
}
```

The `getCallerPrincipal()` method is used to obtain the `Principal` object representing the client that is currently accessing the bean. The `Principal` object can, for example, be used by the Ship bean to track the identity of clients making updates:

```
 public class ShipBean implements EntityBean {
    String modifiedBy;
    EntityContext context;
    ...
    public void setTonnage(double tons){
        tonnage = tons;
        Principal principal = context.getCallerPrincipal();
        String modifiedBy = principal.getName();
    }
    ...
}
```

The `isCallerInRole()` method tells you whether the client accessing the bean is a member of a specific role, identified by a role name. This method is useful when more access control is needed than simple method-based access control can provide. In a banking system, for example, the Teller role might be allowed to make withdrawals, but only a Manager can make withdrawals over $10,000. This kind of fine-grained access control cannot be addressed through

EJB's security attributes because it involves a business logic problem. Therefore, we can use the `isCallerInRole()` method to augment the automatic access control provided by EJB. First, let's assume that all Managers also are Tellers. Let's also assume that the deployment descriptor for the Account bean specifies that clients that are members of the Teller role can invoke the `withdraw()` method. The business logic in the `withdraw()` method uses `isCallerInRole()` to further refine access control so that only the Manager role can withdraw over $10,000.00.

```
public class AccountBean implements EntityBean {
    int id;
    double balance;
    EntityContext context;

    public void withdraw(Double withdraw)
    throws AccessDeniedException {

        if (withdraw.doubleValue() > 10000) {
           boolean isManager = context.isCallerInRole("Manager");
           if (!isManager) {
              // Only Managers can withdraw more than 10k.
              throw new AccessDeniedException();
           }
        }
        balance = balance - withdraw.doubleValue();

    }
    ...
}
```

The `EJBContext` contains some deprecated methods that were used in EJB 1.0 but were deprecated in EJB 1.1 and have been abandoned in EJB 2.0. Support for these deprecated methods is optional for EJB 1.1 containers, which can host EJB 1.0 beans. EJB containers that do not support the deprecated security methods will throw a `RuntimeException`. The deprecated security methods are based on EJB 1.0's use of the `Identity` object instead of the `Principal` object. The semantics of the deprecated methods are basically the same, but because `Identity` is an abstract class, it has proven to be too difficult to use.

The `getEnvironment()` method has been replaced by the JNDI Environment Naming Context, which is discussed later in the book. Support in EJB 1.1 for the deprecated `getEnvironment()` method is discussed in detail in Chapter 12.

The transactional methods (`getUserTransaction()`, `setRollbackOnly()`, `getRollbackOnly()`) are described in detail in Chapter 14.

The material on the `EJBContext` as covered in this section apply equally well to session and message-driven beans. There are some exceptions, however, and

these differences are covered in Chapter 12, *Session beans* and Chapter 13, *Message-Driven beans.*

JNDI ENC

Starting with EJB 1.1, the bean-container contract for entity and stateful beans was expanded beyond the `EJBContext` using the Java Naming and Directory Interface (JNDI). A special JNDI name space, which is referred to as the *environment naming context* (ENC), was added which allows any enterprise bean to access environment entries, other beans, and resources such as JDBC `DataSource` objects specific to that enterprise bean.

The JNDI ENC continues to be an extremely important part of the bean-container contract in EJB 2.0. Although the JNDI ENC is used to access JDBC in the bean-managed persistence chapter (Chapter 10), it's not specific to entity beans. The JNDI ENC is used by session, entity, and message-driven beans alike. To avoid unnecessary duplication, a detailed discussion of this important facility is left for Chapter 12, *Session beans.* What you learn about using the JNDI ENC in Chapter 12 applies equally as well to session, entity, and message-driven beans.

The Life Cycle of an Entity Bean

To understand how to best develop entity beans, it is important to understand how the container manages them. The EJB specification defines just about every major event in an entity bean's life, from the time it is instantiated to the time it is garbage collected. This is called the *life cycle*, and it provides the bean developer and EJB vendors with all the information they need to develop beans and EJB servers that adhere to a consistent protocol. To understand the life cycle, we will follow an entity instance through several life-cycle events and describe how the container interacts with the entity bean during these events. Figure 11-2 illustrates the life cycle of an entity instance.

[FIGURE]

Figure 11-2: Entity bean life cycle

We will examine the life cycle of an entity bean and identify the points at which the container would call each of the methods described in the `EntityBean` interface, as well as the find methods, and in EJB 2.0, the select and home methods. Bean instances must implement the `EntityBean` interface, which means that invocations of the callback methods are invocations on the bean instance itself.

At each stage of the entity bean's life cycle the bean container provides varying levels of access. For example, the `EntityContext.getPrimary()` method will not work if it's invoked during in the `ejbCreate()` method, but it does

work when called in the `ejbPostCreate()` method. Other `EJBContext` methods have similar restrictions, as does the JNDI ENC. While this section touches on the accessibility of these methods, a complete table that details what is available in each bean class method (`ejbCreate()`, `ejbActivate()`, `ejbLoad()`, etc.) can be found in Appendix B.

Does Not Exist

The entity bean begins life as a collection of files. Included in that collection are the bean's deployment descriptor, component interfaces and all the supporting classes generated at deployment time. At this stage, no instance of the bean exists.

The Pooled State

When the EJB server is started, it reads the EJB's files and instantiated several instances of the entity bean's bean class, which it places in a pool. The instances are created by calling the `Class.newInstance()` method on the bean class. The `newInstance()` method creates an instance using the default constructor, which has no arguments.[1] This means that the persistent fields of the bean instances are set at their default values; the instances themselves do not represent any data in the database.

Immediately following the creation of an instance, and just before it is placed in the pool, the container assigns the instance its `EntityContext`. The `EntityContext` is assigned by calling the `setEntityContext()` method defined of the `EntityBean` interface which is implemented by the bean class. After the instance has been assigned its context, it is entered into the instance pool.

In the instance pool, the bean instance is available to the container as a candidate for servicing client requests. Until it is requested, however, the bean instance remains inactive unless it is used to service a query methods (finder or select methods) or ejbHome requests. Bean instances in the Pooled state typically service query and ejbHome requests, which makes perfectly good sense because they aren't busy, and these methods don't rely on the bean instance's state. All instances in the Pooled state are equivalent. None of the instances are assigned to an EJB object, and none of them has meaningful state.

---

1 Constructors should never be defined in the bean class. The default no-argument constructor must be available to the container.

The Ready State

When a bean instance is in the Ready State, it can accept client requests. A bean instance moves to the Ready State when the container assigns it to an EJB object. This occurs under two circumstances: when a new entity bean is being created or when the container is activating an entity.

Transitioning from the Pooled state to the Ready State via creation

When a client application invokes the `create()` method on an EJB home, several operations must take place before the EJB container can return a remote or local reference (EJB object) to the client. First, an EJB object must be created on the EJB server [2]. Once the EJB object is created, a entity bean instance is taken from the instance pool and assigned to the EJB object. Next, the `create()` method, invoked by the client, is delegated to its corresponding `ejbCreate()` method on the bean instance. After the `ejbCreate()` method completes, a primary key is created. In container-managed persistence, the container instantiates and populates the key automatically; in bean-managed persistence the entity bean constructs the primary key manually in the `ejbCreate()` method. Once the primary key is created, the key is embedded in the EJB object, providing it with *identity*. Once the EJB object has identity, the bean instance's `EntityContext` can access information specific to that EJB object, including the primary key and its own remote reference. While the `ejbCreate()` method is executing, the security and transactional information is available.

When the `ejbCreate()` method is done, the `ejbPostCreate()` method on the entity bean instance is called. At this time, the bean instance can perform any post-processing that is necessary before making itself available to the client—modifying relationship fields is typical. While the `ejbPostCreate()` executes, the bean's primary key and access to its own EJB object reference are available through the `EntityContext`. In EJB 2.0, the `ejbPostCreate()` method can be used to initialize the container-managed relationship fields.

Finally, after the successful completion of the `ejbPostCreate()` method, the home is allowed to return a remote or local reference—an EJB object—to the client. The bean instance and EJB object are now ready to service method requests from the client. This is one way that the bean instance can move from the Pooled state to the Ready State.

---

2 This is only a conceptual model. In reality an EJB container and the EJB object may be the same thing or perhaps a single EJB object provides a multiplexing service for all entities of the same type. The implementation details are not as important as understanding the life cycle protocol.

Transitioning from the Pooled state to the Ready State via a query method

When a query method is executed, each EJB object that is found as a result of the query will be realized by transitioning an instance from the Pooled state to the Ready State. When an entity bean is found, it is assigned to an EJB object and its EJB object reference is returned to the client. A found bean follows the same protocol as a passivated bean; it is activated when the client invokes a business method. A found bean can be considered to be a passivated bean and will move into the Ready State through activation as described in the next section.

In many cases (depending on the EJB vendor), found entity beans don't actually migrate into the ready state until they are accessed by the client. So, for example, of a find method returns a collection of entity beans, the entity beans may not be activated until they are obtained from the collection or when accessed directly by the client. This saves resources by activating entity beans lazily (as needed).

Transitioning from the Pooled state to the Ready State via activation

The activation process can also move an entity bean instance from the Pooled state to the Ready State. Activation facilitates resource management by allowing a few bean instances to service many EJB objects. Activation was explained in Chapter 2, but we will revisit the process as it relates to the entity bean instance's life cycle. Activation presumes that the entity bean has previously been *passivated*. More is said about this state transition later; for now, suffice it to say that when a bean instance is passivated, it frees any resources that it does not need and leaves the EJB object for the instance pool. When the bean instance returns to the pool, the EJB object is left without an instance to delegate client requests to. The EJB object maintains its stub connection on the client, so as far as the client is concerned, the entity bean hasn't changed. When the client invokes a business method on the EJB object, the EJB object must obtain a new bean instance. This is accomplished by activating a bean instance.

When a bean instance is activated, it leaves the instance pool (the Pooled State) to be assigned to an EJB object. Once assigned to the proper EJB object, the `ejbActivate()` method is called—the instance's `EntityContext` can now provide information specific to the EJB object, but it cannot provide security or transactional information. The `ejbActivate()` callback method can be used in the bean instance to re-obtain any resources or perform some other work needed before servicing the client.

When an entity bean instance is activated, non-persistent instance fields of the bean instance may contain arbitrary values (dirty values) and must be reinitialized in the `ejbActivate()` method.

In container-managed persistence, container-managed fields are automatically synchronized with the database after `ejbActivate()` is invoked and before a

business method can be serviced by the bean instance. The order in which these things happen in CMP entity beans is:

1. `ejbActivate()` is invoked on the bean instance.
2. Persistent fields are synchronized automatically.
3. `ejbLoad()` notifies the bean that its persistent fields have been synchronized.
4. Business methods are invoked as needed.

In bean-managed persistence, persistent fields are synchronized by the `ejbLoad()` method after `ejbActivate()` has been called and before a business method can be invoked. Here is the order of operations in bean-managed persistence:

1. `ejbActivate()` is invoked on the bean instance.
2. `ejbLoad()` is called to let the bean synchronize its persistent fields.
3. Business methods are invoked as needed.

Transitioning from the Ready State to the Pooled state via passivation

A bean can move from the Ready State to the Pooled state via passivation, which is the process of disassociating a bean instance from an EJB object when it is not busy. After a bean instance has been assigned to an EJB object, the EJB container can passivate the instance at any time, provided that the instance is not currently executing a method. As part of the passivation process, the `ejbPassivate()` method is invoked on the bean instance. This callback method can be used by the instance to release any resources or perform other processing prior to leaving the EJB object. When `ejbPassivate()` has completed, the bean instance is disassociated from the EJB object server and returned to the instance pool. The bean instance is now back in the Pooled State.

A bean-managed entity instance should not try to save its state to the database in the `ejbPassivate()` method; this activity is reserved for the `ejbStore()` method. The container will invoke `ejbStore()` to synchronize the bean instance's state with the database prior to passivating the bean.

The most fundamental thing to remember is that, for entity beans, passivation is simply a notification that the instance is about to be disassociated from the EJB object. Unlike stateful session beans, an entity bean instance's fields are not serialized and held with the EJB object when the bean is passivated. Whatever values are held in the instance's non-persistent fields when it was assigned to the EJB object will be carried with it to its next assignment.

Transitioning from the Ready State to the Pooled state via removal

A bean instance also moves from the Ready State to the Pooled state when it is removed. This occurs when the client application invokes one of the `remove()` methods on the bean's EJB object or EJB home. With entity beans, invoking a remove method means that the entity's data is deleted from the database. Once the entity's data has been deleted from the database, it is no longer a valid entity. The `EntityContext` can provide the EJB object with identity information during the execution of the `ejbRemove()` method. Once the `ejbRemove()` method has finished, the bean instance is moved back to the instance pool and out of the Ready State. It is important that the `ejbRemove()` method release any resources that would normally be released by `ejbPassivate()`, which is not called when a bean is removed. This can be done, if need be, by invoking the `ejbPassivate()` method within the `ejbRemove()` method body.

In bean-managed persistence, the `ejbRemove()` method is implemented by the entity bean developer and include code to delete the entity bean's data from the database. The EJB container will invoke the `ejbRemove()` method in response to a client's invocation of the `remove()` method on one of the component interfaces.

In container-managed persistence, the `ejbRemove()` method notifies the entity bean instance that its data is about to be removed form the database. Immediately following the `ejbRemove()` call, the container detetes the entity bean's data.

In EJB 2.0 CMP the container also cleans up the entity bean's relationships with other entity beans in the database. If a cascade delete is specified, it removes each entity bean in the cascade delete relationships. This involves activating each entity bean and calling its `ejbActivate()` methods, loading each entity bean's state by calling its `ejbLoad()` method, calling the `ejbRemove()` on all of the entity beans in the cascade relationship, and then deleting their data. This process can continue in a chain until all the cascade-delete operations of all the relationships have completed.

Life in the Ready State

A bean is in the Ready State when it is associated with an EJB object and is ready to service requests from the client. When the client invokes a business method, like `Ship.getName()`, on the bean's remote or local reference (EJB object), the method invocation is received by the EJB server and delegated to the bean instance. The instance performs the method and returns the results. As long as the bean instance is in the Ready State, it can service all the business methods invoked by the client. Business methods can be called zero or more times in any order.

In addition to servicing business methods, an entity bean in the ready state can also execute select methods, which are called by the bean instance on itself while servicing a business method or ejbHome method.

The `ejbLoad()` and `ejbStore()` methods, which synchronize the bean instance's state with the database, can be called only when the bean is in the Ready State. These methods can be called in any order, depending on the vendor's implementation. Some vendors call `ejbLoad()` before every method invocation and `ejbStore()` after every method invocation, depending on the transactional context. Other vendors call these methods less frequently.

In bean-managed persistence, the `ejbLoad()` method should always use the `EntityContext.getPrimaryKey()` to obtain data from the database and not trust any primary key or other data that the bean has stored in one of its fields. (This is how we implemented it in the bean-managed version of the Ship bean in Chapter 10.) It should be assumed, however, that the state of the bean is valid when calling the `ejbStore()` method.

In container-managed persistence, the `ejbLoad()` method is always called immediately following the synchronization of the bean's container-managed fields with the database—in other words, right after the container updates the state of the bean instance with data from the database. This provides an opportunity to perform any calculations or reformat data before the instance can service business method invocations from the client. The `ejbStore()` method is called just before the database is synchronized with the state of the bean instance—just before the container writes the container-managed fields to the database. This provides the CMP entity bean instance with an opportunity to change the data in the container-managed fields prior to their persistence to the database.

In bean-managed persistence, the `ejbLoad()` and `ejbStore()` methods are called when the container deems it appropriate to synchronize the bean's state with the database. These are the only callback methods that should be used to synchronize the bean's state with the database. Do not use `ejbActivate()`, `ejbPassivate()`, `setEntityContext()`, or `unsetEntityContext()` to access the database for the purpose of synchronization. The `ejbCreate()` and `ejbRemove()` methods, however, can be used to insert and delete (respectively) the entity's data from the database.

End of the Life Cycle

A bean instance's life cycle ends when the container decides to remove it from the pool and allow it to be garbage collected. This happens under a few different circumstances. If the container decides to reduce the number of instances in the pool—usually to conserve resources—it releases one or more bean instances

and allows them to be garbage collected. The ability to adjust the size of the instance pool allows the EJB server to manage its resources (the number of threads, available memory, etc.) so that it can achieve the highest possible performance. This behavior is typical of a CTM.

When an EJB server is shut down, most containers release all the bean instances so that they can be safely garbage collected. Finally, some containers may decide to release an instance that is behaving unfavorably or an instance that has suffered from some kind of unrecoverable error that makes it unstable. For example, anytime an entity bean instance throws a type of `RuntimeException` from any of its methods, the EJB container will evict that instance from memory and replace it with a stable instance from the instance pool.

When an entity bean instance leaves the instance pool to be garbage collected, the `unsetEntityContext()` method is invoked by the container to alert the bean instance that it is about be destroyed. This callback method lets the bean instance release any resources it maintains before being garbage collected. Once the bean's `unsetEntityContext()` method has been called it will be garbage collected.

The bean instance's `finalize()` method may or may not be invoked following the `unsetEntityContext()` method. A bean should not rely on its `finalize()` method, since each vendor handles evicting instances differently.

12

Session Beans

Chapters 6 through 11 demonstrated that entity beans provide an object-oriented interface that makes it easier for developers to create, modify, and delete data from the database. Entity beans make developers more productive by encouraging reuse and reducing development costs. A concept like a Ship can be reused throughout a business system without having to redefine, recode, or retest the business logic and data access.

However, entity beans are not the entire story. We have also seen another kind of enterprise bean: the session bean. Session beans fill the gaps left by entity beans. They are useful for describing interactions between other beans (workflow) or for implementing particular tasks. Unlike entity beans, session beans don't represent shared data in the database, but they can access shared data. This means that we can use session beans to read, update, and insert data. For example, we might use a session bean to provide lists of information, such as a list of all available cabins. Sometimes we might generate the list by interacting with entity beans, like the cabin list we developed in the TravelAgent EJB in Chapter 4. More frequently, session beans will generate lists by accessing the database directly.

So when do you use an entity bean and when do you use a session bean to directly access data? Good question! As a rule of thumb, an entity bean is developed to provide a safe and consistent interface to a set of shared data that defines a concept. This data may be updated frequently. Session beans access data that spans concepts, is not shared, or is usually read-only.

In addition to accessing data directly, session beans can represent *workflow*. Workflow describes all the steps required to accomplish a particular task, such as booking passage on a ship or renting a video. Session beans are part of the same

business API as entity beans, but as workflow components, they serve a different purpose. Session beans can manage the interactions between entity beans, describing how they work together to accomplish a specific task. The relationship between session beans and entity beans is like the relationship between a script for a play and the actors that perform the play. Where entity beans are the actors, the session bean is the script. Actors without a script can each serve a function individually, but only in the context of a script can they tell a story. In terms of our example, it makes no sense to have a database full of cabins, ships, customers, and other objects if we can't create interactions between them, like booking a customer for a cruise.

Session beans are divided into two basic types: *stateless* and *stateful*. A *stateless* session bean is a collection of related services, each represented by a method; the bean maintains no state from one method invocation to the next. When you invoke a method on a stateless session bean, it executes the method and returns the result without knowing or caring what other requests have gone before or might follow. Think of a stateless session bean as a set of procedures or batch programs that execute a request based on some parameters and return a result. Stateless session beans tend to be general-purpose or reusable, such as a software service.

A *stateful* session bean is an extension of the client application. It performs tasks on behalf of the client and maintains state related to that client. This state is called *conversational state* because it represents a continuing conversation between the stateful session bean and the client. Methods invoked on a stateful session bean can write and read data to and from this conversational state, which is shared among all methods in the bean. Stateful session beans tend to be specific to one scenario. They represent logic that might have been captured in the client application of a two-tier system. Session beans, whether they are stateful or stateless, are not persistent like entity beans. In other words, session beans are not saved to the database.

Depending on the vendor, stateful session beans may have a timeout period. If the client fails to use the stateful bean before it times out, the bean instance is destroyed and the EJB object reference is invalidated. This prevents the stateful session bean from lingering long after a client has shut down or otherwise finished using it. A client can also explicitly remove a stateful session bean by calling one of its remove methods.

Stateless session beans have longer lives because they don't retain any conversational state and are not dedicated to one client, but they still aren't saved in a database because they don't represent any data. Once a stateless session bean has finished a method invocation for a client, it can be reassigned to any other EJB object to service a new client. A client can maintain a connection to a stateless session bean's EJB object, but the bean instance itself is free to service requests from any client. Because it doesn't contain any state information, there's no difference between one client and the next. Stateless session beans may also have a timeout period and can be removed by the client,

but the impact of these events is different than with a stateful session bean. With a stateless session bean, a timeout or remove operation simply invalidates the EJB object reference for that client; the bean instance is not destroyed and is free to service other client requests.

The Stateless Session Bean

A stateless session bean is very efficient and relatively easy to develop. Stateless session beans require few server resources because they are neither persistent nor dedicated to one client. Because they aren't dedicated to one client, many EJB objects can share a few instances of a stateless bean. A stateless session bean does not maintain conversational state relative to the client it is servicing, so it can be swapped freely between EJB objects. As soon as a stateless instance services a method invocation, it can be swapped to another EJB object immediately. Because there is no conversational state, a stateless session bean doesn't require passivation or activation, further reducing the overhead of swapping. In short, they are lightweight and fast!

Stateless session beans often perform services that are fairly generic and reusable. The services may be related, but they are not interdependent. This means that everything a method needs to know has to be passed via the method's parameters. This provides an interesting limitation. Stateless session beans can't remember anything from one method invocation to the next, which means that they have to take care of the entire task in one method invocation. The only exception to this rule is information obtainable from the `SessionContext` and the JNDI ENC.

Stateless session beans are EJB's version of the traditional transaction processing applications, which are executed using a procedure call. The procedure executes from beginning to end and then returns the result. Once the procedure is done, nothing about the data that was manipulated or the details of the request are remembered. There is no state.

These restrictions don't mean that a stateless session bean can't have instance variables or maintain some kind of internal state. There's nothing that prevents you from keeping a variable that tracks the number of times a bean has been called or that saves data for debugging. An instance variable can even hold a reference to a live resource like a URL connection for writing debugging data, verifying credit cards, or anything else that might be useful. However, it's important to remember that this state can never be visible to a client. A client can't assume that the same bean instance will service it every time. If these instance variables have different values in different bean instances, their values will appear to change randomly as stateless session beans are swapped from one client to another. Therefore, any resources that you reference in instance variables should be generic. For example, each bean instance might reasonably record debugging messages in a different file—that might be the only way to figure out what was happening on a large server with many bean instances. The

client doesn't know or care where debugging output is going. However, it would be clearly inappropriate for a stateless bean to remember that it was in the process of making a reservation for Madame X—the next time it is called, it may be servicing another client entirely.

Stateless session beans can be used for report generation, batch processing, or some stateless services like validating a credit card. Another good application would be a StockQuote EJB that returns a stock's current price. Any activity that can be accomplished in one method call is a good candidate for the high-performance stateless session bean.

EJB 1.1: Downloading the Missing Pieces

Both the TravelAgent EJB and the ProcessPayment EJB, which we develop in this chapter, depend on other entity beans, some of which we developed earlier in this book and several that you can download from O'Reilly's web site. The Cabin was developed in Chapter 4, but we still need several other beans to develop this example. The other beans are the Cruise, Customer, and Reservation EJBs. The source code for these beans is available with the rest of the examples for this book at the O'Reilly download site. Instructions for downloading code are available in the preface of this book and in the workbook.

Before you can use these beans, you will need to create some new tables in your database. Here are the table definitions that the new entity beans will need. The Cruise EJB maps to the `CRUISE` table:

```
CREATE TABLE CRUISE
(
   ID          INT PRIMARY KEY,
   NAME        CHAR(30),
   SHIP_ID     INT
)
```

The Customer EJB maps to the `CUSTOMER` table:

```
CREATE TABLE CUSTOMER
(
   ID             INT PRIMARY KEY,
   FIRST_NAME     CHAR(30),
   LAST_NAME      CHAR(30),
   MIDDLE_NAME    CHAR(30)
)
```

The Reservation EJB maps to the `RESERVATION` table:

```
CREATE TABLE RESERVATION
(
   CUSTOMER_ID    INT,
   CABIN_ID       INT,
   CRUISE_ID      INT,
```



```
    AMOUNT_PAID DECIMAL (8,2),
    DATE_RESERVED DATE
)
```

Once you have created the tables, deploy these beans as container-managed entities in your EJB server and test them to ensure that they are working properly.

The ProcessPayment EJB

Chapters 2 and 3 discussed the TravelAgent EJB, which had a business method called `bookPassage()` that uses the ProcessPayment EJB. The next section develops a complete definition of the TravelAgent EJB, including the logic of the `bookPassage()` method. At this point, however, we are interested in the ProcessPayment EJB, which is a stateless bean used by the TravelAgent EJB. The TravelAgent EJB uses the ProcessPayment EJB to charge the customer for the price of the cruise.

The process of charging customers is a common activity in Titan's business systems. Not only does the reservation system need to charge customers, but so do Titan's gift shops, boutiques, and other related businesses. The process of charging a customer for services is common to many systems, so it has been encapsulated in its own bean.

Payments are recorded in a special database table called `PAYMENT`. The `PAYMENT` data is batch processed for accounting purposes and is not normally used outside of accounting. In other words, the data is only inserted by Titan's system; it's not read, updated, or deleted. Because the process of making a charge can be completed in one method, and because the data is not updated frequently or shared, a stateless session bean has been chosen for processing payments. Several different forms of payment can be used: credit card, check, or cash. We will model these payment forms in our stateless ProcessPayment EJB.

PAYMENT: The database table

The ProcessPayment EJB accesses an existing table in Titan's system called the `PAYMENT` table. Create a table in your database called `PAYMENT` with this definition:

```
CREATE TABLE PAYMENT
(
    customer_id     NUMERIC,
    amount          DECIMAL(8,2),
    type            CHAR(10),
    check_bar_code  CHAR(50),
    check_number    INTEGER,
    credit_number   NUMERIC,
    credit_exp_date DATE
```



```
)
```

ProcessPaymentRemote: The remote interface

A stateless session bean, like any other bean, needs a component interface. While EJB 1.1 uses only remote interfaces, in EJB 2.0 a session beans may have either a local or remote interface. For EJB 2.0 we'll develop both.

For the remote interface, we obviously need a `byCredit()` method because the TravelAgent EJB uses it. We can also identify two other methods that we'll need: `byCash()` for customers paying cash and `byCheck()` for customers paying with a personal check.

Here is a complete definition of the remote interface for the ProcessPayment EJB:

```
package com.titan.processpayment;

import java.rmi.RemoteException;
import java.util.Date;
import com.titan.customer.Customer;

public interface ProcessPaymentRemote extends javax.ejb.EJBObject {

    public boolean byCheck(CustomerRemote customer, CheckDO check,
double amount)
      throws RemoteException,PaymentException;

    public boolean byCash(CustomerRemote customer, double amount)
      throws RemoteException,PaymentException;

    public boolean byCredit(CustomerRemote customer, CreditCardDO card,
double amount)
      throws RemoteException,PaymentException;
}
```

Remote interfaces in session beans follow the same rules as the entity beans. Here we have defined the three business methods, `byCheck()`, `byCash()`, and `byCredit()`, which take information relevant to the form of payment used and return a `boolean` value that indicates the success of the payment. In addition to the required `RemoteException`, these methods can throw an application-specific exception, the `PaymentException`. The `PaymentException` is thrown if any problems occur while processing the payment, such as a low check number or an expired credit card. Notice, however, that nothing about the `ProcessPaymentRemote` interface is specific to the reservation system. It could be used just about anywhere in Titan's system. In addition, each method defined in the remote interface is completely independent of the others. All the data that is required to process a payment is obtained through the method's arguments.

As an extension of the `javax.ejb.EJBObject` interface, the remote interface of a session bean inherits the same functionality as the remote interface of an entity bean. However, the `getPrimaryKey()` method throws a `RemoteException`, since session beans do not have a primary key to return:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
    public abstract EJBHome getEJBHome()
     throws RemoteException;
    public abstract Handle getHandle()
     throws RemoteException;
    public abstract Object getPrimaryKey()
     throws RemoteException;
    public abstract boolean isIdentical(EJBObject obj)
     throws RemoteException;
    public abstract void remove()
     throws RemoteException, RemoveException;
}
```

The `getHandle()` method returns a serializable handle object, just like the `getHandle()` method in the entity bean. For stateless session beans, this handle can be serialized and reused any time, as long as the stateless bean type is still available in the container that generated the handle.

> Unlike stateless session beans, stateful session beans are only available through the handle for as long as that specific bean instance is kept alive on the EJB server. If the client explicitly destroys the stateful session bean using one of the `remove()` methods, or if the bean times out, the instance is destroyed and the handle becomes invalid. As soon as the server removes a stateful session bean, its handle is no longer valid and will throw a `RemoteException` when its `getEJBObject()` is invoked.

A remote reference to the bean can be obtained from the handle by invoking its `getEJBObject()` method:

```
public interface javax.ejb.Handle {
    public abstract EJBObject getEJBObject()
      throws RemoteException;
}
```

The ProcessPayment EJB has its own package, which means it has its own directory in our development tree, *dev/com/titan/processpayment*. That's where we'll store all the code and compile class files for this bean.

Dependent Objects: The CreditCardDO and CheckDO classes

The ProcessPayment EJB's remote interface uses two classes in its definition that are particularly interesting: the `CreditCardDO` and `CheckDO` classes. The definitions for these classes are as follows:

```
/* CreditCard.java */
package com.titan.processpayment;

import java.util.Date;

public class CreditCardDO implements java.io.Serializable {
    final static public String MASTER_CARD = "MASTER_CARD";
    final static public String VISA = "VISA";
    final static public String AMERICAN_EXPRESS =
        "AMERICAN_EXPRESS";
    final static public String DISCOVER = "DISCOVER";
    final static public String DINERS_CARD = "DINERS_CARD";

    public long number;
    public Date expiration;
    public String type;

    public CreditCard(long nmbr, Date exp, String typ) {
        number = nmbr;
        expiration = exp;
        type = typ;
    }
}

/* Check.java */
package com.titan.processpayment;

public class CheckDO implements java.io.Serializable {
    String checkBarCode;
    int checkNumber;
    public Check(String barCode, int number) {
        checkBarCode = barCode;
        checkNumber = number;
    }
}
```

The `CreditCardDO` and `CheckDO` are dependent objects (DO standards for Dependent Object) a concept that was explored with the Address EJB in Chapter 6. If you examine the class definitions of the `CreditCardDO` and `CheckDO` classes, you will see that they are not enterprise beans. They are simply serializable Java classes. These classes provide a convenient mechanism for transporting and binding together related data. `CreditCardDO`, for example, binds all the credit card data together in once class, making it easier to pass the information across the network as well as making our interfaces a little cleaner.

PaymentException, An application exception

Any remote or local interface, whether it's for an entity bean or a session bean, can throw application exceptions. Application exceptions are created by the bean developer and should describe a business logic problem—in this particular case, a problem making a payment. Application exceptions should be meaningful to the client, providing an explanation of the error that is both brief and relevant.

It's important to understand what exceptions to use and when to use them. The `RemoteException` indicates subsystem-level problems and is used by the RMI facility. Likewise, exceptions like `javax.naming.NamingException` and `java.sql.SQLException` are thrown by other Java subsystems; usually these should not be thrown explicitly by your beans. The Java Compiler requires that you use `try`/`catch` blocks to capture checked exceptions like these.

In EJB 2.0, the `EJBException` can express container problems processing local interface invocations. The `EJBException` is an unchecked exception so you won't get a compile error if you don't write code to handle it. However, under certain circumstances it's a good idea to catch `EJBException`, while in other circumstances it's best to propagate it.

When a checked exception from a subsystem (JDBC, JNDI, JMS, etc.) is caught by a bean method, it should be rethrown as an `EJBException` or an application exception. You would rethrow a checked exception as an `EJBException` if it represented a system-level problem; checked exceptions are rethrown as application exceptions when they result from business logic problems. Your beans incorporate your business logic; if a problem occurs in the business logic, that problem should be represented by an application exception. When an `EJBException` or someother type of `RuntimeException` is thrown by the enterprise bean, the exception is first processed by the container, which discards the bean instance and replaces it with another. After the container processes the exception, it then propagates an exception to the client. For remote clients, the container throws a `RemoteException`; for local clients (co-located enterprise beans), the container rethrows the original `EJBException` or `RuntimeException` that was thrown by the bean instance.

The `PaymentException` describes a specific business problem, so it is an application exception. Application exceptions extend `java.lang.Exception`. If you choose to include any instance variables in the exception, they should all be serializable. Here is the definition of `ProcessPayment` application exception:

```
package com.titan.processpayment;

public class PaymentException extends java.lang.Exception {
    public PaymentException() {
```

```
        super();
    }
    public PaymentException(String msg) {
        super(msg);
    }
}
```

ProcessPaymentHomeRemote: The home interface

The home interface of a stateless session bean must declare a single `create()` method with no arguments. This is a requirement of the EJB specification. It is illegal to define `create()` methods with arguments, because stateless session beans don't maintain conversational state that needs to be initialized. There are no find methods in session beans, because session beans do not have primary keys and do not represent data in the database.

Although EJB 2.0 defines `create<SUFFIX>()` methods for stateful and entity beans, stateless session beans may only define a single `create()` method, with no suffix and no arguments. This is also the case in EJB 1.1. The reason for this restriction has to do with the life cycle of stateless session beans, which is explained later in the chapter.

Here is the definition of the remote home interface for the ProcessPayment EJB:

```
package com.titan.processpayment;

import java.rmi.RemoteException;
import javax.ejb.CreateException;

public interface ProcessPaymentHomeRemote extends javax.ejb.EJBHome {
   public ProcessPayment create()
       throws RemoteException, CreateException;
}
```

The `CreateException` is mandatory, as is the `RemoteException`. The `CreateException` can be thrown by the bean itself to indicate an application error in creating the bean. A `RemoteException` is thrown when other system errors occur, for example, when there is a problem with network communication or when an unchecked exception is thrown from the bean class.

The `ProcessPaymentHomeRemote` interface, as an extension of the `javax.ejb.EJBHome`, offers the same `EJBHome` methods as entity beans. The only difference is that `remove(Object primaryKey)` doesn't work because session beans don't have primary keys. If `EJBHome.remove(Object primaryKey)` is invoked on a session bean (stateless or stateful), a `RemoteException` is thrown. Logically, this method should never be invoked on the remote home interface of a session bean. Here are the definitions of the `javax.ejb.EJBHome` interface for EJB 1.1 and 2.0:

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {
```



```
    public abstract HomeHandle getHomeHandle()
      throws RemoteException;
    public abstract EJBMetaData getEJBMetaData()
      throws RemoteException;
    public abstract void remove(Handle handle)
      throws RemoteException, RemoveException;
    public abstract void remove(Object primaryKey)
      throws RemoteException, RemoveException;
}
```

The home interface of a session bean can return the `EJBMetaData` for the bean, just like an entity bean. `EJBMetaData` is a serializable object that provides information about the bean's interfaces. The only difference between the `EJBMetaData` for a session bean and an entity bean is that the `getPrimaryKeyClass()` on the session bean's `EJBMetaData` throws a `java.lang.RuntimeException` when invoked:

```
public interface javax.ejb.EJBMetaData {
    public abstract EJBHome getEJBHome();
    public abstract Class getHomeInterfaceClass();
    public abstract Class getPrimaryKeyClass();
    public abstract Class getRemoteInterfaceClass();
    public abstract boolean isSession();
    public abstract boolean isStateless();  // EJB 1.0 only
}
```

ProcessPaymentBean: The bean class

As stated earlier, the ProcessPayment EJB accesses data that is not generally shared by systems, so it is an excellent candidate for a stateless session bean. This bean really represents a set of independent operations that can be invoked and then thrown away—another indication that it's a good candidate for a stateless session bean. Here is the definition of the `ProcessPaymentBean` class, which supports the remote interface functionality:

```
package com.titan.processpayment;
import com.titan.customer.*;

import java.sql.*;
import java.rmi.RemoteException;
import javax.ejb.SessionContext;

import javax.naming.InitialContext;
import javax.sql.DataSource;
import javax.ejb.EJBException;
import javax.naming.NamingException;

public class ProcessPaymentBean implements javax.ejb.SessionBean {

    final public static String CASH = "CASH";
    final public static String CREDIT = "CREDIT";
```



```
final public static String CHECK = "CHECK";

public SessionContext context;

public void ejbCreate() {
}

public boolean byCash(CustomerRemote customer,
                      double amount)
throws PaymentException{
    return process(getCustomerID(customer),amount,
                   CASH,null,-1,-1,null);
}

public boolean byCheck(CustomerRemote customer,
                       CheckDO check, double amount)
throws PaymentException{
    int minCheckNumber = getMinCheckNumber();
    if (check.checkNumber > minCheckNumber) {
      return process(getCustomerID(customer), amount, CHECK,
                   check.checkBarCode,check.checkNumber,
                   -1,null);
    }
    else {
       throw new PaymentException(
          "Check number is too low. Must be at least "+
           minCheckNumber);
    }
}
public boolean byCredit(CustomerRemote customer,
                        CreditCardDO card, double amount)
throws PaymentException {
    if (card.expiration.before(new java.util.Date())) {
       throw new PaymentException("Expiration date has"+
                                                " passed");
    }
    else {
       return process(getCustomerID(customer), amount,
                      CREDIT, null,-1, card.number,
             new java.sql.Date(card.expiration.getTime()));
    }
}
private boolean process(Integer customerID, double amount,
                        String type, String checkBarCode,
                        int checkNumber, long creditNumber,
                        java.sql.Date creditExpDate)
    throws PaymentException{

    Connection con = null;

    PreparedStatement ps = null;
```



```
        try {
            con = getConnection();
            ps = con.prepareStatement
             ("INSERT INTO payment (customer_id, amount, type,"+
                  "check_bar_code,check_number,credit_number,"+
                  "credit_exp_date) VALUES (?,?,?,?,?,?,?)");
            ps.setInt(1,customerID.intValue());
            ps.setDouble(2,amount);
            ps.setString(3,type);
            ps.setString(4,checkBarCode);
            ps.setInt(5,checkNumber);
            ps.setLong(6,creditNumber);
            ps.setDate(7,creditExpDate);
            int retVal = ps.executeUpdate();
            if (retVal!=1) {
                throw new EJBException("Payment insert failed");
            }
            return true;
        } catch(SQLException sql) {
            throw new EJBException(sql);
        } finally {
             try {
                 if (ps != null) ps.close();
                 if (con!= null) con.close();
             } catch(SQLException se){se.printStackTrace();}
        }
    }
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public void setSessionContext(SessionContext ctx) {
        context = ctx;
    }
    private Integer getCustomerID(Customer customer) {
        try {
            (Integer)customer.getPrimaryKey();
        } catch(RemoteException re) {
            throw new EJBException(re);
        }
    }
    private Connection getConnection() throws SQLException {
        // Implementations shown below
    }
    private int getMinCheckNumber() {
        // Implementations shown below
    }
}
```

The three payment methods all use the private helper method `process()`, which does the work of adding the payment to the database. This strategy

reduces the possibility of programmer error and makes the bean easier to maintain. The `process()` method simply inserts the payment information into the `PAYMENT` table. The use of JDBC in this method should be familiar to you from your work on the bean-managed Ship EJB in Chapter 10. The JDBC connection is obtained from the `getConnection()` method as shown in the following code listing.

```
private Connection getConnection() throws SQLException {
    try {
        InitialContext jndiCntx = new InitialContext();
        DataSource ds = (DataSource)
            jndiCntx.lookup("java:comp/env/jdbc/titanDB");
        return ds.getConnection();
    } catch(NamingException ne){throw new EJBException(ne);}
}
```

The `byCheck()` and the `byCredit()` methods contain some logic to validate the data before processing it. The `byCredit()` method verifies that the credit card's expiration data does not precede the current date. If it does, a `PaymentException` is thrown.

The `byCheck()` method verifies that the check is above a minimum number, as determined by a property that's defined when the bean is deployed. If the check number is below this value, a `PaymentException` is thrown. The property is obtained from the `getMinCheckNumber()` method. We can use the JNDI ENC to read the value of the `minCheckNumber` property.

```
private int getMinCheckNumber() {
         try {
            InitialContext jndiCntx = new InitialContext( );
            Integer value = (Integer)
                jndiCntx.lookup("java:comp/env/minCheckNumber");
            return value.intValue();
         } catch(NamingException ne){throw new EJBException(ne);}
}
```

Thus, we are using an environment property set in the deployment descriptor to change the business behavior of a bean. It is a good idea to capture thresholds and other limits in the environment properties of the bean rather than hardcoding them. This gives you greater flexibility. If, for example, Titan decided to raise the minimum check number, you would only need to change the bean's deployment descriptor, not the class definition. (You could also obtain this type of information directly from the database.)

JNDI ENC: Accessing environment properties

In EJB, the bean container contract includes the JNDI environment naming context (JNDI ENC). The JNDI ENC is a JNDI name space that is specific to each bean type. This name space can be referenced from within any bean, not just entity beans, using the name `"java:comp/env"`. The enterprise naming

context provides a flexible, yet standard, mechanism for accessing properties, other beans, and resources from the container.

We've already seen the JNDI ENC several times. In Chapter 10, we used it to access a resource factory, the `DataSource`. The `ProcessPaymentBean` also uses the JNDI ENC to access a `DataSource` in the `getConnection()` method; further, it uses the JNDI ENC to access an environment property in the `getMinCheckNumber()` method. This section examines the use of the JNDI ENC to access environment properties.

Named properties can be declared in a bean's deployment descriptor. The bean accesses these properties at runtime by using the JNDI ENC. Properties can be of type `String` or one of several primitive wrapper types including `Integer`, `Long`, `Double`, `Float`, `Byte`, `Boolean`, and `Short`. By modifying the deployment descriptor, the bean deployer can change the bean's behavior without changing its code. As we've seen in the ProcessPayment EJB, we could change the minimum check number that we're willing to accept by modifying the `minCheckNumber` property at deployment. Two ProcessPayment EJBs deployed in different containers could easily have different minimum check numbers, as shown in the following example:

```
<ejb-jar>
   <enterprise-beans>
      <session>
          <env-entry>
             <env-entry-name>minCheckNumber</env-entry-name>
             <env-entry-type>java.lang.Integer</env-entry-type>
             <env-entry-value>2000</env-entry-value>
          </env-entry>
            ...
      </session>
        ...
  <enterprise-beans>
    ...
</ejb-jar>
```

EJBContext

The `EJBContext.getEnvironment()` method is optional in EJB 2.0 and 1.1, which means that it may or may not be supported. If it is not functional, the method will throw a `RuntimeException`. If it is functional, it returns only those values declared in the deployment descriptor as follows (where `minCheckNumber` is the property name):

```
<ejb-jar>
    <enterprise-beans>
        <session>
            <env-entry>
                <env-entry-name>
                    ejb10-properties/minCheckNumber
                </env-entry- name>
```

```
                <env-entry-type>
                     java.lang.String
                </env-entry-name>
                <env-entry-value>20000</env-entry-value>
            </env-entry>
            ...
        </session>
        ...
    </enterprise-beans>
     ...
</ejb-jar>
```

The `ejb10-properties` subcontext specifies that the property `minCheckNumber` is available from both JNDI ENC context `"java:comp/env/ejb10-properties/minCheckNumber"` (as a `String` value), and from the `getEnvironment()` method.

Only those properties declared under the `ejb10-properties` subcontext are available via the `EJBContext`. Furthermore, such properties are only available through the `EJBContext` in containers that choose to support the EJB 1.0 `getEnvironment()` method; all other containers will throw a `RuntimeException`. It's expected that most EJB 2.0 vendors will have dropped support for this feature. In either case, developers are encouraged to use the JNDI ENC to obtain property values and to stop using the `EJBContext.getEnvironment()` method.

The ProcessPayment EJB's deployment descriptor

Deploying the ProcessPayment EJB presents no significant problems. It's essentially the same as deploying entity beans, except that the ProcessPayment EJB has no primary key or persistent fields. Here is the XML deployment descriptor for the ProcessPayment EJB:

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
 <enterprise-beans>
   <session>
      <description>
            A service that handles monetary payments.
      </description>
      <ejb-name>ProcessPaymentBean</ejb-name>
      <home>
        com.titan.processpayment.ProcessPaymentHomeRemote
      </home>
      <remote>
         com.titan.processpayment.ProcessPaymentRemote
```



```
      </remote>
      <ejb-class>
         com.titan.processpayment.ProcessPaymentBean
      </ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
           <env-entry-name>minCheckNumber</env-entry-name>
           <env-entry-type>java.lang.Integer</env-entry-type>
           <env-entry-value>2000</env-entry-value>
      </env-entry>
      <resource-ref>
           <description>DataSource for the Titan database</description>
           <res-ref-name>jdbc/titanDB</res-ref-name>
           <res-type>javax.sql.DataSource</res-type>
           <res-auth>Container</res-auth>
      </resource-ref>

  </session>
</enterprise-beans>

<assembly-descriptor>
  <security-role>
     <description>
        This role represents everyone who is allowed full access
        to the ProcessPayment EJB.
     </description>
    <role-name>everyone</role-name>
  </security-role>

  <method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>ProcessPaymentBean</ejb-name>
        <method-name>*</method-name>
    </method>
  </method-permission>

  <container-transaction>
    <method>
       <ejb-name>ProcessPaymentBean</ejb-name>
       <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

The deployment descriptor for EJB 1.1 is exactly the same, except its header specifies the EJB 1.1 specification and deployment descriptor.

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
```



```
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
```

📖 Exercise 12.1, The ProcessPayment EJB

EJB 2.0: Local Component Interfaces

Like entity beans, stateless session beans can define local component interfaces. This allows the local interfaces of a stateless session bean to be used by other co-located enterprise beans, including other stateless and stateful session beans and even entity beans. Obviously, it's more efficient to use local component interfaces between two beans in the same container system than to use the remote interfaces.

The process of defining local interfaces for a stateless or stateful session bean is the same as that for entity beans. The local interfaces extend `javax.ejb.EJBLocalObject` (for business methods) and `javax.ejb.EJBLocalHome` (for the home interfaces). These interfaces are then defined in the XML deployment descriptor in the `<local>` and `<local-home>` elements.

For the sake of brevity, we will not define local interfaces for either the stateless ProcessPayment EJB or the stateful TravelAgent EJB developed in the next section. Your experience in Chapters 5, 6, and 7 at creating local interfaces for entity beans can be applied easily to any kind of session bean.

The Life Cycle of a Stateless Session Bean

Just as the entity bean has a well-defined life cycle, so does the stateless session bean. The stateless session bean's life cycle has two states: *Does Not Exist* and *Method-Ready Pool*. The Method-Ready Pool is similar to the instance pool used for entity beans. This is one of the significant life-cycle differences between stateless and stateful session beans; stateless beans define instance pooling in their life cycle and stateful beans do not.[1] Figure 12-1 illustrates the states and transitions that a stateless session bean instance goes through in its lifetime.

[FIGURE]

Figure 12-1: Stateless session bean life cycle

1 Some vendors do *not* pool stateless instances, but may instead create and destroy instances with each method invocation. This is an implementation-specific decision that shouldn't impact the specified life cycle of the stateless bean instance.

Does Not Exist

When a bean is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.

The Method-Ready Pool

Stateless bean instances enter the Method-Ready Pool as the container needs them. When the EJB server is first started, it will probably create a number of stateless bean instances and enter them into the Method-Ready Pool. (The actual behavior of the server depends on the implementation.) When the number of stateless instances servicing client requests is insufficient, more can be created and added to the pool.

Transitioning to the Method-Ready Pool

When an instance transitions from the Does Not Exist state to the Method-Ready Pool, three operations are performed on it. First, the bean instance is instantiated by invoking the `Class.newInstance()` method on the stateless bean class.

> Enterprise bean classes, entity, session and message-driven beans alike, must never define constructors. Take care of initialization needs within `ejbCreate()` and other callback methods. The container instantiates instances of the bean class using `Class.newInstance()`, which requires a no-argument constructor.

> Although the life cycle of a bean instance is defined by the specification, the actual implementation by EJB vendors need only support the specified life cycle as perceived by the bean class and the client. For this reason, a bean developer must only depend on behavior described by the specification. The specification does not describe the behavior of Java language constructors; it only describes the behavior of the create and callback methods in the bean class.

Second, the `SessionBean.setSessionContext(SessionContext context)` method is invoked on the bean instance. This is when the instance receives its reference to the `EJBContext` for its lifetime. The `SessionContext` reference may be stored in a nontransient instance field of the stateless session bean.

Finally, the no-argument `ejbCreate()` method is invoked on the bean instance. Remember that a stateless session bean only has one `ejbCreate()` method, which takes no arguments. The `ejbCreate()` method is invoked only

once in the life cycle of the stateless session bean; when the client invokes the `create()` method on the EJB home, it is not delegated to the bean instance.

Stateless session beans are not subject to activation, so they can maintain open connections to resources for their entire life cycle.[2] The `ejbRemove()` method should close any open resources before the stateless session bean is evicted from memory at the end of its life cycle. More about `ejbRemove()` later in this section.

Life in the Method-Ready Pool

Once an instance is in the Method-Ready Pool, it is ready to service client requests. When a client invokes a business method on an EJB object, the method call is delegated to any available instance in the Method-Ready Pool. While the instance is executing the request, it is unavailable for use by other EJB objects. Once the instance has finished, it is immediately available to any EJB object that needs it. This is slightly different from the instance pool for entity beans described in Chapter 11. In the entity instance pool, a bean instance might be swapped in to service an EJB object for several method invocations. Stateless session instances are only dedicated to an EJB object for the duration of the method.

Although vendors can choose different strategies to support stateless session beans, it's likely that vendors will use an instance-swapping strategy similar to that used for entity beans (the strategy utilized by entity beans is described in Chapter 11). However, the swap is very brief, lasting only as long as the business method needs to execute. When an instance is swapped in, its `SessionContext` changes to reflect the context of its EJB object and the client invoking the method. The bean instance may be included in the transactional scope of the client's request, and it may access `SessionContext` infomation specific to the client request, for example, the security and transactional methods. Once the instance has finished servicing the client, it is disassociated from the EJB object and returned to the Method-Ready Pool.

Stateless session beans are not subject to activation and never have their `ejbActivate()` or `ejbPassivate()` callback methods invoked. The reason is simple: stateless instances have no conversational state that needs to be preserved. (*Stateful* session beans do depend on activation, as we'll see later.)

Clients that need a remote or local reference to a stateless session bean begin by invoking the `create()` method on the bean's EJB home:

---

2 The duration of a stateless bean instance's life is assumed to be very long. However, some EJB servers may actually destroy and create instances with every method invocation, making this strategy less attractive. Consult your vendor's documentation for details on how your EJB server handles stateless instances.


```
Object ref = jndiConnection.lookup("ProcessPaymentHome");
ProcessPaymentHomeRemote home = (ProcessPaymentHomeRemote)
PortableRemoteObject.narrow(ref,ProcessPaymentHomeRemote.class);

ProcessPaymentRemote pp = home.create();
```

Unlike the entity bean and stateful session bean, invoking the `create()` method does not result in a call to the bean's `ejbCreate()` method. In stateless session beans, calling the EJB home's `create()` method results in the creation of an EJB object for the client, but that is all. The `ejbCreate()` method of a stateless session bean is only invoked once in the life cycle of an instance—when it is transitioning from the Does Not Exist state to the Method-Ready Pool. It isn't reinvoked every time a client requests a remote reference to the bean.

That's why stateless session beans are limited to a single no-argument create method; there is no way for the container to anticipate which create method the client might invoke, so only one standard no-argument `create()` method is allowed.

Transitioning out of the Method-Ready Pool: The death of a stateless bean instance

Bean instances leave the Method-Ready Pool for the Does Not Exist state when the server no longer needs the instance. This occurs when the server decides to reduce the total size of the Method-Ready Pool by evicting one or more instances from memory. The process begins by invoking the `ejbRemove()` method on the instance. At this time, the bean instance should perform any cleanup operations, like closing open resources. The `ejbRemove()` method is only invoked once in the life cycle of a stateless session bean's instance—when it is about to transition to the Does Not Exist state. When a client invokes one of the `remove()` methods on a stateless session bean's remote or home interface, it is not delegated to the bean instance. The client's invocations of this method simply invalidate the stub and releases the EJB object; it notifies the container that the client no longer needs the bean. The container itself invokes the `ejbRemove()` method on the stateless instance, but only at the end of the instance's life cycle. Again, this is very different from both stateful session beans and entity beans, which suffer more destructive consequences when the client invokes a remove method. During the `ejbRemove()` method, the `SessionContext` and access to the JNDI ENC is still available to the bean instance. Following the execution of the `ejbRemove()` method, the bean is dereferenced and eventually garbage collected.

The Stateful Session Bean

Stateful session beans offer an alternative that lies between entity beans and stateless session beans. Stateful session beans are dedicated to one client for

the life of the bean instance; a stateful session bean acts on behalf of a client as its agent. They are not swapped among EJB objects or kept in an instance pool like entity and stateless bean instances. Once a stateful session bean is instantiated and assigned to an EJB object, it is dedicated to that EJB object for its entire life cycle.[3]

Stateful session beans maintain conversational state, which means that the instance variables of the bean class can cache data relative to the client between method invocations. This makes it possible for methods to be interdependent, so that changes made by methods to the bean's state can affect the result of subsequent method invocations. In contrast, the stateless session beans we have been talking about do not maintain conversational state. Although stateless beans may have instance variables, these fields are not specific to one client. A stateless instance is swapped among many EJB objects, so you can't predict which instance will service a method call. With stateful session beans, every method call from a client is serviced by the same instance (at least conceptually), so the bean instance's state can be predicted from one method invocation to the next.

Although stateful session beans maintain conversational state, they are not themselves persistent like entity beans. Entity beans represent data in the database; their persistent fields are written directly to the database. Stateful session beans, like stateless beans, can access the database but do not represent data in the database. In addition, stateful beans are not used concurrently like entity beans. If you have an entity EJB object that wraps an instance of the ship called *Paradise*, for example, all client requests for that ship will be coordinated through the same EJB object.[4] With stateful session beans, the EJB object is dedicated to one client—stateful session beans are not used concurrently.

Stateful session beans are often thought of as extensions of the client. This makes sense if you think of a client as being made up of operations and state. Each task may rely on some information gathered or changed by a previous operation. A GUI client is a perfect example: when you fill in the fields on a GUI client you are creating conversational state. Pressing a button executes an operation that might fill in more fields, based on the information you entered previously. The information in the fields is conversational state.

Stateful session beans allow you to encapsulate the business logic and the conversational state of a client and move it to the server. Moving this logic to

---

3 This is a conceptual model. Some EJB containers may actually use instance swapping with stateful session beans but make it appear as if the same instance is servicing all requests. Conceptually, however, the same stateful session bean instance services all requests.

4 This is a conceptual model. Some EJB containers may actually use separate EJB objects for concurrent access to the same entity, relying on the database to control concurrency. Conceptually, however, the end result is the same.

the server thins the client application and makes the system as a whole easier to manage. The stateful session bean acts as an agent for the client, managing processes or *workflow* to accomplish a set of tasks; it manages the interactions of other beans in addition to direct data access over several operations to accomplish a complex set of tasks. By encapsulating and managing workflow on behalf of the client, stateful beans present a simplified interface that hides the details of many interdependent operations on the database and other beans from the client.

EJB 2.0: Modifying the Reservation EJB

The Reservation EJB that was used in Chapter 7 will be modified slightly so that it can be created with all its relationships identified right away. To accommodate this, we overload the `ejbCreate()` method:

```
public abstract class ReservationBean
implements javax.ejb.EntityBean {
    public Integer ejbCreate(CustomerRemote customer,
                             CruiseLocal cruise,
                             CabinLocal cabin, double price){
        setAmountPaid(price);
    }
    public void ejbPostCreate(CustomerRemote customer,
                              CruiseLocal cruise,
                              CabinLocal cabin, double price)
    throws javax.ejb.CreateException{

        setCruise(cruise);
        setCabin(cabin);
        try{
        Integer primKey = (Integer)customer.getPrimaryKey();
        CustomerLocalHome home = (CustomerLocalHome)
        jndiContext.lookup("java:comp/env/ejb/CustomerHome");
        CustomerLocal custL = home.findByPrimaryKey(primKey);
        setCustomer(custL);
        }catch(FinderException fe){
            throw new CreateException("Invalid Customer");
        }
    }
```

Relationship fields use local EJB object references, so we must convert the `CustomerRemote` reference to a `CustomerLocal` reference in order to set the Reservation EJB's customer relationship field. This is accomplished using the JNDI ENC to locate the local home interface and then executing the `findByPrimaryKey()` method. As an alternative, you could have implemented an `ejbSelect` method in the Reservation EJB to locate the `CustomerLocal` reference.

The TravelAgent EJB

The TravelAgent EJB, which we have already seen, is a stateful session bean that encapsulates the process of making a reservation on a cruise. We will develop this bean further to demonstrate how stateful session beans can be used as workflow objects.

Although the TravelAgent EJB will use the local interfaces of other beans, we will not develop a local interface for the TravelAgent EJB. The rules for developing local interfaces for stateful session beans are the same as those for stateless and entity beans. The TravelAgent EJB is designed to be used only by remote clients and therefore doesn't require a set of local component interfaces.

TravelAgent: The remote interface

In Chapter 4, we developed an early version of the `TravelAgentRemote` interface that contained a single business method, `listCabins()`. We are going to remove the `listCabins()` method and redefine the TravelAgent EJB so that it behaves like a workflow object. Later in the chapter, we will add a modified listing method for obtaining a more specific list of cabins for the user.

As a stateful session bean that models workflow, TravelAgent manages the interactions of several other beans while maintaining conversational state. The following code contains the modified `TravelAgentRemote` interface:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.FinderException;
import com.titan.cruise.Cruise;
import com.titan.customer.Customer;
import com.titan.processpayment.CreditCard;

public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public void setCruiseID(Integer cruise)
    throws RemoteException, FinderException;

    public void setCabinID(Integer cabin)
    throws RemoteException, FinderException;

    public TicketDO bookPassage(CreditCardDO card, double price)
        throws RemoteException,IncompleteConversationalState;
}
```

The purpose of the TravelAgent EJB is to make cruise reservations. To accomplish this task, the bean needs to know which cruise, cabin, and customer make up the reservation. Therefore, the client using the TravelAgent EJB needs to gather this kind of information before making the booking. The `TravelAgentRemote` interface provides methods for setting the IDs of the

cruise and cabin that the customer wants to book. We can assume that the cabin ID came from a list and that the cruise ID came from some other source. The customer is set in the `create()` method of the home interface—more about this later.

Once the customer, cruise, and cabin are chosen, the TravelAgent EJB is ready to process the reservation. This operation is performed by the `bookPassage()` method, which needs the customer's credit card information and the price of the cruise. `bookPassage()` is responsible for charging the customer's account, reserving the chosen cabin in the right ship on the right cruise, and generating a ticket for the customer. How this is accomplished is not important to us at this point; when we are developing the remote interface, we are only concerned with the business definition of the bean. We will discuss the implementation when we talk about the bean class.

Note that the `bookPassage()` method throws an application-specific exception, `IncompleteConversationalState`. This exception is used to communicate business problems encountered while booking a customer on a cruise. The `IncompleteConversationalState` exception indicates that the TravelAgent EJB didn't have enough information to process the booking. The `IncompleteConversationalState` application exception class is defined below:

```
package com.titan.travelagent;

public class IncompleteConversationalState extends java.lang.Exception {
   public IncompleteConversationalState(){super();}
   public IncompleteConversationalState(String msg){super(msg);}
}
```

Dependent Object: TicketDO

Like the `CreditCardDO` and `CheckDO` classes used in the ProcessPayment EJB, the `TicketDO` class that `bookPassage()` returns is defined as a pass-by-value object. It can be argued that a ticket should be an entity bean since it is not dependent and may be accessed outside the context of the TravelAgent EJB. However, determining how a business object is used can also dictate whether it should be a bean or simply a class. The `TicketDO` object, for example, could be digitally signed and emailed to the client as proof of purchase. This wouldn't be feasible if the `TicketDO` object had been an entity bean. Enterprise beans are only referenced through their remote interfaces and are not passed by value, as are serializable objects such as `TicketDO`, `CreditCardDO`, and `CheckDO`. As an exercise in pass-by-value, we define the TicketDO as a simple serializable ojbect instead of a bean.

EJB 2.0: TicketDO

EJB 2.0 utilizes the local interfaces of Customer, Cruise, and Cabin EJB's when creating a new `TicketDO`.


```
package com.titan.travelagent;

import com.titan.cruise.CruiseLocal;
import com.titan.cabin.CabinLocal;
import com.titan.customer.CustomerRemote;

public class TicketDO implements java.io.Serializable {
    public Integer customerID;
    public Integer cruiseID;
    public Integer cabinID;
    public double price;
    public String description;

    public TicketDO(CustomerRemote customer,
                    CruiseLocal cruise, CabinLocal cabin,
                    double price)
    throws javax.ejb.FinderException, RemoteException,
           javax.naming.NamingException {

       description = customer.getFirstName()+
           " " + customer.getLastName() +
           " has been booked for the "
           + cruise.getName() +
           " cruise on ship " +
             cruise.getShip().getName() + ".\n" +
           " Your accommodations include " +
             cabin.getName() +
           " a " + cabin.getBedCount() +
           " bed cabin on deck level " + cabin.getDeckLevel() +
           ".\n Total charge = " + price;
        customerID = (Integer)customer.getPrimaryKey();
        cruiseID = (Integer)cruise.getPrimaryKey();
        cabinID = (Integer)cabin.getPrimaryKey();
        price = amount;
    }

    public String toString() {
        return description;
    }
}
```

EJB 1.1: TicketDO

EJB 1.1 utilizes the remote interfaces of Customer, Cruise, and Cabin EJB's when creating a new `TicketDO`.

```
package com.titan.travelagent;

import com.titan.cruise.CruiseRemote;
import com.titan.cabin.CabinRemote;
import com.titan.customer.CustomerRemote;
import java.rmi.RemoteException;
```



```
public class TicketDO implements java.io.Serializable {
    public Integer customerID;
    public Integer cruiseID;
    public Integer cabinID;
    public double price;
    public String description;

    public TicketDO(CustomerRemote customer,
                    CruiseRemote cruise, CabinRemote cabin,
                    double price)
    throws javax.ejb.FinderException, RemoteException,
           javax.naming.NamingException {

       description = customer.getFirstName()+
           " " + customer.getLastName() +
           " has been booked for the "
           + cruise.getName() +
           " cruise on ship " + cruise.getShipID() + ".\n" +
           " Your accommodations include " +
             cabin.getName() +
           " a " + cabin.getBedCount() +
           " bed cabin on deck level " + cabin.getDeckLevel() +
           ".\n Total charge = " + price;

        customerID = (Integer)customer.getPrimaryKey();
        cruiseID = (Integer)cruise.getPrimaryKey();
        cabinID = (Integer)cabin.getPrimaryKey();
        price = amount;

    }
    public String toString() {
        return description;
    }
}
```

TravelAgentHomeRemote: The home interface

Starting with the `TravelAgentHomeRemote` interface that we developed in Chapter 4, we can modify the `create()` method to take a remote reference to the customer who is making the reservation:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import com.titan.customer.Customer;

public interface TravelAgentHomeRemote extends javax.ejb.EJBHome {

    public TravelAgent create(CustomerRemote cust)
        throws RemoteException, CreateException;
```

```
}
```

The `create()` method in this home interface requires that a remote reference to a Customer EJB be used to create the TravelAgent EJB. Because there are no other `create()` methods, you can't create a TravelAgent EJB if you don't know who the customer is. The Customer EJB reference provides the TravelAgent EJB with some of the conversational state it will need to process the `bookPassage()` method.

Taking a peek at the client view

Before settling on definitions for your component interfaces, it is a good idea to figure out how the bean will be used by clients. Imagine that the TravelAgent EJB is used by a Java application with GUI fields. The GUI fields capture the customer's preference for the type of cruise and cabin. We start by examining the code used at the beginning of the reservation process:

```
Context jndiContext = getInitialContext();
Object ref = jndiContext.lookup("CustomerHome");
CustomerHomeRemote customerHome =(CustomerHomeRemote)
    PortableRemoteObject.narrow(ref, CustomerHomeRemote.class);

String ln = tfLastName.getText();
String fn = tfFirstName.getText();
String mn = tfMiddleName.getText();
Customer customer = customerHome.create(nextID, ln, fn, mn);

ref = jndiContext.lookup("TravelAgentHome");
TravelAgentHomeRemote home = (TravelAgentHomeRemote)
PortableRemoteObject.narrow(ref, TravelAgentHomeRemote.class);

TravelAgentRemote agent = home.create(customer);
```

This snippet of code creates a new Customer EJB based on information the travel agent gathered over the phone. The `CustomerRemote` reference is then used to create a TravelAgent EJB. Next, we gather the cruise and cabin choices from another part of the applet:

```
Integer cruise_id =
                  new Integer(textField_cruiseNumber.getText());

Integer cabin_id =
                  new Integer( textField_cabinNumber.getText());

agent.setCruiseID(cruise_id);
agent.setCabinID(cabin_id);
```

The user chooses the cruise and cabin that the customer wishes to reserve. These IDs are set in the TravelAgent EJB, which maintains the conversational state for the whole process.

At the end of the process, the travel agent completes the reservation by processing the booking and generating a ticket. Because the TravelAgent EJB has maintained the conversational state, caching the customer, cabin, and cruise information, only the credit card and price are needed to complete the transaction:

```
long cardNumber = Long.parseLong(textField_cardNumber.getText());
Date date =
    dateFormatter.format(textField_cardExpiration.getText());
String cardBrand = textField_cardBrand.getText();
CreditCardDO card = new CreditCardDO(cardNumber,date,cardBrand);
double price =
double.valueOf(textField_cruisePrice.getText()).doubleValue();
TicketDO ticket = agent.bookPassage(card,price);
PrintingService.print(ticket);
```

We can now move ahead with development; this summary of how the client will use the TravelAgent EJB confirms that our remote interface and home interface definitions are workable.

TravelAgentBean: The bean class

We now implement all the behavior expressed in the new remote interface and home interface for the TravelAgent EJB. Here is a partial definition of the new `TravelAgentBean`:[5]

EJB 2.0: TravelAgentBean

```
import com.titan.reservation.*;

import java.sql.*;
import javax.sql.DataSource;
import java.util.Vector;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.ejb.EJBException;

public class TravelAgentBean implements javax.ejb.SessionBean {

    public CustomerRemote customer;
    public CruiseLocal cruise;
    public CabinLocal cabin;
```

5 If you're modifying the bean developed in Chapter 4, remember to delete the `listCabin()` method. We will add a new implementation of that method later in this chapter.

```
public javax.ejb.SessionContext ejbContext;

public javax.naming.Context jndiContext;

public void ejbCreate(CustomerRemote cust) {
    customer = cust;
}
public void setCabinID(Integer cabinID)
 throws javax.ejb.FinderException {
    try {
        CabinHomeLocal home = (CabinHomeLocal)
        jndiContext.lookup("java:comp/env/ejb/CabinHome");

        cabin = home.findByPrimaryKey(cabinID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID)
    throws javax.ejb.FinderException {
    try {
       CruiseHomeLocal home = (CruiseHomeLocal)
       jndiContext.lookup("java:comp/env/ejb/CruiseHome");

       cruise = home.findByPrimaryKey(cruiseID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }

}
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null)
    {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome =
        (ReservationHomeLocal)
        jndiContext.lookup
                   ("java:comp/env/ejb/ReservationHome");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);

        Object ref = jndiContext.lookup
                   ("java:comp/env/ejb/ProcessPaymentHome");

        ProcessPaymentHomeRemote ppHome =
```



```
            (ProcessPaymentHomeRemote)
            PortableRemoteObject.narrow
                      (ref, ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket =
                   new TicketDO(customer,cruise,cabin,price);
            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}

    public void setSessionContext(javax.ejb.SessionContext cntx)
    {

        ejbContext = cntx;
        try {
          jndiContext = new javax.naming.InitialContext();
        } catch(NamingException ne) {

          throw new EJBException(ne);
        }
    }
}
```

EJB 1.1: TravelAgentBean

```
import com.titan.reservation.*;

import java.sql.*;
import javax.sql.DataSource;
import java.util.Vector;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.ejb.EJBException;

public class TravelAgentBean implements javax.ejb.SessionBean {

    public CustomerRemote customer;
    public CruiseRemote cruise;
    public CabinRemote cabin;

    public javax.ejb.SessionContext ejbContext;

    public javax.naming.Context jndiContext;
```

```
public void ejbCreate(CustomerRemote cust) {
    customer = cust;
}
public void setCabinID(Integer cabinID)
 throws javax.ejb.FinderException {
    try {
        CabinHomeRemote home = (CabinHomeRemote)
        getHome("CabinHome",CabinHomeRemote.class);
        cabin = home.findByPrimaryKey(cabinID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID)
    throws javax.ejb.FinderException {
    try {
       CruiseHomeRemote home = (CruiseHomeRemote)
       getHome("CruiseHome", CruiseHomeRemote. class);
       cruise = home.findByPrimaryKey(cruiseID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }

}
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome =
        (ReservationHomeRemote)getHome("ReservationHome",
                                ReservationHomeRemote.class);
        ReservationRemote reservation =
            resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
                    getHome("ProcessPaymentHome",
                            ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
               new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```



```
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}

    public void setSessionContext(javax.ejb.SessionContext cntx)
    {

        ejbContext = cntx;
        try {
          jndiContext = new javax.naming.InitialContext();
        } catch(NamingException ne) {

          throw new EJBException(ne);
        }
    }
    protected Object getHome(String name,Class type) {
        try {
            Object ref =
                  jndiContext.lookup("java:comp/env/ejb/"+name);
            return PortableRemoteObject.narrow(ref, type);
    } catch(NamingException ne) {
        throw new EJBException(ne);
    }
}
```

There is a lot of code to digest in the `TravelAgentBean` class definition, so we will approach it in small pieces. First, let's examine the `ejbCreate()` method:

```
public class TravelAgentBean implements javax.ejb.SessionBean {

    public CustomerRemote customer;
        ...

    public javax.ejb.SessionContext ejbContext;
    public javax.naming.Context jndiContext;

    public void ejbCreate(CustomerRemote cust) {
        customer = cust;
    }
```

When the bean is created, the remote reference to the Customer EJB is passed to the bean instance and maintained in the `customer` field. The `customer` field is part of the bean's conversational state. We could have obtained the customer's identity as an integer ID and constructed the remote reference to the Customer EJB in the `ejbCreate()` method. However, we passed the reference directly to demonstrate that remote references to beans can be passed from a client application to a bean. They can also be returned from the bean to the client and passed between beans on the same EJB server or between EJB servers.

References to the `SessionContext` and JNDI context are held in fields called `ejbContext` and `jndiContext`. The `ejb` and `jndi` prefixes help to avoid confusion between the different content types.

When a bean is passivated, the JNDI ENC must be maintained as part of the bean's conversational state. This means that the JNDI context should not be `transient`. Once a field is set to reference the JNDI ENC, the reference remains valid for the life of the bean. In the `TravelAgentBean`, we set the field `jndiContext` to reference the JNDI ENC when the `SessionContext` is set a the beginning of the bean's life cycle:

```
public void setSessionContext(javax.ejb.SessionContext cntx) {
    ejbContext = cntx;
    try {
        jndiContext = new InitialContext();
    } catch(NamingException ne) {
        throw new EJBException(ne);
    }
}
```

The EJB container makes special accommodations for references to `SessionContext`, the JNDI ENC, references to other beans (remote and home interface types) and the JTA `UserTransaction` type, which is discussed in detail in Chapter 14. The container must maintain any instance fields that reference objects of these types as part of the conversational state, even if they are not serializable. All other fields must be serializable or `null` when the bean is passivated.

The TravelAgent EJB has methods for setting the desired cruise and cabin. These methods take Integer IDs as arguments and retrieve references to the appropriate Cruise or Cabin EJB from the appropriate home interface. These references are also a part of the TravelAgent EJB's conversational state:

EJB 2.0: setCabinID() and getCabinID()

```
public void setCabinID(Integer cabinID)
    throws javax.ejb.FinderException {
    try {
        CabinHomeLocal home = (CabinHomeLocal)
        jndiContext.lookup("java:comp/env/ejb/CabinHome");

        cabin = home.findByPrimaryKey(cabinID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID)
    throws javax.ejb.FinderException {
    try {
        CruiseHomeLocal home = (CruiseHomeLocal)
```



```
        jndiContext.lookup("java:comp/env/ejb/CruiseHome");

        cruise = home.findByPrimaryKey(cruiseID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
```

EJB 1.1: setCabinID() and getCabinID()

```
public void setCabinID(Integer cabinID)
    throws javax.ejb.FinderException {
    try {
        CabinHomeRemote home =
        (CabinHome)getHome("CabinHome",CabinHome.class);
        cabin = home.findByPrimaryKey(cabinID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }
}
public void setCruiseID(Integer cruiseID)
    throws javax.ejb.FinderException {
    try {
        CruiseHome home =
        (CruiseHome)getHome("CruiseHome", CruiseHome. class);
        cruise = home.findByPrimaryKey(cruiseID);
    } catch(RemoteException re) {
        throw new EJBException(re);
    }

}
```

It may seem strange that we set these values using the Integer IDs, but we keep them in the conversational state as entity bean references. Using the Integer IDs for these objects is simpler for the client, which doesn't work with their entity bean references. In the client code, we get cabin and cruise IDs from text fields. Why make the client obtain a bean reference to the Cruise and Cabin EJBs when an ID is simpler? In addition, using the IDs is cheaper than passing a remote reference in terms of network traffic. We need the EJB object references to these bean types in the `bookPassage()` method, so we use their IDs to obtain actual entity bean references. We could have waited until the `bookPassage()` method was invoked before reconstructing the remote references, but this way we keep the `bookPassage()` method simple.

JNDI ENC and EJB References

The JNDI ENC can be used to obtain a reference to the home interface of other beans. Using the ENC lets you avoid hardcoding vendor-specific JNDI

properties into the bean. In other words, the JNDI ENC allows EJB references to be network and vendor independent.

In the EJB 2.0 listing for the `TravelAgentBean`, the JNDI ENC is used to access both the remote home interface of the ProcessPayment EJB as well as the local home interfaces of the Cruise and Cabin EJBs. This illustrates the flexiblity of the JNDI ENC, which can provide a directory for both local and remote enterprise beans.

In the EJB 1.1 listing for the `TravelAgentBean` class, `getHome()` is a convenience method that hides the details of obtaining remote references to EJB home objects. The `getHome()` method uses the `jndiContext` reference to obtain references to the Cabin, Ship, ProcessPayment, and Cruise home objects.

The EJB specification recommends that all EJB references be bound to the `"java:comp/env/ejb"` context, which is the convention followed here. In the TravelAgent EJB, we pass in the name of the home object we want and append it to the `"java:comp/env/ejb"` context to do the lookup.

Remote EJB references in the JNDI ENC

The deployment descriptor provides a special set of tags for declaring remote EJB references. Here's how the `<ejb-ref>` tag and its subelements are used:

```
<ejb-ref>
    <ejb-ref-name>ejb/ProcessPaymentHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <home>
        com.titan.processpayment.ProcessPaymentHomeRemote
    </home>
    <remote>
       com.titan.processpayment.ProcessPaymentRemote
    </remote>
</ejb-ref>
```

The `<ejb-ref>` tag and its subelements should be self-explanatory: they define a name for the bean within the ENC, declare the bean's type, and give the names of its remote and home interfaces. When a bean is deployed, the deployer maps the `<ejb-ref>` elements to actual beans in a way specific to the vendor. The `<ejb-ref>` elements can also be linked by the application assembler to beans in the same deployment (a subject covered in detail in Chapter 16, which is about the XML deployment descriptors). EJB 2.0 developers should try to use local component interfaces for beans located in the same deployment and container.

At deployment time, the EJB container's tools map the remote references declared in the `<ejb-ref>` elements to entity beans in other EJB containers, which might located on the same machine or at a different node on the network.

EJB 2.0: Remote EJB references in the JNDI ENC

The deployment descriptor also provides a special set of tags, the `<ejb-local-ref>` elements, to declare local EJB references: enterprise beans that are co-located in the same container and deployed in the same EJB JAR file. The `<ejb-local-ref>` elements are declared immediately after the `<ejb-ref>` elements.

```
<ejb-local-ref>
    <ejb-ref-name>ejb/CruiseHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    < local-home >
        com.titan.cruise.CruiseHomeLocal
    </local-home >
    <local>
        com.titan.cruise.CruiseLocal
    </local>
    <ejb-link>CruiseEJB</ejb-link>
</ejb-local-ref>
<ejb-local-ref>
    <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home >
        com.titan.cabin.CabinHomeLocal
    </local-home >
    <local>
        com.titan.cabin.CabinLocal
    </local>
    <ejb-link>CabinEJB</ejb-link>
</ejb-local-ref>
```

The `<ejb-local-ref>` tag defines a name for the bean within the ENC, declares the bean's type, and gives the names of its local component interfaces. The `<ejb-local-ref>` elements should be linked explicitly to other co-located beans using the `<ejb-link>` element, but this is not required—the application assembler or deployer can do it later. The value of the `<ejb-link>` element within the `<ejb-local-ref>` must equal the `<ejb-name>` of the appropriate bean in the same JAR file.

At deployment time the EJB container's tools map the local references declared in the `<ejb-local-ref>` elements to entity beans that are co-located in the same container system.

The bookPassage() method

The last point of interest in our bean definition is the `bookPassage()` method. This method leverages the conversational state accumulated by `ejbCreate()`, `setCabinID()`, and `setCruiseID()` methods to process a reservation for a customer.

EJB 2.0: bookPassage() method

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome =
        (ReservationHomeLocal)
        jndiContext.lookup
                    ("java:comp/env/ejb/ReservationHome");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);

        Object ref = jndiContext.lookup
                    ("java:comp/env/ejb/ProcessPaymentHome");

        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
        PortableRemoteObject.narrow
                    (ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

EJB 1.1: bookPassage() method

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome =
        (ReservationHomeRemote)getHome("ReservationHome",
                                ReservationHomeRemote.class);
        ReservationRemote reservation =
```



```
            resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
                    getHome("ProcessPaymentHome",
                            ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        // EJB 1.0: throw new RemoteException("",e);
        throw new EJBException(e);
    }
}
```

This method exemplifies the workflow concept. It uses several beans, including the Reservation, ProcessPayment, Customer, Cabin and the Cruise EJBs to accomplish one task: book a customer on a cruise. Deceptively simple, this method encapsulates several interactions that ordinarily might have been performed on the client. For the price of one `bookPassage()` call from the client, the TravelAgent EJB performs many operations:

1. Look up and obtain a reference to the Reservation EJB's EJB home.
2. Create a new Reservation EJB resulting in a database insert.
3. Look up and obtain a remote reference to the ProcessPayment EJB's EJB home.
4. Create a new ProcessPayment EJB.
5. Charge the customer's credit card using the ProcessPayment EJB.
6. Generate a new `TicketDO` with all the pertinent information describing the customer's purchase.

From a design standpoint, encapsulating the workflow in a stateful session bean means a less complex interface for the client and more flexibility for implementing changes. We could, for example, easily change the `bookPassage()` method to check for overlapped booking (when a customer books passage on two different cruises that overlap). This type of enhancement would not change the remote interface, so the client application wouldn't need modification. Encapsulating workflow in stateful session beans allows the system to evolve over time without impacting clients.

In addition, the type of clients used can change. One of the biggest problems with two-tier architectures—besides scalability and transactional control—is that the business logic is intertwined with the client logic. This makes it difficult to reuse the business logic in a different kind of client. With stateful session beans this is not a problem, because stateful session beans are an extension of the client but are not bound to the client's presentation. Let's say that our first

implementation of the reservation system used a Java applet with GUI widgets. The TravelAgent EJB would manage conversational state and perform all the business logic while the applet focused on the GUI presentation. If, at a later date, we decide to go to a thin client (HTML generated by a Java servlet, for example), we would simply reuse the TravelAgent EJB in the servlet. Because all the business logic is in the stateful session bean, the presentation (Java applet or servlet or something else) can change easily.

The TravelAgent EJB also provides transactional integrity for processing the customer's reservation. If any one of the operations within the body of the `bookPassage()` method fails, all the operations are rolled back so that none of the changes are accepted. If the credit card can't be charged by the ProcessPayment EJB, the newly created Reservation EJB and its associated record are removed. The transactional aspects of the TravelAgent EJB are explained in detail in Chapter 14.

In EJB 2.0, remote and local EJB references can be used within the same workflow. For example, the `bookPassage()` method uses local references when accessing the Cruise and Cabin beans, but remote references when accessing the ProcessPayment and Customer EJB. This is totally appropriate. The EJB container ensures that failures when accessing remote or local EJB references will impact the entire transaction.

Why use a Reservation entity bean?

If we have a Reservation EJB, why do we need a TravelAgent EJB? Good question! The TravelAgent EJB uses the Reservation EJB to create a reservation, but it also has to charge the customer and generate a ticket. These are not activities that are specific to the Reservation EJB, so they need to be captured in a stateful session bean that can manage workflow and transactional scope. In addition, the TravelAgent EJB also provides listing behavior, which spans concepts in Titan's system. It would have been inappropriate to include any of these other behaviors in the Reservation entity bean. (For EJB 2.0 readers, the Reservation EJB was developed in chapter 7. For EJB 1.1 readers, the code for this bean is available on the O'Reilly web site.)

listAvailableCabins(): Listing behavior

As promised, we are going to bring back the cabin-listing behavior we played around with in Chapter 4. This time, however, we are not going to use the Cabin EJB to get the list; instead, we will access the database directly. Accessing the database directly is a double-edged sword. On one hand, we don't want to access the database directly if entity beans exist that can access the same information. Entity beans provide a safe and consistent interface for a particular set of data. Once an entity bean has been tested and proven, it can be reused throughout the system, substantially reducing data integrity problems. The Reservation EJB is an example of that kind of usage. In addition, entity beans can

pull together disjointed data and apply additional business logic such as validation, limits, and security to ensure that data access follows the business rules.

But entity beans cannot define every possible data access needed, and they shouldn't. One of the biggest problems with entity beans is that they tend to become bloated over time. Huge entity beans with dozens of methods are a sure sign of poor design. Entity beans should be focused on providing data access to a very limited, but conceptually bound, set of data. You should be able to update, read, and insert records or data. Data access that spans concepts, like listing behavior, should not be encapsulated in one entity bean.

Systems always need listing behavior to present clients with choices. In the reservation system, for example, customers need to choose a cabin from a list of *available* cabins. The word *available* is key to the definition of this behavior. The Cabin EJB can provide us with a list of cabins, but it doesn't know whether any given cabin is available. For EJB 2.0, Chapter 7 defined the Cabin-Reservation relationship as *unidirectional* where the Reservation was aware of its Cabin relationships, but not the other way around.

The question of whether a cabin is available is relevant to the process using it—in this case TravelAgent EJB—but may not be relevant to the cabin itself. As an analogy, an automobile entity would not care what road it's on; it is only concerned with characteristics that describe its state and behavior. An automobile-tracking system would be concerned with the location of individual automobiles.

To get availability information, we need to compare the list of cabins on our ship to the list of cabins that have already been reserved. The `listAvailableCabins()` method does exactly that. It uses a complex SQL query to produce a list of cabins that have not yet been reserved for the cruise chosen by the client:

```
public String [] listAvailableCabins(int bedCount)
    throws IncompleteConversationalState {
    if (cruise == null)
        throw new IncompleteConversationalState();

    Connection con = null;
    PreparedStatement ps = null;;
    ResultSet result = null;
    try {
        Integer cruiseID = (Integer)cruise.getPrimaryKey();
        Integer shipID = (Integer)
                        cruise.getShip().getPrimaryKey();
        con = getConnection();
        ps = con.prepareStatement(
            "select ID, NAME, DECK_LEVEL  from CABIN "+
            "where SHIP_ID = ? and ID NOT IN "+
            "(SELECT CABIN_ID FROM RESERVATION “+
```

```
            " WHERE CRUISE_ID = ?)");

        ps.setInt(1,shipID.intValue());
        ps.setInt(2,cruiseID.intValue());
        result = ps.executeQuery();
        Vector vect = new Vector();
        while(result.next()) {
            StringBuffer buf = new StringBuffer();
            buf.append(result.getString(1));
            buf.append(',');
            buf.append(result.getString(2));
            buf.append(',');
            buf.append(result.getString(3));
            vect.addElement(buf.toString());
        }
        String [] returnArray = new String[vect.size()];
        vect.copyInto(returnArray);
        return returnArray;
    }
    catch (Exception e) {
        throw new EJBException(e);
    }
    finally {
        try {
        if (result != null) result.close();
        if (ps != null) ps.close();
        if (con!= null) con.close();
        }catch(SQLException se){se.printStackTrace();}
    }
}
```

EJB 1.1 readers use almost exactly the same code for `listAvailableCabins()` except for how the Ship EJB's ID is obtained. EJB 1.1 readers should replace the line:

```
Integer shipID = (Integer)

cruise.getShip().getPrimaryKey();
```

With the line:

```
Integer shipID =cruise.getShipID();
```

This change is necessary because EJB 1.1 doesn't support relationship fields.

As you can see, the SQL query is complex. It could have been defined using a method like `Cabin.findAvailableCabins(Cruise cruise)` in the Cabin EJB. However, this method would be difficult to implement because the Cabin EJB would need to access the Reservation EJB's data, which is a navigable relationship. Another reason for accessing the database directly is to demonstrate that this kind of behavior is both normal and, in some cases

preferred. In some cases, the query is fairly specific to the scenario and is not reusable. To avoid adding finder methods for every possible query, you can instead simply use direct database access as shown in the `listAvailableCabins()` method. Direct database access generally has less of an impact on performance because the container doesn't have to manifest EJB object references, but its also less reusable. These things must be considered when deciding if a query for information should be done using direct database access or if a new finder method should be defined.

The `listAvailableCabins()` method returns an array of `String` objects to the remote client. This is important because we could have opted to return an collection of remote Cabin references, but we didn't. The reason is simple: we want to keep the client application as lightweight as possible. A list of `String` objects is much more lightweight than the alternative, a collection of remote references. In addition, a collection of remote references means that client would be working with many stubs, each with its own connection to EJB objects on the server. By returning a lightweight string array, we reduce the number of stubs on the client, which keeps the client simple and conserves resources on the server.

To make this method work, you need to create a `getConnection()` method for obtaining a database connection and add it to the `TravelAgentBean`:

```
private Connection getConnection() throws SQLException {
    try {
       DataSource ds = (DataSource)jndiContext.lookup(
           "java:comp/env/jdbc/titanDB");
       return ds.getConnection();
    } catch(NamingException ne) {throw new EJBException(ne);}
}
```

Change the remote interface for TravelAgent EJB to include the `listAvailableCabins()` method:

```
package com.titan.travelagent;

import java.rmi.RemoteException;
import javax.ejb.FinderException;
import com.titan.cruise.Cruise;
import com.titan.customer.Customer;
import com.titan.processpayment.CreditCard;

public interface TravelAgentRemote extends javax.ejb.EJBObject {

    public void setCruiseID(Integer cruise)
    throws RemoteException, FinderException;

    public void setCabinID(Integer cabin)
    throws RemoteException, FinderException;

    public TicketDO bookPassage(CreditCardDO card, double price)
        throws RemoteException,IncompleteConversationalState;
```

```
    public String [] listAvailableCabins(int bedCount)
        throws RemoteException, IncompleteConversationalState;
}
```

EJB 2.0: The TravelAgent deployment descriptor

The following listing is an abbreviated version of the XML deployment descriptor use for the TravelAgent application. It defined not only the TravelAgent EJB, but also the Customer, Cruise, Cabin and Reservation EJBs. The ProcessPayment EJB is not defined in this deployment descriptor because it is assumed to be deployed in a separate JAR file, or possibly even an EJB server on a different network node.

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
 <enterprise-beans>
   <session>
      <ejb-name>TravelAgentBean</ejb-name>
      <home>com.titan.travelagent.TravelAgentHome</home>
      <remote>com.titan.travelagent.TravelAgent</remote>
      <ejb-class>
          com.titan.travelagent.TravelAgentBean
      </ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>

      <ejb-ref>
           <ejb-ref-name>ejb/ProcessPaymentHome</ejb-ref-name>
           <ejb-ref-type>Session</ejb-ref-type>
           <home>
              com.titan.processpayment.ProcessPaymentHomeRemote
           </home>
           <remote>
              com.titan.processpayment.ProcessPaymentRemote
           </remote>
      </ejb-ref>
      <ejb-local-ref>
           <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
           <ejb-ref-type>Entity</ejb-ref-type>
           <local-home>
              com.titan.cabin.CabinHomeLocal
           </local-home>
           <local>com.titan.cabin.CabinLocal</local>
      </ejb-local-ref>
      <ejb-local-ref>
           <ejb-ref-name>ejb/CruiseHome</ejb-ref-name>
```



```
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>
               com.titan.cruise.CruiseHomeLocal
            </local-home>
            <local>com.titan.cruise.CruiseLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ReservationHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>
               com.titan.reservation.ReservationHomeLocal
            </local-home>
            <local>com.titan.reservation.ReservationLocal</local>
        </ejb-local-ref>

        <resource-ref>
            <description>
                DataSource for the Titan database
            </description>
            <res-ref-name>jdbc/titanDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </session>
    <entity>
        <ejb-name>CabinEJB</ejb-name>
        <local-home>com.titan.cabin.CabinHomeLocal</local-home>
        <local>com.titan.cabin.CabinLocal</local>
        ...
    </entity>
    <entity>
        <ejb-name>CruiseEJB</ejb-name>
        <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
        <local>com.titan.cruise.CruiseLocal</local>
        ...
    </entity>
    <entity>
        <ejb-name>ReservationEJB</ejb-name>
        <local-home>
           com.titan.reservation.ReservationHomeLocal
        </local-home>
        <local>com.titan.reservation.ReservationLocal</local>
        ...
    </entity>
</enterprise-beans>
<assembly-descriptor>
    <security-role>
        <description>
           This role represents everyone
        </description>
       <role-name>everyone</role-name>
```

```
  </security-role>

  <method-permission>
    <role-name>everyone</role-name>
    <method>
        <ejb-name>TravelAgentBean</ejb-name>
        <method-name>*</method-name>
    </method>
  </method-permission>

  <container-transaction>
    <method>
       <ejb-name>TravelAgentBean</ejb-name>
       <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
 </assembly-descriptor>
</ejb-jar>
```

EJB 1.1: The TravelAgent deployment descriptor

Use the following XML deployment descriptor when deploying the TravelAgent EJB. The most important difference between this descriptor and the deployment descriptor used for the ProcessPayment EJB is the `<session-type>` tag, which states that this bean is stateful, and the use of the `<ejb-ref>` elements to describe beans that are referenced through the ENC:

```
<?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>
 <enterprise-beans>
   <session>
      <description>
          Acts as a travel agent for booking passage on a ship.
      </description>
      <ejb-name>TravelAgentBean</ejb-name>
      <home>com.titan.travelagent.TravelAgentHome</home>
      <remote>com.titan.travelagent.TravelAgent</remote>
      <ejb-class>
          com.titan.travelagent.TravelAgentBean
      </ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>

      <ejb-ref>
           <ejb-ref-name>ejb/ProcessPaymentHome</ejb-ref-name>
           <ejb-ref-type>Session</ejb-ref-type>
```



```
            <home>
               com.titan.processpayment.ProcessPaymentHome
            </home>
            <remote>
               com.titan.processpayment.ProcessPayment
            </remote>
        </ejb-ref>
        <ejb-ref>
            <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.titan.cabin.CabinHome</home>
            <remote>com.titan.cabin.Cabin</remote>
        </ejb-ref>
        <ejb-ref>
            <ejb-ref-name>ejb/CruiseHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.titan.cruise.CruiseHome</home>
            <remote>com.titan.cruise.Cruise</remote>
        </ejb-ref>
        <ejb-ref>
            <ejb-ref-name>ejb/CustomerHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.titan.customer.CustomerHome</home>
            <remote>com.titan.customer.Customer</remote>
        </ejb-ref>
        <ejb-ref>
            <ejb-ref-name>ejb/ReservationHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.titan.reservation.ReservationHome</home>
            <remote>com.titan.reservation.Reservation</remote>
        </ejb-ref>

        <resource-ref>
            <description>
                DataSource for the Titan database
            </description>
            <res-ref-name>jdbc/titanDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </session>
</enterprise-beans>

<assembly-descriptor>
  <security-role>
     <description>
        This role represents everyone
     </description>
    <role-name>everyone</role-name>
  </security-role>
```

```
    <method-permission>
      <role-name>everyone</role-name>
      <method>
          <ejb-name>TravelAgentBean</ejb-name>
          <method-name>*</method-name>
      </method>
    </method-permission>

    <container-transaction>
      <method>
         <ejb-name>TravelAgentBean</ejb-name>
         <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Once you have generated the deployment descriptor, *jar* the TravelAgent EJB and deploy it in your EJB server. You will also need to deploy the Reservation, Cruise, and Customer EJBs that you downloaded earlier. Based on the business methods in the remote interface of the TravelAgent EJB and your past experiences with the Cabin, Ship, and ProcessPayment EJBs, you should be able to create your own client application to test this code.

📖 Exercise 12.2, The TravelAgent EJB

The Life Cycle of a Stateful Session Bean

The biggest difference between the stateful session bean and the other bean types is that stateful session beans don't use instance pooling. Stateful session beans are dedicated to one client for their entire life, so there is no swapping or pooling of instances.[6] Instead of pooling instances, stateful session beans are simply evicted from memory to conserve resources. The EJB object remains connected to the client, but the bean instance is dereferenced and garbage collected during inactive periods. This means that a stateful bean must be passivated before it is evicted to preserve the conversational state of the instance, and it must be activated to restore the state when the EJB object becomes active again.

---

6 Some vendors use pooling with stateful session beans, but that is a proprietary implementation and shouldn't impact the specified life cycle of the stateful session bean.

The bean's perception of its life cycle depends on whether or not it implements a special interface called `javax.ejb.SessionSynchronization`. This interface defines an additional set of callback methods that notify the bean of its participation in transactions. A bean that implements `SessionSynchronization` can cache database data across several method calls before making an update. We have not discussed transactions in detail yet, so we will not consider this part of the bean's life cycle until Chapter 14. This section describes the life cycle of stateful session beans that do not implement the `SessionSynchronization` interface.

The life cycle of a stateful session bean has three states: Does Not Exist, Method-Ready, and Passivated. This sounds a lot like a stateless session bean, but the Method-Ready state is significantly different from the Method-Ready Pool of stateless beans. Figure 12-2 shows the state diagram for stateful session beans.

[FIGURE]

Figure 12-2: stateful session bean life cycle

Does Not Exist State

When a stateful bean instance is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.

The Method-Ready State

Transitioning to the Method-Ready state

When a client invokes the `create()` method on an EJB home of a stateful session bean, its life cycle begins. When the `create()` method is received by the container, the container invokes `newInstance()` on the bean class, creating a new instance of the bean. Next, the container invokes `setSessionContext()` on the instance, handing it its reference to the `SessionContext`, which it must maintain for life. At this point, the bean instance is assigned to its EJB object. Finally, the container invokes the `ejbCreate()` method on the instance that matches the `create()` method invoked by the client. Once `ejbCreate()` has completed, the container returns the EJB object's reference to the client. The instance is now in the Method-Ready State and is ready to service business methods invoked by the client on the bean's remote reference.

Life in the Method-Ready state

While in the Method-Ready State, the bean instance is free to receive method invocations from the client, which may involve controlling the workflow of other

beans or accessing the database directly. During this time, the bean can maintain conversational state and open resources in its instance variables.

Transitioning out of the Method-Ready state

Bean instances leave the Method-Ready state to enter either the Passivated state or the Does Not Exist state. During its lifetime, a bean instance will be passivated and activated zero or more times. Depending on how the client uses the stateful bean, the EJB container's load, and the passivation algorithm used by the vendor, a bean instance may be passivated several times in its life or not at all. The bean enters the Does Not Exist state if it is removed. A client application can remove a bean by invoking one of the `remove()` methods on the client API, or the container can choose to remove the bean.

The container can also move the bean instance from the Method-Ready State to the Does Not Exist state if the bean times out. Timeouts are declared at deployment time in a manner specific to the EJB vendor. When a timeout occurs, the `ejbRemove()` method is *not* invoked. A stateful bean cannot time out while a transaction is in progress.

Passivated State

During the lifetime of a stateful session bean, there may be periods of inactivity, when the bean instance is not servicing methods from the client. To conserve resources, the container can passivate the bean instance while it is inactive by preserving its conversational state and evicting the bean instance from memory.

When a stateful bean is passivated, the instance fields are read and then written to the secondary storage associated with the EJB object. When the stateful session bean has been successfully passivated, the instance is evicted from memory; it is destroyed.

When a bean is about to be passivated, its `ejbPassivate()` method is invoked, alerting the bean instance that it is about to enter the Passivated state. At this time, the bean instance should close any open resources and set all nontransient, nonserializable fields to `null`. This will prevent problems from occurring when the bean is serialized. Transient fields will simply be ignored.

A bean's conversational state may consist of only primitive values, objects that are serializable, and the following special types:

EJB 2.0 and 1.1

- `javax.ejb.SessionContext`
- `javax.ejb.EJBHome` (home interface types)
- `javax.ejb.EJBObject` (remote interface types)
- `javax.jta.UserTransaction` (bean transaction interface)
- `javax.naming.Context` (only when it references the JNDI ENC)

EJB 2.0 only

- `javax.ejb.EJBLocalHome` (home interface types)
- `javax.ejb.EJBLocalObject` (remote interface types)
- References to Managed Resource Factories (e.g., `javax.sql.DataSource`)

The types in this list (and their subtypes) are handled specially by the passivation mechanism. They don't need to be serializable; they will be maintained through passivation and restored automatically to the bean instance when it is activated.

A bean instance's conversational state will be written to secondary storage to preserve it when the instance is passivated and destroyed. Containers can use standard Java serialization to preserve the bean instance, or some other mechanism that achieves the same result. Some vendors, for example, will simply read the values of the fields and store them in a cache. The container is required to preserve remote references to other beans with the conversational state. When the bean is activated, the container must restore any bean references automatically. The container must also restore any references to the special types listed earlier.

Fields declared `transient` will not be preserved when the bean is passivated. Except for the special types listed earlier, all fields that are nontransient and nonserializable must be set to `null` before the instance is passivated or else the container will destroy the bean instance, making it unavailable for continued use by the client. References to special types must automatically be preserved with the serialized bean instance by the container so that they can be reconstructed when the bean is activated.

When the client makes a request on an EJB object whose bean is passivated, the container activates the instance. This involves deserializing the bean instance and reconstructing the `SessionContext` reference, bean references, and managed resource factories (EJB 2.0 only) held by the instance before it was passivated. When a bean's conversational state has been successfully restored, the `ejbActivate()` method is invoked. The bean instance should open any resources that cannot be passivated and initialize the value of any transient fields within the `ejbActivate()` method. Once `ejbActivate()` is complete, the bean is back in the Method-Ready state and available to service client requests delegated by the EJB object.

In EJB 1.1, open resources such as sockets or JDBC connections must be closed whenever the bean is passivated. In stateful session beans, open resources will not be maintained for the life of the bean instance. When a stateful session bean is passivated, any open resource can cause problems with the activation mechanism.

The activation of a bean instance follows the rules of Java serialization. The exception to this is transient fields. In Java serialization, transient fields are set to their default values when an object is deserialized; primitive numbers become zero, Boolean fields `false`, and object references `null`. In EJB, transient fields do not have to be set to their initial values; therefore, they could contain arbitrary values when the bean is activated. The value held by transient fields following activation is unpredictable across vendor implementations, so don't depend on them to be initialized. Instead, use `ejbActivate()` to reset their values.

System Exceptions

Whenever a system exception is thrown by a bean method, the container invalidates the EJB object and destroys the bean instance. The bean instance moves directly to the Does Not Exist state and the `ejbRemove()` method is *not* invoked.

A system exception is any unchecked exception, including `EJBException`. Checked exceptions thrown from subsystems are usually wrapped in an `EJBException` and rethrown as system exceptions. A checked exception thrown by a subsystem does not need to be handled this way if the bean can safely recover from the exception. In most cases, however, the subsystem exception should be rethrown as an `EJBException`.

In EJB 1.1, the `java.rmi.RemoteException` is also considered a system exception for backward compatibly with EJB 1.0. However, throwing the `RemoteException` from a bean class method is discouraged. Throwing a `RemoteException` from a bean class method has been deprecated.

13

Message-Driven Beans

This section is divided into two subsections: *JMS as a Resource*, and *Message-Driven Beans.* The first section describes the Java Message Service (JMS) and its role as a resource that is available to any enterprise bean (session, entity, or message-driven). An enterprise bean can use the JMS API to send messages to other applications through a virtual channel called a topic or queue. Readers unfamiliar with JMS should read the first section before proceeding to the second section, which provides an overview of the message-driven bean.

The second section in this chapter addresses the new enterprise bean type, the message-driven bean. A message-driven bean is an asynchronous bean activated by message delivery. In EJB 2.0, vendors are required to support a JMS-based message-driven bean that listens to a specific topic or queue, and processes JMS messages as they are delivered.

All EJB 2.0 vendors must, by default, support a JMS provider. Most EJB 2.0 vendors have a JMS provider built in, but some may also support other JMS providers. For example, VENDOR XXX uses Sonic Software's SonicMQ as its JMS service. Regardless of how the EJB 2.0 vendor provides the JMS service, having one is pretty much a requirement if the vendor expects to support message-driven beans. The advantage of this forced adoption of JMS is that EJB developers cannot expect to have a working JMS provider on which messages can be both consumed and delivered.

JMS as a resource

JMS is a standard vendor-neutral API that is part of the J2EE platform and can be used to access enterprise messaging systems. An enterprise messaging system

(a.k.a. message-oriented middleware) facilitates the exchange of messages among software applications over a network. JMS is analogous to JDBC: Whereas JDBC is an API that can be used to access many different relational databases, JMS provides the same vendor-independent access to enterprise messaging systems. Many enterprise messaging products currently support JMS, including IBM's MQSeries, BEA's Weblogic JMS service, Sun Microsystems' Java Message Queue, and Progress' SonicMQ to name a few. Software applications that use the JMS API for sending or receiving messages are called JMS clients and are portable across brands of JMS vendors.

Messaging clients in JMS are called *JMS clients*, and the messaging system—the MOM—including the JMS service provider is called the *JMS provider*. A *JMS application* is a business system composed of many JMS clients and, generally, one JMS provider.

In EJB, enterprise beans of all types can use JMS to send messages to other Java applications or to message-driven beans. JMS facilitates sending messages from enterprise beans by using a messaging service, sometimes called a message broker or router. Message brokers have been around for a couple of decades the oldest and most established being IBM's MQSeries, but JMS is fairly new and is specifically designed to deliver a variety of messages types from one Java application to another.

Reimplimenting the TravelAgent EJB with JMS

As an example we can modify the TravelAgent EJB developed in Chapter 12 so that it uses JMS to alert some other Java application that a reservation was made. The following code shows how to modify the `bookPassage()` method so that the TravelAgent EJB will send a simple text message based on the description information from the `TicketDO`:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome =
        (ReservationHomeLocal)
        jndiContext.lookup
                    ("java:comp/env/ejb/ReservationHome");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);

        Object ref = jndiContext.lookup
```

```
                    ("java:comp/env/ejb/ProcessPaymentHome");

        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
        PortableRemoteObject.narrow
                    (ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentLocal process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);

        String ticketDescription = ticket.toString();

        TopicConnectionFactory factory = (TopicConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/TopicFactory");

        Topic topic = (Topic)
        jndiContext.lookup("java:comp/env/ejb/TicketTopic");

        TopicConnection connect = factory.createTopicConneciton();

        TopicSession session =
        connect.createTopicSession(true,0);


        TopicPublisher publisher = session.createPublisher(topic);

        TextMessage textMsg = new TextMessage(ticketDescription);
        publisher.publish(textMsg);
        connect.close();

        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

A lot of new code was needed in order to send a message. However, while it may look a little overwhelming at first, the basics of JMS are not all that complicated.

TopicConnectionFactory and Topic

In order to send a JMS message we need a connection to the JMS provider and a destination address for the messages. The connection to the JMS provider is made possible by a JMS connection factory; the destination address of the message is identified by a `Topic` object. Both the connection factory and the `Topic` object are obtained from the TravelAgent EJB's JNDI ENC.

```
TopicConnectionFactory factory = (TopicConnectionFactory)
jndiContext.lookup("java:comp/env/jms/TopicFactory");

Topic topic = (Topic)
jndiContext.lookup("java:comp/env/ejb/TicketTopic");
```

The `TopicConnectionFactory` in JMS is similar in function to the `DataSource` in JDBC. Just as the `DataSource` provides a JDBC connection to a database, the `TopicConnectionFactory` provides a JMS connection to a message router.

The `Topic` object itself represents a network independent destination to which the message will be addressed. In JMS, messages are sent to destinations—either topics or queues—instead of sending them directly to other applications. Destinations in JMS are analogous to e-mail lists or news groups; any application with the proper credentials can subscribe to any destination and send messages and receive messages from that destination. JMS decouples applications by allowing them to send messages to each other through a destination, which serves as virtual channel. This example uses a `Topic` type destination, but JMS also supports `Queue` type destinations. The difference between these types is explained in more detail later.

TopicConnection and TopicSession

The `TopicConnectionFactory` is used to create a `TopicConnection`, which is an actual connection to the JMS provider:

```
TopicConnection connect = factory.createTopicConneciton();

TopicSession session =
connect.createTopicSession(true,0);
```

Once a `TopicConnection` is obtained, it can be used to create a `TopicSession`. A `TopicSession` allows the Java developer to group the actions of sending and receiving messages. In most cases, you will only need a single `TopicSession`, but occasionally having more than one `TopicSession` object is helpful.

The `createTopicSession()` method is defined with two parameters:

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

These arguments are ignored at runtime because the EJB container manages the transaction and acknowledgment mode of any JMS resource obtained from the JNDI ENC. The specification recommends that developers use the arguments `true` for `transacted` and `0` for `acknowlegeMode`, but since they are supposed to be ignored, it should not matter what you use.

TopicPublisher

The `TopicSession` is used to create a `TopicPublisher`. The `TopicPublisher` is used to send messages from the TravelAgent EJB to the destination specified by the topic. Any JMS clients that subscribe or listen to that topic will receive a copy of the message:

```
TopicPublisher publisher = session.createPublisher(topic);

TextMessage textMsg = new TextMessage(ticketDescription);
publisher.publish(textMsg);
```

Message Types

In JMS, a message is a Java object with two parts: a header and a message body. The header is composed of delivery information and metadata, while the message body carries the application data, which can take several forms: text, serializable objects, byte streams, etc. The JMS API defines several message types (`TextMessage`, `MessageMap`, `ObjectMessage`, and others) and provides methods for delivering messages to, and receiving messages from, other applications.

For example, we can change the TravelAgent EJB so that it sends a `MapMessage` instead of a `TextMessage`:

```
TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
...
TopicPublisher publisher = session.createPublisher(topic);

MapMessage mapMsg = new MapMessage();
textMsg.setInt("CustomerID", ticket.customerID.intValue());
textMsg.setInt("CruiseID", ticket.cruiseID.intValue());
textMsg.setInt("CabinID", ticket.cabinID.intValue());
textMsg.setDouble("Price", ticket.price);

publisher.publish(mapMsg);
```

The attributes of the `MapMessage` (`CustomerID`, `CruiseID`, `CabinID`, and `Price`) can be accessed by name from those JMS clients that receive it.

As an alternative, The TravelAgent EJB could be modified to use the `ObjectMessage` type, which would allow us to send the entire `TicketDO` object as the message using Java serialization:

```
TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
...
TopicPublisher publisher = session.createPublisher(topic);

ObjectMessage objectMsg = new ObjectMessage();
ObjectMsg.setObject(ticket);
```

```
publisher.publish(objectMsg);
```

In addition to the `TextMessage`, `MapMessage` and `ObjectMessage`, JMS provides two other message types: `StreamMessage` and `BytesMessage`. `StreamMessage` can take as its payload the contents of an I/O stream. `BytesMessage` can take any array of bytes, which it treats as opaque data.

XML Deployment Descriptor

When a JMS resource is used, it must be declared in the bean's XML deployment descriptor, in a manner similar to the JDBC resource used by the Ship EJB in Chapter 10:

```
<enterprise-beans>
   <session>
      <ejb-name>TravelAgentBean</ejb-name>
      ...

     <resource-ref>
        <res-ref-name>jms/TopicFactory</res-ref-name>
        <res-type>javax.jms.TopicConnectionFactory</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
    <resource-ref>
         <res-ref-name>jdbc/titanDB</res-ref-name>
         <res-type>javax.sql.DataSource</res-type>
         <res-auth>Container</res-auth>
    </resource-ref>
    <resource-env-ref>
        <resource-env-ref-name>
            jms/TicketTopic
        </resource-env-ref-name>
        <resource-env-ref-type>
            javax.jms.Topic
        </resource-env-ref-type>
    </resource-env-ref>
  </session>
```

The `<resource-ref>` for the JMS `TopicConnectionFactory` is similar to the `<resource-ref>` declaration for the JDBC `DataSource`. The JNDI ENC name, interface type, and authorization protocol are declared. In addition to the `<resource-ref>`, the TravelAgent EJB must also declare the `<resource-env-ref>`, which lists any "administered objects" associated with a `<resource-ref>` entry. In this case, we declare the `Topic` used for sending a Ticket message. While the bean is under development, the `<resource-env-ref>` is only used for declaring JMS destinations. At deployment time the deployer will map the JMS `TopicConnectionFactory` and `Topic` declared by the `<resource-ref>` and `<resource-env-ref>` elements to a JMS factory and topic.

JMS Application Client

To get a better idea of how JMS is used, we can create a Java application whose sole purpose is receiving and processing reservation messages. We will develop a very simple JMS client that simply prints a description of each ticket as it receives the messages. We'll assume that the TravelAgent is using the TextMessage to send a description of the Ticket to the JMS clients. The following code shows how the JMS application client might look.

```
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.Topic;
import javax.jms.TopicSubscriber;
import javax.jms.JMSException;
import javax.naming.InitalContext;


public class JmsClient_1 extends javax.jms.MessageListener{

    public static void main(String [] args){

        if(args.length != 2)
            throw new Exception("Wrong number of arguments");

        new JmsClient_1(args[0], args[1]);

        while(true){Thread.sleep(10000);}

    }

    public JmsClient_1(String factoryName, String topicName)
    throws Exception{

        InitialContext jndiContext = getInitialContext()

        TopicConnectionFactory factory = (TopicConnectionFactory)
        jndiContext.lookup(factoryName);

        Topic topic = (Topic)
        jndiContext.lookup(topicName);

        TopicConnection connect = factory.createTopicConneciton();

        TopicSession session =
        connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

        TopicSubscriber subscriber = session.createSubscriber(topic);
```

```
        subscriber.setMessageListener(this);

        connect.start();
    }

    public void onMessage(Message message){
        try{

        TextMessage textMsg = (TextMessage)message;
        String text = textMsg.getText();
        System.out.println("\n RESERVATION RECIEVED:\n"+text);

        }catch(JMSException jmsE){
            jmsE.printStackTrace();
        }
    }

    public static InitialContext getInitialContext(){
        // create vendor speicific JNDI Context here
    }
}
```

The constructor of `JmsClient_1` obtains the `TopicConnectionFactory` and `Topic` from the JNDI `InitialContext`. This context is created with vendor-specific properties so that the client can connect to the same JMS provider as the one used by the TravelAgent EJB. For example, the `getInitialContext()` method for the Weblogic Application server would be coded as follows:

```
public static InitialContext getInitialContext(){
    Properties env =new Properties();
    env.put(Context.SECURITY_PRINCIPAL, "guest");
    env.put(Context.SECURITY_CREDENTIALS, "guest");
    env.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
    env.put("Context.PROVIDER_URL, "t3://localhost:7001");
    return new InitialContext(env);
}
```

Once the client has the `TopicConnectionFactory` and `Topic`, it creates a `TopicConnection` and a `TopicSession` in the same way as the TravelAgent EJB. The big difference comes when the `TopicSession` object is used to create a `TopicSubscriber` instead of a `TopicPublisher`. The `TopicSubscriber` is designed specifically to process incoming messages that are published to its specified `Topic`.

```
TopicSession session =
connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

TopicSubscriber subscriber = session.createSubscriber(topic);
```

```
subscriber.setMessageListener(this);

connect.start();
```

The `TopicSubscriber` can receive messages directly, or it can delegate the processing of the messages to a `javax.jms.MessageListener`. We chose to have `JmsClient_1` implement the `MessageListener` interface so that it can process the messages itself. `MessageListener` objects implement a single method, `onMessage()`, which is invoked every time a new message is sent to the subscriber's topic. In this case, every time the TravelAgent EJB sends a reservation message to the topic, the JMS client will have its `onMessage()` method invoked so that it can receive a copy of the message and process it.

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

  TextMessage textMsg = (TextMessage)message;
  String text = textMsg.getText();
  System.out.println("\n RESERVATION RECIEVED:\n"+text);

  }catch(JMSException jmsE){
      jmsE.printStackTrace();
  }
}
```

📖 Exercise 13.1, JMS and the TravelAgent EJB

JMS is Asynchronous

One of the principal advantages of JMS messaging is that it's asynchronous. In other words, a JMS client can send a message without having to wait for a reply. Contrast this flexibility with the synchronous messaging of Java RMI. RMI is an excellent choice for assembling transactional components, but is too restrictive for some uses. Each time a client invokes a bean's method it blocks until the method completes execution. This lock-step processing makes the client dependent on the availability of the EJB server, resulting in a tight coupling between the client and enterprise bean.

In JMS, a client sends messages asynchronously to a topic, to which other JMS clients subscribe or listen. When a JMS client sends a message, it doesn't wait for a reply; it sends the message to a router, which is responsible for forwarding it to other clients. Clients sending messages are decoupled from the clients receiving them; senders are not dependent on the availability of receivers.

The limitations of RMI make JMS an attractive alternative for communicating with other applications. Using the standard JNDI environment-naming context, an enterprise bean can obtain a JMS connection to a JMS provider and use it to

deliver asynchronous messages to other Java applications. As an example, a TravelAgent session bean can use JMS to notify other applications that an order has been processed.

Figure 13-1: Using JMS with the TravelAgent EJB

In this case, the applications receiving JMS messages from the TravelAgent EJB may be message-driven beans, other Java applications in the enterprise, or applications in other organizations that benefit from being notified that an order has been processed. Examples might include business partners who share customer information or an internal marketing application that adds customers to a catalog mailing list.

JMS enables the enterprise bean to send messages without blocking. The enterprise bean doesn't know who will receive the message, because it delivers the message to a virtual channel (destination) and not directly to another application. Applications can choose to subscribe to that virtual channel and receive notification of new reservations.

An interesting aspect of enterprise messaging in general and JMS in particular, is that the de-coupled asynchronous nature of the technology means that transactions and security contexts of the sender are not propagated to the receiver of the message. The sender can be authenticated against the JMS provider (message router) but it doesn't propagate its security context. For example, when the TravelAgent EJB sends the ticket message, it may be authenticated by the JMS provider but it won't propagate the security context. When a JMS client receives the message from the TravelAgent EJB, it will have no idea about the security context under which it was sent. This is how it should be; the sender and receiver often operate in different environments with different security domains.

Similarly, transactions are never propagated from the sender to the receiver. For one thing, the sender has no idea who the receivers of the message will be. If the message is sent to a topic there could be one receiver or thousands, managing a distributed transaction under such ambiguous circumstances is not tenable. In addition, the clients receiving the message may not get it for a long time after its sent. Clients may be down or otherwise unable to receive messages; one key strength of JMS is that it allows senders and receivers to be temporally de-coupled. Transactions are designed to be executed very quickly because they lock of resources; the possibility of a long transaction with an unpredictable end is also not tenable.

A JMS client can, however, have a distributed transaction with the JMS provider so that it manage the send or receive operation in the context of a transaction. For example, if the TravelAgent EJB's transaction fails or any reason, the JMS provider will discard the ticket message sent by the TravelAgent EJB. Transactions and JMS are covered in more detail in Chapter 14.

JMS Messaging Models: Publish/Subscribe and Point-to-Point

JMS provides two types of messaging models: publish-and-subscribe and point-to-point queuing. The JMS specification refers to these as *messaging domains*. In JMS terminology, publish-and-subscribe and point-to-point are frequently shortened to pub/sub and p2p (or PTP) respectively. This chapter uses both the long and short forms throughout.

In the simplest sense, publish-and-subscribe is intended for a one-to-many broadcast of messages, while point-to-point is intended for one-to-one delivery of messages (See Figure 13-1).

[FIGURE use figure 1-4 from JMS book]

Figure 13-2: JMS Messaging Domains

A JMS client that produces a message is called a *producer*, while a JMS client that receives a message is called a *consumer*. A JMS client can be both a producer and a consumer. When we use the term *consumer* or *producer*, we mean a JMS client that consumes messages or produces messages, respectively. We use this terminology throughout the book.

Publish and Subscribe

In pub/sub, one producer can send a message to many consumers through a virtual channel called a *topic*. Consumers, which receive messages, can choose to *subscribe* to a topic. Any messages addressed to a topic are delivered to all the topic's consumers. Every consumer receives a copy of each message. The pub/sub messaging model is by and large a push-based model, where messages are automatically broadcast to consumers without them having to request or poll the topic for new messages.

In the pub/sub messaging model, the producer sending the message is not dependent on the consumers receiving the message. Optionally, JMS clients that use pub/sub can establish durable subscriptions that allow consumers to disconnect and later reconnect and collect messages that were published while they were disconnected.

The TravelAgent EJB in this chapter uses the pub/sub programming model with a `Topic` as a destination. The `TopicPublisher` sends messages from the TravelAgent EJB to the `Topic`.

Point To Point

The point-to-point messaging model allows JMS clients to send and receive messages both synchronously and asynchronously via virtual channels known

as *queues*. The p2p messaging model has traditionally been a pull- or polling-based model, where messages are requested from the queue instead of being pushed to the client automatically[1].

A queue may have multiple receivers, but only one receiver may consume each message. As shown in Figure 13-1, the JMS provider will take care of doling out the work, ensuring that each message is consumed once by the next available receiver in the group. The JMS specification does not dictate the rules for distributing messages among multiple receivers, although some JMS vendors have chosen to implement this as a load balancing capability.

The messaging API for p2p is very similar to that used for pub/sub. The following shows how the TravelAgent EJB could be modified to use the `Queue`-based p2p API instead of the `Topic`-based pub/sub model used in the earlier example.

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {
        ...

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);

        String ticketDescription = ticket.toString();

        QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/QueueFactory");

        Queue queue = (Queue)
        jndiContext.lookup("java:comp/env/ejb/TicketQueue");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
        connect.createQueueSession(true,0);

        QueueSender sender = session.createSender(queue);

        TextMessage textMsg = new TextMessage(ticketDescription);
        sender.send(textMsg);
        connect.close();

        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
```

[1] In JMS, an option allows p2p clients to use a push model similar to pub/sub.

```
    }
}
```

Which messaging model should you use?

The rationale behind the two models lies in the origin of the JMS specification. JMS started out as a way of providing a common API for accessing existing messaging systems. At the time of its conception, some messaging vendors had a p2p model, and some had a pub/sub model. Hence JMS needed to provide an API for both models to gain wide industry support. The JMS 1.0.2 specification does not require a JMS provider to support both models, although most JMS vendors do.

Almost anything that can be done with the pub/sub model can be done with point-to-point, and vice versa. An analogy can be drawn to developers' programming language preferences. In theory, any application that can be written with Pascal can also be written with C. Anything that can be written in C++ can also be written in Java. In some cases it comes down to a matter of preference, or which model you are already familiar with.

In most cases, the decision about which model to use depends on the distinct merits of each model. With pub/sub, any number of subscribers can be listening on a topic, all receiving copies of the same message. The publisher may not care if everybody is listening, or even if nobody is listening. For example, consider a publisher that broadcasts stock quotes. If any particular subscriber is not currently connected and misses out on a great quote, the publisher is not concerned. In contrast, a point-to-point session is likely to be intended for a one-on-one conversation with a specific application at the other end. In this scenario, every message really matters.

The range and variety of the data that the messages represent can be a factor as well. Using pub/sub, messages are dispatched to the consumers based on filtering that is provided through the use of specific topics. Even when messaging is being used to establish a one-on-one conversation with another known application, it can be advantageous to use pub/sub with multiple topics to segregate different kinds of messages. Each kind of message can be dealt with separately through its own unique consumer and `onMessage()` handler.

Point-to-point is more convenient when you want one receiver to process any given message once. This is perhaps the most critical difference between the two models: point-to-point guarantees that only one consumer processes a given message. This is extremely important when messages need to be processed separately but in tandem.

Entity and session beans shouldn't receive messages

`JmsClient_1` was designed to consume messages produced by the TravelAgent EJB. Can another entity or session bean receive those message also? The answer is yes, but it's a really bad idea.

Entity and session beans respond to Java RMI calls from EJB clients and cannot be programmed to respond to JMS messages as do message-driven beans. That means it's impossible to write a session or entity bean that will be driven by incoming messages. The inability to make EJBs respond to JMS messages was why message-driven beans were introduced in EJB 2.0. Message-driven beans are designed to subscribe or listen to topics and queues and to process messages delivered to those destinations. They fill an important niche; we'll learn more about how to program them in the next section.

It is, however, possible to develop an entity or session bean that can consume a JMS message from a business method, but the method must be called by an EJB client first. For example, when the business method on the Hypothetical EJB is called, it sets up a JMS session and then attempts to read a message from a queue.

```
public class HypotheticalBean implements javax.ejb.SessionBean {
    InitialContext jndiContext;

    public String businessMethod( ){

        try{

        QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/QueueFactory");

        Queue topic = (Queue)
        jndiContext.lookup("java:comp/env/jms/Queue");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
        connect.createQueueSession(true,0);

        QueueReceiver receiver = session.createReciever(queue);

        TextMessage textMsg = (TextMessage)reciever.receive();

        connect.close();

        return textMsg.getText();

       }catch(Exception e){
           throws new EJBException(e);
```

```
        }

    }
    …
}
```

The `QueueReceiver`, which is a message consumer, is used to proactively fetch a message from the queue. While this has been programmed correctly, it is a dangerous operation because a call to the `QueueReceiver.receive()` method blocks the thread until a message becomes available. If a message is never delivered to the receiver's queue, the thread will block indefinitely! In other words, if no one every sends a message to the queue we are listening too, then the `QueueReceiver` will just sit their waiting forever.

To be fair, there are other `receive()` methods that are less dangerous. For example, `receive(long timeout)` allows you to specify a time after which the `QueueReceiver` should stop blocking the thread and give up waiting for a message. There is also `receiveNoWait()`, which checks for a message and returns `null` if there are none waiting, thus avoiding a prolonged thread block.

While the alternative `receive()` methods are much safer, this is still a dangerous operation to perform. There is no guarantee that the less risky `receive()` methods will perform as expected, and the risk of programmer error (e.g., using the wrong `receive()` method) is too risky. Besides, the message-driven bean provides you with a powerful and simple enterprise bean that is especially designed to consume JMS messages. This book recommends that you do not attempt to consume messages from entity or session beans.

Learning more about JMS

JMS (and enterprise messaging in general) represents a powerful paradigm in distributed computing. In my opinion, the Java Message Service is as important as Enterprise JavaBeans itself, and should be well understood before it's used in development.

While this chapter has provided a brief overview of JMS, we have only been able to present you with enough material to prepare you for the discussion of message-driven beans in the next section. To understand JMS and how it is used, you will need to study it independently. For a detailed treatment of JMS, see Java Message Service (O'Reilly, 2000). Taking the time to learn JMS is well worth the effort.

Message-Driven Beans

Message-driven beans (MDBs) are stateless, server-side, transaction-aware components for processing asynchronous JMS messages. Newly introduced in EJB 2.0, message-driven beans process messages delivered via the Java Message Service.

Message-driven beans can receive JMS messages and process them using the same robust component-based infrastructure that session and entity beans enjoy. While a message-driven bean is responsible for processing messages, its container takes care of automatically managing the component's entire environment including transactions, security, resources, concurrency, message acknowledgment, etc.

One of the most important aspects of message-driven beans is that they can consume and process messages concurrently. This capability provides a significant advantage over traditional JMS clients, which must be custom-built to manage resources, transactions, and security in a multi-threaded environment. The message-driven bean containers provided by EJB manage concurrency automatically, so the bean developer can focus on the business logic of processing the messages. The MDB can receive hundreds of JMS messages from many different applications and process them all at the same time, because the container can have many instances of any MDB executing concurrently.

The container also ensures that any operations on resources or enterprise beans accessed by an MDB are executed in the same transaction and that the security identity associated with the MDB is propagated to the resources and enterprise beans it accesses.

A message-driven bean is a complete enterprise bean, just like a session or entity bean, but there are some important differences. While a message-driven bean has a bean class and XML deployment descriptor, it does not have remote or home interfaces. These interfaces are absent because the message-driven bean is not accessible via the Java RMI API; it responds only to asynchronous messages.

ReservationProcessor EJB

The ReservationProcessor EJB is a message-driven bean that receives JMS messages notifying it of new reservations. The ReservationProcessor is an automated version of the TravelAgent EJB that processes reservations sent via JMS by other travel organizations. It requires no human intervention; it's completely automated.

The JMS messages that notify the ReservationProcessor EJB of new reservations might come from another application in the enterprise or an application in some other organization. When the ReservationProcessor EJB receives a message, it creates a new Reservation EJB (adding it to the database), processes the payment using the ProcessPayment EJB, and sends out a ticket.

[FIGURE}

Figure 13-3: The ReservationProcessor EJB processing reservations

ReservationProcessorBean

Here is a partial definition of the `ReservationProcessorBean` class. Some methods are left empty; they will be filled in later. Notice that the `onMessage()` method contains the business logic of the bean class; it is similar to the business logic developed in the `bookPassage()` method of the TravelAgent EJB in Chapter 12.

```
package com.titan.reservationprocessor;

import javax.jms.Message;
import javax.jms.MapMessage;
import com.titan.customer.*;
import com.titan.cruise.*;
import com.titan.cabin.*;
import com.titan.reservation.*;
import com.titan.processpayment.*;


public class ReservationProcessorBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener {

    MessageDrivenContext ejbContext;
    Context jndiContext;

    public void setMessageDrivenContext(MessageDrivenContext mdc){
        ejbContext = mdc;
        try{
           jndiContext = new InitialContext();
        }catch(NamingException ne){
           throw new EJBException(ne);
        }
    }

    public void ejbCreate(){}

    public void onMessage(Message message) {
        try {
            MapMessage reservationMsg = (MapMessage)message;
```

```
        Integer customerPk = (Integer)
                reservationMsg.getObject("CustomerID");
        Integer cruisePk =   (Integer)
                reservationMsg.getObject("CruiseID");
        Integer cabinPk =    (Integer)
                reservationMsg.getObject("CabinID");

        double price = reservationMsg.getDouble("Price");

        CreditCardDO card = (CreditCardDO)
                reservationMsg.getObject("CreditCard");

        CustomerLocal customer = getCustomer(customerPk);
        CruiseLocal cruise = getCruise(cruisePk);
        CabinLocal cabin = getCabin(cabinPk);

        ReservationHomeLocal resHome = (ReservationHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/ReservationHome");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);

        Object ref = jndiContext.lookup
                    ("java:comp/env/ejb/ProcessPaymentHome");

        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
        PortableRemoteObject.narrow
                    (ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentLocal process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
            new TicketDO(customer,cruise,cabin,price);


        deliverTicket(reservationMsg, ticket);

    } catch(Exception e) {
        throw new EJBException(e);
    }
}

public void deliverTicket(MapMessage reservationMsg){

    // create a ticket and send it to the proper destination
}
public CustomerRemote getCustomer(Integer key)
throws NamingException, ObjectNotFoundException{
```

```
        // get a remote reference to the Customer EJB
    }
    public CruiseLocal getCruise(Integer key)
    throws NamingException, ObjectNotFoundException{
        // get a local reference to the Cruise EJB
    }
    public CabinLocal getCabin(Integer key)
    throws NamingException, ObjectNotFoundException{
        // get a local reference to the Cabin EJB
    }

    public void ejbRemove(){
        try{
           jndiContext.close();
           ejbContext = null;
        }catch(NamingException ne){ /* do nothing */ }
    }
}
```

MessageDrivenBean Interface

The message-driven bean class is required to implement the `javax.ejb.MessageDrivenBean` interface, which defines callback methods similar to those in entity and session beans. Here is the definition of the `MessageDrivenBean` interface.

```
package javax.ejb;

public interface MessageDrivenBean extends javax.ejb.EnterpriseBean {
    public void setMessageDrivenContext(MessageDrivenContext context)
    throws EJBException;
    public void ejbRemove( ) throws EJBException;
}
```

The `setMessageDrivenContext()` method is called at the beginning of the MDB's life cycle and provides the MDB instance with a reference to its `MessageDrivenContext`:

```
    MessageDrivenContext ejbContext;
    Context jndiContext;

    public void setMessageDrivenContext(MessageDrivenContext mdc){
        ejbContext = mdc;
        try{
           jndiContext = new InitialContext();
        }catch(NamingException ne){
           throw new EJBException(ne);
        }
    }
```

The `setMessageDrivenContext()` method in the `ReservationProcessorBean` class sets the `ejbContext` instance field

to the MessageDrivenContext, which was passed into the method. It also obtains a reference to the JNDI ENC, which it stores in the jndiContext. MDBs may have instance fields that are similar to a stateless session beans instance fields. These instance fields are carried with the MDB instance for its life time and may be reused every time it processes a new message. Unlike stateful session beans, MDBs do not have "conversational" state and are not specific to a single JMS client. MDB instances are used to processes messages from many different JMS clients, and are tied to a specific topic or queue to which they subscribe or listen, not to a specific JMS client. They are stateless in the same way that stateless session beans are stateless.

ejbRemove() provides the MDB instance an opportunity to clean up any resources it stores in its instance fields. In this case, we use it to close the JNDI Context and set the ejbContext field to null. These operations are not absolutely necessary, but they illustrate the kind of operation that an ejbRemove() method might do. Note that ejbRemove() is called at the end of the MDB's life cycle, before it is garbage collected. It may not be called if the EJB server hosting the MDB fails or if an EJBException is thrown by the MDB instance in one its other methods. When an EJBException is thrown by any method in the MDB instance, the instance is immediately removed from memory and the transaction is rolled back.

MessageDrivenContext

The MessageDrivenContext simply extends the EJBContext and doesn't add any new methods. The EJBContext is defined as:

```
package javax.ejb;
public interface EJBContext {

   // transaction methods
   public javax.transaction.UserTransaction getUserTransaction()
   throws java.lang.IllegalStateException;
   public boolean getRollbackOnly()
       throws java.lang.IllegalStateException;
   public void setRollbackOnly()
       throws java.lang.IllegalStateException;

   // EJB home methods
   public EJBHome getEJBHome();
   public EJBLocalHome getEJBLocalHome();

   // security methods
   public java.security.Principal getCallerPrincipal();
   public boolean isCallerInRole(java.lang.String roleName);

   // deprecated methods
   public java.security.Identity getCallerIdentity();
   public boolean isCallerInRole(java.security.Identity role);
```

```
    public java.util.Properties getEnvironment();

}
```

Only the transactional methods that the `MessageDrivenContext` inherits from `EJBContext` are available to message-driven beans. The home methods (`getEJBHome()` and `getEJBLocalHome()`) throw a `RuntimeException` if invoked, because MDBs do not have home interfaces or EJB home objects. The security methods (`getCallerPrincipal()` and `isCallerInRole()`) also throw a `RuntimeException` if invoked on a `MessageDrivenContext`. When an MDB services a JMS message there is no "caller" so a security context doesn't exist that can be obtained from the caller. Remember that JMS is asynchronous, and doesn't propagate the sender's security context to the receiver—that wouldn't make sense since senders and receivers tend to operate in different environments.

MDBs usually execute in a container-initiated or bean-initiated transaction, so the transaction methods allow the MDB to manage its context. The transaction context is not propagated from the JMS sender, but is a transaction that is either initiated by the container or by the bean explicitly using `javax.jta.UserTransaction`. The transaction methods in the `EJBContext` are explained in more detail in Chapter 14.

Message-driven beans also have access to their own JNDI environment naming context (ENC) which provides the MDB instance access to environment entries, other enterprise beans, and resources. The ReservationProcessor EJB takes advantage of the JNDI ENC to obtain references to the Customer, Cruise, Cabin, Reservation, and ProcessPayment EJB as well as a JMS `QueueConnectionFactory` and `Queue` for sending out tickets.

MessageListener Interface

In addition to the `MessageDrivenBean` interface, MDBs implement the `javax.jms.MessageListener` interface, which defines the `onMessage()` method; bean developers implement this method to process JMS messages received by a bean. It's in this `onMessage()` method that the bean processes the JMS message.

```
package javax.jms;
public interface MessageListener {
    public void onMessage(Message message);
}
```

It's interesting to consider why the MDB implements the `MessageListener` interface separately from the `MessageDrivenBean` interface. Why not just put the `onMessage()` method, `MessageListener` only method, in the `MessageDrivenBean` interface so that there is only interface for the MDB class to implement? This was the solution taken by an early proposed version of

EJB 2.0. However, it was quickly realized that message-driven beans could, in the future, process messages from other types of systems, not just JMS. To make the MDB open to other messaging systems, it was decided that it should implement the `javax.jms.MessageListener` interface separately, thus separating the concept of the message-driven bean from the types of messages it can process. In a future version of the specification other types of MDB might be available for things like SMTP (e-mail) or JAXM (Java API for XML Messaging) for ebXML. Other technologies will use different methods rather then `onMessage()` which is specific to JMS.

The onMessage() method: Workflow and Integration for B2B

The `onMessage()` method is where all the business logic goes. As messages arrive they are passed to the MDB using its `onMessage()` method by the container. When the method returns, the MDB is ready to process a new message.

In the ReservationProcessor EJB, the `onMessage()` method extracts information about a reservation from a `MapMessage` and uses that information to create a reservation in the system:

```
public void onMessage(Message message) {
    try {
        MapMessage reservationMsg = (MapMessage)message;

        Integer customerPk = (Integer)
                reservationMsg.getObject("CustomerID");
        Integer cruisePk =   (Integer)
                reservationMsg.getObject("CruiseID");
        Integer cabinPk =    (Integer)
                reservationMsg.getObject("CabinID");

        double price = reservationMsg.getDouble("Price");

        CreditCardDO card = (CreditCardDO)
                reservationMsg.getObject("CreditCard");
```

JMS is frequently used as an integration point for business-to-business applications, so it's easy to imagine the reservation message coming from one of Titan's business partners, perhaps a 3rd party processor or branch travel agency.

The ReservationProcessor needs to access the Customer, Cruise and Cabin EJBs in order to process the reservation. The MapMessage contains the primary keys for these entities; the ReservationProcessor EJB uses helper methods (`getCustomer()`, `getCruise()`, and `getCabin()`) methods to look up the entity beans and obtain EJB object references to them:

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

```
    CustomerLocal customer = getCustomer(customerPk);
    CruiseLocal cruise = getCruise(cruisePk);
    CabinLocal cabin = getCabin(cabinPk);
    ...
}

public CustomerLocal getCustomer(Integer key)
throws NamingException, ObjectNotFoundException{

    CustomerHomeLocal home = (CustomerHomeLocal)
    jndiContext.lookup("java:comp/env/ejb/CustomerHome");
    CustomerLocal customer = home.findByPrimaryKey(key);
    return customer;
}
public CruiseLocal getCruise(Integer key)
throws NamingException, ObjectNotFoundException{

    CruiseHomeLocal home = (CruiseHomeLocal)
    jndiContext.lookup("java:comp/env/ejb/CruiseHome");
    CruiseLocal cruise = home.findByPrimaryKey(key);
    return cruise;
}
public CabinLocal getCabin(Integer key)
throws NamingException, ObjectNotFoundException{

    CabinHomeLocal home = (CabinHomeLocal)
    jndiContext.lookup("java:comp/env/ejb/CabinHome");
    CabinLocal cabin = home.findByPrimaryKey(key);
    return cabin;
}
```

Once the information is extracted from the `MapMessage`, it is used to create a reservation and process the payment. This is basically the same workflow that was used by the TravelAgent EJB in Chapter 12. A Reservation EJB is created that represents the reservation itself, and a ProcessPayment EJB is created to process the credit card payment.

```
ReservationHomeLocal resHome = (ReservationHomeLocal)
    jndiContext.lookup("java:comp/env/ejb/ReservationHome");

ReservationLocal reservation =
    resHome.create(customer, cruise, cabin, price);

Object ref = jndiContext.lookup
            ("java:comp/env/ejb/ProcessPaymentHome");

ProcessPaymentHomeRemote ppHome =
(ProcessPaymentHomeRemote)
PortableRemoteObject.narrow
```

```
            (ref, ProcessPaymentHomeRemote.class);

ProcessPaymentLocal process = ppHome.create();
process.byCredit(customer, card, price);

TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);

deliverTicket(reservationMsg, ticket);
```

This illustrates that the MDB can access any other entity or session bean, and use those other beans to complete a task, just like session beans. In this way, the MDB fulfills its role as an integration point in B2B application scenarios. MDB can manage a process and interact with other beans as well as resources. For example, it's commonplace for an MDB to use JDBC to access a database based on the contents of the message it's processing.

Sending Messages from a Message-Driven Bean

MDB can also send messages using JMS. The `deliverTicket()` method sends the Ticket information to a destination defined by the sending JMS client:

```
public void deliverTicket(MapMessage reservationMsg, TicketDO ticket)
throws NamingException, JMSException{

    Queue queue = (Queue)reservationMsg.getJMSReplyTo();

    QueueConnectionFactory factory = (QueueConnectionFactory)
    jndiContext.lookup("java:comp/env/jms/QueueFactory");

    QueueConnection connect = factory.createQueueConneciton();

    QueueSession session =
    connect.createQueueSession(true,0);

    QueueSender sender = session.createSender(queue);

    ObjectMessage message = new ObjectMessage(ticket);

    sender.send(message);

    connect.close();

}
```

As stated earlier, every message type has two parts: a message header and a message body (a.k.a. payload). The message header contains routing information, and may also have properties for message filtering and other attributes, including a *JMSReplyTo* attribute. When a JMS client sends a message, it may set the *JMSReplyTo* attribute to be any destination accessible to its JMS provider. In the case of the reservation message, the sender set the

JMSReplyTo attribute to the `Queue` to which the resulting `Ticket` should be sent. Another application can access this `Queue` to read tickets and distribute them to customers, or store the information in the sender's database.

The JMSReplyTo address can also be used to report business errors that occur while processing the message. For example, if the Cabin is already reserved, the ReservationProcessor EJB might send an error message to the *JMSReplyTo* queue explaining that the reservation could not be processed. Including this type of error handling is left as an exercise for the reader.

XML Deployment Descriptor

MDBs have XML deployment descriptors, just like entity and session beans. They can be deployed alone or, more often than not, deployed together with other enterprise beans. For example, the ReservationProcessor EJB would have to be deployed in the same JAR using the same XML deployment descriptor as the Customer, Cruise, and Cabin beans if it's going to use their local interfaces.

Here's how the XML deployment descriptor that defines the ReservationProcessor EJB. This deployment descriptor also defines the Customer, Cruise, Cabin, and other beans, but these are left out for brevity.

```
<enterprise-beans>
    ...
    <message-driven>
        <ejb-name>ReservationProcessorEJB</ejb-name>
        <ejb-class>
        com.titan.reservationprocessor.ReservationProcessorBean
        </ejb-class>
        <transaction-type>Container</transaction-type>
        <message-selector>MessageFormat = "Version 2.3"</message-selector>
        <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
        <message-driven-destination>
            <destination-type>javax.jms.Queue</destination-type>
        </message-driven-destination>
        <ejb-ref>
            <ejb-ref-name>ejb/ProcessPaymentHome</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>
                com.titan.processpayment.ProcessPaymentHomeRemote
            </home>
            <remote>
                com.titan.processpayment.ProcessPaymentRemote
            </remote>
        </ejb-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CustomerHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>
```

```
                com.titan.customer.CustomerHomeLocal
            </local-home>
            <local>com.titan.customer.CustomerLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CruiseHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>
                com.titan.cruise.CruiseHomeLocal
            </local-home>
            <local>com.titan.cruise.CruiseLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CabinHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>
                com.titan.cabin.CabinHomeLocal
            </local-home>
            <local>com.titan.cabin.CabinLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ReservationHome</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>
                com.titan.reservation.ReservationHomeLocal
            </local-home>
            <local>com.titan.reservation.ReservationLocal</local>
        </ejb-local-ref>
        <security-identity>
            <run-as>MANAGER</run-as>
        </security-identity>
        <resource-ref>
            <res-ref-name>jms/QueueFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </message-driven>
    ...
<enterprise-beans>
```

An MDB is declared in a `<message-driven>` element within the `<enterprise-beans>` element, alongside `<session>` and `<entity>` beans. Similar to `<session>` bean types, it defines an `<ejb-name>`, `<ejb-class>` and `<transaction-type>`, but it does not define component interfaces (local or remote). MDBs don't have component interfaces, so these definitions aren't needed.

message-selector

An MDB can also declare a `<message-selector>` element, which is unique to message-driven beans.

```
<message-selector>MessageFormat = "Version 3.4"</message-selector>
```

Message selectors allow an MDB to be more selective about the messages it receives from a particular topic or queue. Message selectors use `Message` properties as criteria in conditional expressions[2]. These conditional expressions use boolean logic to declare which messages should be delivered to a client.

Message properties, upon which message selectors are based, are additional headers that can be assigned to a message. They give the application developer or JMS vendor the ability to attach more information to a message. The `Message` interface provides several accessor and mutator methods for reading and writing properties. Properties can have a `String` value, or one of several primitive values (`boolean`, `byte`, `short`, `int`, `long`, `float`, `double`). The naming of properties, together with their values and conversion rules, is strictly defined by JMS.

The ReservationProcessor uses a message selector filter to select messages that meet a specific format. In this case the format is "Version 2.3"; this is a string that Titan uses to identify messages of type `MapMessage` and contain the name values `CustomerID`, `CruiseID`, `CabinID`, `CreditCard`, and `Price`. In other words, by specifying a "MessageFormat" on every reservation message, we can write MDBs that are designed to process that type of message. If a new business partner needs to use a different type of Message object, we only need a new message version and an MDB to process it.

This is how a JMS producer would go about setting a `MessageFormat` property on a Message:

```
Message message = new MapMessage();
message.setPropery("MessageFormat"," Version 3.4");

// set the reservation named values

sender.send(message);
```

The message selectors are based on a subset of the SQL-92 conditional expression syntax that is used in the WHERE clauses of SQL statements. They can become fairly complex, including the use of literal values, boolean expressions, unary operators, etc.

---

[2] Message selectors are also based on message headers, which is outside the scope of this chapter.

Message Selector Examples

Here are three complex selectors used in hypothetical environments. Although you will have to use your imagination a little, the purpose of these examples is to convey the power of the message selectors. When a selector is declared, the identifier always refers to a property name or JMS header name. For example, the selector "`UserName !='William'`" assumes that there is a property in the message named `UserName`, which can be compared to the value `'William'`.

Managing claims in an HMO

Due to some fraudulent claims, an automatic process is implemented using MDBs that will audit all claims submitted by patients who are employees of the ACME manufacturing company with visits to chiropractors, psychologists, and dermatologists:

```
<message-selector>
<![CDATA[
PhysicianType IN ('Chiropractic','Psychologists','Dermatologist') AND
PatientGroupID LIKE 'ACME%'
]]>
</message-selector>
```

> MDB `<message-selector>` statements are declared in XML deployment descriptors. XML assigns a variety of characters like the greater than ('>') and less than ('<') special meaning, so using these symbols in the `<message-selector>` statements will cause parsing errors unless CDATA sections are used. This is the same reason CDATA section were needed in EJB QL `<ejb-ql>` statements as explained in Chapter 8.

Notification of certain bids on inventory

A supplier wants notification of requests-for-bids on specific inventory items at specific quantities:

```
<message-selector>
<![CDATA[
 InventoryID ='S93740283-02'AND Quantity BETWEEN 1000 AND 13000 ";
]]>
</message-selector>
```

Selecting recipients for a catalog mailing

An online retailer wants to deliver a special catalog to any customer that orders more then $500.00 worth of merchandise where the average price per item ordered is greater than $75.00 and the customer resides in one several states. The retailer creates a special application that subscribes to the order processing topic and

processes catalog deliveries for only those customers that meet the defined criteria:

```
<message-selector>
<![CDATA[
   TotalCharge >500.00 AND ((TotalCharge /ItemCount)>=75.00)
   AND State IN ('MN','WI','MI','OH')";
]]>
</message-selector>
```

acknowledge-mode

JMS has the concept of acknowledgment, which means that the JMS client notifies the JMS provider (message router) that a message was received. In EJB, it's the MDB container's responsibility to send acknowledgements to the JMS provider when it receives a message. To acknowledge a message is to tell the JMS provider that MDB container has received the message and processed it using an MDB instance. Without an acknowledgement, the JMS provider will not know whether the MDB container has received the message, so it will try to redeliver it. This can cause problems. For example, once we have processed a reservation message using the ReservationProcessor EJB, we don't want to receive the same message again.

When transactions are involved, the acknowledgment mode set by the bean provider is ignored. In this case, the acknowledgment is performed within the context of the transaction. If the transaction succeeds, the message is acknowledged. If the transaction fails, the message is not acknowledged. So if the MDB is using container-managed transactions, as it will in most cases, the acknowledgment mode is ignored by the MDB container. When using container-managed transactions with a `Required` transaction attribute, the `<acknowledge-mode>` is usually not specified; we included it in the deployment descriptor for the sake of discussion.

```
<acknowledge-mode>Auto-acknowledge</acknowledge-mode>
```

When the MDB executes with bean-managed transactions, or with the container-managed transaction attribute `NotSupported` (see Chapter 14), then the value of `<acknowledge-mode>` becomes important.

Two values can be specified for `<acknowledge-mode>`: `Auto-acknowledge` and `Dups-ok-acknowledge`. The first tells the container that it should send an acknowledgement to the JMS provider soon after the message is given to an MDB instance to process. The `Dups-ok-acknowledge` tells the container that it doesn't have to send the acknowledgement immediately; any time after the message is given to the MDB instance will be fine. With `Dups-ok-acknowledge`, it's possible for the MDB container to delay acknowledgement so long that the JMS provider assumes that the message was not received and so sends a "duplicate" message.

Obviously, with `Dups-ok-acknowledge`, your MDBs must be able to handle duplicate messages correctly.

`Auto-acknowledge` duplicate messages because the acknowledgement is sent immediately. Therefore, the JMS provider won't send a duplicate. In most cases an MDB will want to use `Auto-acknowledge`, to avoid processing the same message twice. `Dups-ok-acknowledge` exists because it may allow a JMS provider to optimize its use of the network. In practice, the overhead of an acknowledgment is so small, and the frequency of communication between the MDB container and JMS provider is so high, that `Dups-ok-acknowledge` doesn't have a big impact of performance.

message-driven-destination

The `<message-driven-destination>` element designates the type of destination that the MDB is subscribed to or listens to. The allowed values for this element are `javax.jms.Queue` and `javax.jms.Topic`. In the case of the ReservationProcessor EJB, this value is set to `javax.jms.Queue` meaning that the MDB is getting its messages via the p2p messaging model from a `Queue`.

```
<message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
</message-driven-destination>
```

When the MDB is deployed, the deployer will map the MDB so that it listens to a real `Queue` on the network.

When the `<destination-type>` is a `javax.jms.Topic`, the `<subscription-durability>` element must be declared with either `Durable` or `NonDurable` as its value.

```
<message-driven-destination>
    <destination-type>javax.jms.Topic</destination-type>
    <subscription-durability>Durable</subscription-durability>
</message-driven-destination>
```

The `<subscription-durability>` element determines whether or not the MDB's subscription to the `Topic` is durable. A `Durable` subscription outlasts an MDB's connection to the JMS provider. So if the EJB server suffers a partial failure, is shut down, or is otherwise disconnected from the JMS provider, the messages that it would have received will not be lost. While a `Durable` MDB is disconnected from the JMS provider, it is the responsibility of the JMS provider to store any messages the subscriber misses. When the `Durable` MDB reconnects to the JMS provider, the JMS provider sends it all the unexpired messages that accumulated while it was down. This behavior is commonly referred to as *store-and-forward messagin*g. `Durable` MDBs make MDBs tolerant of disconnections, whether they are intentional or the result of a partial failure. If `<subscription-durability>` is `NonDurable`, then

any messages the bean would have received while it was disconnected will be lost. Developers use `NonDurable` subscriptions when it's not critical that all messages be processed. Using a `NonDurable` subscription improves the performance of a JMS provider but significantly reduces the reliability of the MDBs.

When `<destination-type>` is `javax.jms.Queue`, as is the case in the ReservationProcessor EJB, durability is not a factor because of the nature of p2p or Queue based messaging systems. With a queue, messages may only be consumed once, and remain in the `Queue` until they are distributed to one of the Queue's listeners.

The rest of the elements in the deployment descriptor should already be familiar. The `<ejb-ref>` element provides JNDI ENC bindings for a remote EJB home object while the `<ejb-local-ref>` elements provide JNDI ENC bindings for local EJB home objects. Note that the `<resource-ref>` element that defined the JMS `QueueConnectionFactory` used by the ReservationProcessor EJB to send ticket messages is not accompanied by a `<resource-env-ref>` element. The `Queue` to which the tickets are sent is obtained from the *JMSReplyTo* header of the `MapMessage` itself, and not from the JNDI ENC.

The ReservationProcessor Clients

In order to test the ReservationProcessor EJB, we need to develop two new client applications: one to send reservation messages and the other to consume ticket messages produced by the ReservationProcessor EJB.

The Reservation Message Producer

The `JmsClient_ReservationProducer` is designed to send 100 reservation requests very quickly. The speed with which it sends these messages will force many MDB containers to use multiple instances to process the reservation messages.

```
import javax.jms.Message;
import javax.jms.MapMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.Queue;
import javax.jms.QueueSender;
import javax.jms.JMSException;
import javax.naming.InitalContext;

import com.titan.processpayment.CreditCardDO;
```

```
public class JmsClient_ReservationProducer {

    public static void main(String [] args){

        InitialContext jndiContext = getInitialContext()

        QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup(factoryName);

        Queue reservationQueue = (Queue)
        jndiContext.lookup(topicName);

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
        connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

        QueueSender sender = session.createSender(reservationQueue);

        Integer cruiseID = new Integer(1);

        for(int i = 0; i < 100; i++){
            MapMessage message = new MapMessage();

            message.setInt("CruiseID",1);
            message.setInt("CustomerID",i%10);
            message.setInt("CabinID",i);
            message.setDouble("Price", (double)1000+i);

            // the card expires in about 30 days
            Date expirationDate =
                 new Date(System.currentTimeMillis()+43200000);
            CreditCardDO card =
                        new CreditCardDO(9238302830291,
                                         expirationDate,
                                         CreditCardDO.MASTER_CARD);

            message.setObject("CreditCard", card);

            sender.send(message);

        }

        connect.close();
    }

    public static InitialContext getInitialContext()
    throws JMSException{
        // create vendor speicific JNDI Context here
    }
}
```

You may have noticed that the JmsClient_ReservationProducer sets the CustomerID, CruiseID, and CabinID as primitive int values, but the ReservationProcessorBean reads these values as java.lang.Integer types. This is not a mistake. The MapMessage automatically converts any primitive to its proper wrapper if that primitive is read using MapMessage.getObject(). So, for example, a named-value that is loaded into a MapMessage using setInt() can be read as an Integer using getObject():

```
MapMessage mapMsg = new MapMessage();

mapMsg.setInt("TheValue",3);

Integer myInteger = (Integer)mapMsg.getObject("TheValue");

if(myInteger.intValue() == 3 )
    // this will always be true
```

JMS has a cornucopia of features and details which are simply too extensive to cover in this book.

The Ticket Message Consumer

The JmsClient_TicketConsumer is designed to consume all the ticket messages delivered by ReservationProcessor instances to the queue. It consumes the messages and prints out the descriptions.

```
import javax.jms.Message;
import javax.jms.ObjectMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.Queue;
import javax.jms.QueueSender;
import javax.jms.JMSException;
import javax.naming.InitalContext;

import com.titan.travelagent.TicketDO;


public class JmsClient_TicketConsumer
extends javax.jms.MessageListener{

    public static void main(String [] args){

        new JmsClient_TicketConsumer();

        while(true){Thread.sleep(10000);}

    }
```

```
    public JmsClient_TicketConsumer
    throws Exception{

        InitialContext jndiContext = getInitialContext()

        QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup(factoryName);

        Queue ticketQueue = (Queue)
        jndiContext.lookup(topicName);

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
        connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

        QueueReceiver receiver = session.createReceiver(ticketQueue);

        receiver.setMessageListener(this);

        connect.start();
    }

    public void onMessage(Message message){
        try{

        ObjectMessage objMsg = (ObjectMessage)message;
        TicketDO ticket = (TicketDO)obj.Msg.getObject();
        System.out.println("*********************************");
        System.out.println(ticket);
        System.out.println("*********************************");

        }catch(JMSException jmsE){
            jmsE.printStackTrace();
        }
    }
    public static InitialContext getInitialContext()
    throws JMSException{
        // create vendor speicific JNDI Context here
    }
}
```

In order to make the ReservationProcessor EJB work with the two client applications, `JmsClient_ReservationProducer` and `JmsClient_TicketConsumer`, you must configure your EJB container and JMS provider so that it has two queues: one for reservation messages and another for ticket messages.

📖 Exercise 13.2, ReservationProcessor: The Message-driven bean

The Life Cycle of a Message-Driven Bean

Just as the entity and session beans have well-defined life cycles, so does the MDB. The MDB's life cycle has two states: *Does Not Exist* and *Method-Ready Pool*. The Method-Ready Pool is similar to the instance pool used for stateless session beans. Like stateless beans, MDBs define instance pooling in their life cycle.[3] Figure 13-4 illustrates the states and transitions that an MDB instance goes through in its lifetime.

[FIGURE]

Figure 13-4: MDB life cycle

Does Not Exist

When a bean is in the Does Not Exist state, it is not an instance in the memory of the system. In other words, it has not been instantiated yet.

The Method-Ready Pool

MDB instances enter the Method-Ready Pool as the container needs them. When the EJB server is first started, it will probably create a number of MDB instances and enter them into the Method-Ready Pool. (The actual behavior of the server depends on the implementation.) When the number of MDB instances handling incoming messages is insufficient, more can be created and added to the pool.

Transitioning to the Method-Ready Pool

When an instance transitions from the Does Not Exist state to the Method-Ready Pool, three operations are performed on it. First, the bean instance is instantiated by invoking the `Class.newInstance()` method on the MDB class. Second, the `setMessageDrivenContext()` method is invoked when the instance receives its reference to the `EJBContext`. The `MessageDrivenContext` reference may be stored in an instance field of the MDB.

Finally, the no-argument `ejbCreate()` method is invoked on the bean instance. Remember that an MDB only has one `ejbCreate()` method, which takes no arguments. The `ejbCreate()` method is invoked only once in the life cycle of the MDB.

---

3 Some vendors do *not* pool MDB instances, but may instead create and destroy instances with each new message. This is an implementation-specific decision that shouldn't impact the specified life cycle of the stateless bean instance.

MDBs are not subject to activation, so they can maintain open connections to resources for their entire life cycle.[4] The `ejbRemove()` method should close any open resources before the MDB is evicted from memory at the end of its life cycle.

Life in the Method-Ready Pool

Once an instance is in the Method-Ready Pool, it is ready to handle incoming messages. When a message is delivered to an MDB it is delegated to any available instance in the Method-Ready Pool. While the instance is executing the request, it is unavailable to process other messages. The MDB can handle many messages simultaneously, delegating the responsibility of handling each message to a different MDB instance. Once the instance has finished, it is immediately available to handle a new message.

When a message is delegated to an instance, the MDB instance's `MessageDrivenContext` changes to reflect the new transaction context.

MDBs are not subject to activation and do not have `ejbActivate()` or `ejbPassivate()` callback methods. The reason is simple: MDB instances have no conversational state that needs to be preserved. (*Stateful* session beans do depend on activation, as we'll see later.)

Transitioning out of the Method-Ready Pool: The death of an MDB instance

Bean instances leave the Method-Ready Pool for the Does Not Exist state when the server no longer needs the instance. This occurs when the server decides to reduce the total size of the Method-Ready Pool by evicting one or more instances from memory. The process begins by invoking the `ejbRemove()` method on the instance. At this time, the bean instance should perform any cleanup operations, like closing open resources. The `ejbRemove()` method is only invoked once in the life cycle of an MDB's instance—when it is about to transition to the Does Not Exist state. During the `ejbRemove()` method, the `MessageDrivenContext` and access to the JNDI ENC is still available to the bean instance. Following the execution of the `ejbRemove()` method, the bean is dereferenced and eventually garbage collected.

---

4 The duration of an MDB instance's life is assumed to be very long. However, some EJB servers may actually destroy and create instances with every new message, making this strategy less attractive. Consult your vendor's documentation for details on how your EJB server handles stateless instances.

14

Transactions

ACID Transactions

To understand how transactions work, we will revisit the TravelAgent EJB, a stateful session bean that encapsulates the process of making a cruise reservation for a customer. Here is the TravelAgent's `bookPassage()` method in EJB 2.0 and 1.1 versions:

EJB 2.0: bookPassage() method

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome =
        (ReservationHomeLocal)
        jndiContext.lookup
                    ("java:comp/env/ejb/ReservationHome");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);

        Object ref = jndiContext.lookup
                    ("java:comp/env/ejb/ProcessPaymentHome");
```

```
        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
        PortableRemoteObject.narrow
                    (ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

EJB 1.1: bookPassage() method

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome =
        (ReservationHomeRemote)getHome("ReservationHome",
                                ReservationHomeRemote.class);
        ReservationRemote reservation =
            resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
                    getHome("ProcessPaymentHome",
                            ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

The TravelAgent EJB is a fairly simple session bean, and its use of other EJBs is a typical example of business object design and workflow. Unfortunately, good business object design is not enough to make these EJBs useful in an industrial-strength application. The problem is not with the definition of the EJBs or the

workflow; the problem is that a good design doesn't, in and of itself, guarantee that the TravelAgent's `bookPassage()` method represents a good *transaction*. To understand why, we will take a closer look at what a transaction means and what criteria a transaction must meet to be considered reliable.

In business, a transaction usually involves an exchange between two parties. When you purchase an ice cream cone, you exchange money for food; when you work for a company, you exchange skill and time for money (which you use to buy more ice cream). When you are involved in these exchanges, you monitor the outcome to ensure that you don't get "ripped off." If you give the ice cream vendor a $20 bill, you don't want him to drive off without giving you your change; you want to make sure that your paycheck reflects all the hours that you worked. By monitoring these commercial exchanges, you are attempting to ensure the reliability of the transactions; you are making sure that the transaction meets everyone's expectations.

In business software, a transaction embodies the concept of a commercial exchange. A business system transaction (transaction for short) is the execution of a *unit-of-work* that accesses one or more shared resources, usually databases. A unit-of-work is a set of activities that relate to each other and must be completed together. The reservation process is a unit-of-work made up of several activities: recording a reservation, debiting a credit card, and generating a ticket together make up a unit-of-work.

Transactions are part of many different types of systems. In each transaction, the objective is the same: to execute a unit-of-work that results in a reliable exchange. Here are some examples of other types of business systems that employ transactions:

ATM
: The ATM (automatic teller machine) you use to deposit, withdraw, and transfer funds, executes these units-of-work as transactions. In an ATM withdrawal, for example, the ATM checks to make sure you don't overdraw and then debits your account and spits out some money.

Online book order
: You've probably purchased many of your Java books from an online bookseller—maybe even this book. This type of purchase is also a unit-of-work that takes place as a transaction. In an online book purchase, you submit your credit card number, it is validated, and then a charge is made for price of the book, and an order to ship you the book is sent to the bookseller's warehouse.

Medical system
: In a medical system, important data—some of it critical—is recorded about patients every day, including information about clinical visits, medical procedures, prescriptions, and drug allergies. The doctor prescribes the drug, then the system checks for allergies, contraindications, and appropriate dosages. If all tests pass, then the drug can be administered. The tasks just described make up a unit-of-work in a medical system. A unit-

of-work in a medical system may not be financial, but it's just as important. A failure to identify a drug allergy in a patient could be fatal.

As you can see, transactions are often complex and usually involve the manipulation of a lot of data. Mistakes in data can cost money, or even a life. Transactions must therefore preserve data integrity, which means that the transaction must work perfectly every time or not be executed at all. This is a pretty tall order, especially for complex systems. As difficult as this requirement is, however, when it comes to commerce there is no room for error. Units-of-work involving money or anything of value always require the utmost reliability, because errors impact the revenues and the well-being of the parties involved.

To give you an idea of the accuracy required by transactions, think about what would happen if a transactional system suffered from seemingly infrequent errors. ATMs provide customers with convenient access to their bank accounts and represent a significant percentage of the total transactions in personal banking. The number of transactions handled by ATMs are simple but numerous, providing us with a great example of why transactions must be error proof. Let's say that a bank has 100 ATMs in a metropolitan area, and each ATM processes 300 transactions (deposits, withdrawals, or transfers) a day for a total of 30,000 transactions per day. If each transaction, on average, involves the deposit, withdrawal, or transfer of about $100, about three million dollars would move through the ATM system per day. In the course of a year, that's a little over a billion dollars:

(365 days) ? (100 ATMs) ? (300 transactions) ? ($100.00) = $1,095,000,000.00

How well do the ATMs have to perform in order for them to be considered reliable? For the sake of argument, let's say that ATMs execute transactions correctly 99.99% of the time. This seems to be more than adequate: after all, only one out of every ten thousand transactions executes incorrectly. But over the course of a year, if you do the math, that could result in over $100,000 in errors!

$1,095,000,000.00 ? .01% = $109,500.00

Obviously, this is an oversimplification of the problem, but it illustrates that even a small percentage of errors is unacceptable in high-volume or mission-critical systems. For this reason, experts in the field of transaction services have identified four characteristics of a transaction that must be followed in order to say that a system is safe. Transactions must be atomic, consistent, isolated, and durable (ACID)—the four horsemen of transaction services. Here's what each term means:

Atomic

To be atomic, a transaction must execute completely or not at all. This means that every task within a unit-of-work must execute without error. If any of the tasks fails, the entire unit-of-work or transaction is aborted, meaning that changes to the data are undone. If all the tasks execute successfully, the

transaction is committed, which means that the changes to the data are made permanent or durable.

Consistent
: Consistency is a transactional characteristic that must be enforced by both the transactional system and the application developer. Consistency refers to the integrity of the underlying data store. The transactional system fulfills its obligation in consistency by ensuring that a transaction is atomic, isolated, and durable. The application developer must ensure that the database has appropriate constraints (primary keys, referential integrity, and so forth) and that the unit-of-work, the business logic, doesn't result in inconsistent data (data that is not in harmony with the real world it represents). In an account transfer, for example, a debit to one account must equal the credit to the other account.

Isolated
: A transaction must be allowed to execute without interference from other processes or transactions. In other words, the data that a transaction accesses cannot be affected by any other part of the system until the transaction or unit-of-work is completed.

Durable
: Durability means that all the data changes made during the course of a transaction must be written to some type of physical storage before the transaction is successfully completed. This ensures that the changes are not lost if the system crashes.

To get a better idea of what these principles mean, we will examine the TravelAgent EJB in terms of the four ACID properties.

Is the TravelAgent EJB Atomic?

Our first measure of the TravelAgent EJB's reliability is its atomicity: does it ensure that the transaction executes completely or not at all? What we are really concerned with are the critical tasks that change or create information. In the `bookPassage()` method, a Reservation EJB is created, the ProcessPayment EJB debits a credit card, and a `TicketDO` object is created. All of these tasks must be successful for the entire transaction to be successful.

To understand the importance of the atomic characteristic, you have to imagine what would happen if even one of the subtasks failed to execute. If, for example, the creation of a Reservation EJB failed but all other tasks succeeded, your customer would probably end up getting bumped from the cruise or sharing the cabin with a stranger. As far as the travel agent is concerned, the `bookPassage()` method executed successfully because a `TicketDO` was generated. If a ticket is generated without the creation of a reservation, the state of the business system becomes inconsistent with reality because the customer paid for a ticket but the reservation was not recorded. Likewise, if the ProcessPayment EJB fails to charge the customer's credit card, the customer gets

a free cruise. He may be happy, but management isn't. Finally, if the `TicketDO` is never created, the customer would have no record of the transaction and probably wouldn't be allowed onto the ship.

So the only way `bookPassage()` can be completed is if all the critical tasks execute successfully. If something goes wrong, the entire process must be aborted. Aborting a transaction requires more than simply not finishing the tasks; in addition, all the tasks that did execute within the transaction must be undone. If, for example, the creation of the Reservation EJB and `ProcessPayment.byCredit()` method succeeded but the creation of the `TicketDO` failed throwing an exception from constructor, then the reservation record and payment records must not be added to the database.

Is the TravelAgent EJB Consistent?

In order for a transaction to be consistent, the state of the business system must make sense after the transaction has completed. In other words, the *state* of the business system must be consistent with the reality of the business. This requires that the transaction enforce the atomic, isolated, and durable characteristics of the transaction, and it also requires diligent enforcement of integrity constraints by the application developer. If, for example, the application developer fails to include the credit card charge operation in the `bookPassage()` method, the customer would be issued a ticket but would never be charged. The data would be inconsistent with the expectation of the business—a customer should be charged for passage. In addition, the database must be set up to enforce integrity constraints. For example, it should not be possible for a record to be added to the `RESERVATION` table unless the `CABIN_ID`, `CRUISE_ID`, and `CUSTOMER_ID` foreign keys map to corresponding records in the `CABIN`, `CRUISE`, and `CUSTOMER` tables, respectively. If a `CUSTOMER_ID` is used that doesn't map to a `CUSTOMER` record, referential integrity should cause the database to throw an error message.

Is the TravelAgent EJB Isolated?

If you are familiar with the concept of thread synchronization in Java or row-locking schemes in relational databases, isolation will be a familiar concept. To be isolated, a transaction must protect the data that it is accessing from other transactions. This is necessary to prevent other transactions from interacting with data that is in transition. In the TravelAgent EJB, the transaction is isolated to prevent other transactions from modifying the EJBs that are being updated. Imagine the problems that would arise if separate transactions were allowed to change any entity bean at any time—transactions would walk all over each other. You could easily have several customers book the same cabin because their travel agents happened to make their reservations at the same time.

The isolation of data accessed by EJBs doesn't mean that the entire application shuts down during a transaction. Only those entity beans and data directly affected by the transaction are isolated. In the TravelAgent EJB, for example, the transaction isolates only the Reservation EJB created. There can be many Reservation EJBs in existence; there's no reason these other EJBs can't be accessed by other transactions.

Is the TravelAgent EJB Durable?

To be durable, the `bookPassage()` method must write all changes and new data to a permanent data store before it can be considered successful. While this may seem like a no-brainer, often it isn't what happens in real life. In the name of efficiency, changes are often maintained in memory for long periods of time before being saved on a disk drive. The idea is to reduce disk accesses—which slow systems down—and only periodically write the cumulative effect of data changes. While this approach is great for performance, it is also dangerous because data can be lost when the system goes down and memory is wiped out. Durability requires the system to save all updates made within a transaction as the transaction successfully completes, thus protecting the integrity of the data.

In the TravelAgent EJB, this means that the new `RESERVATION` and `PAYMENT` records inserted are made persistent before the transaction can complete successfully. Only when the data is made durable are those specific records accessible through their respective EJBs from other transactions. Hence, durability also plays a role in isolation. A transaction isn't finished until the data is successfully recorded.

Ensuring that transactions adhere to the ACID principles requires careful design. The system has to monitor the progress of a transaction to ensure that it does all its work, that the data is changed correctly, that transactions don't interfere with each other, and that the changes can survive a system crash. Engineering all this functionality into a system is a lot of work, and not something you would want to reinvent for every business system you worked on. Fortunately, EJB is specifically designed to support transactions automatically, making the development of transactional systems easier. The rest of this chapter examines how EJB supports transactions implicitly (through declarative transaction attributes) and explicitly (through the Java Transaction API).

Declarative Transaction Management

One of the primary advantages of Enterprise JavaBeans is that it allows for declarative transaction management. Without this feature, transactions must be controlled using explicit transaction demarcation. This involves the use of fairly complex APIs like the OMG's OTS (Object Transaction Service) or its Java implementation, JTS (Java Transaction Service). Explicit demarcation is difficult for developers to use at best, particularly if you are new to transactional systems.

In addition, explicit transaction demarcation requires that the transactional code be written within the business logic, which reduces the clarity of the code and more importantly creates inflexible distributed objects. Once transaction demarcation is "hardcoded" into the business object, changes in transaction behavior require changes to the business logic itself. We talk more about explicit transaction management and EJB later in this chapter.

With EJB's declarative transaction management, the transactional behavior of EJBs can be controlled using the deployment descriptor, which sets transaction attributes for individual enterprise bean methods. This means that the transactional behavior of a EJB within an application can be changed easily without changing the EJB's business logic. In addition, a EJB deployed in one application can be defined with very different transactional behavior than the same EJB deployed in a different application. Declarative transaction management reduces the complexity of transactions for EJB developers and application developers and makes it easier for you to create robust transactional applications.

Transaction Scope

Transaction scope is a crucial concept for understanding transactions. In this context, transaction scope means those EJBs—both session and entity—that are participating in a particular transaction.

In the `bookPassage()` method of the TravelAgent EJB, all the EJBs involved are a part of the same transaction scope. The scope of the transaction starts when a client invokes the TravelAgent EJB's `bookPassage()` method. Once the transaction scope has started, it is *propagated* to both the newly created Reservation EJB and the ProcessPayment EJB:

As you know, a transaction is a unit-of-work that is made up of one or more tasks. In a transaction, all the tasks that make up the unit-of-work must succeed for the entire transaction to succeed; the transaction must be atomic. If any task fails, the updates made by all the other tasks in the transaction will be rolled back or undone. In EJB, tasks are expressed as enterprise bean methods, and a unit-of-work consists of every enterprise bean method invoked in a transaction. The scope of a transaction includes every EJB that participates in the unit-of-work.

It is easy to trace the scope of a transaction by following the thread of execution. If the invocation of the `bookPassage()` method begins a transaction, then logically, the transaction ends when the method completes. The scope of the `bookPassage()` transaction would include the TravelAgent, Reservation, and ProcessPayment EJBs—every EJB touched by the `bookPassage()` method. A transaction is propagated to an EJB when that EJB's method is invoked and included in the scope of a transaction.

A transaction can end if an exception is thrown while the `bookPassage()` method is executing. The exception could be thrown from one of the other EJBs or from the `bookPassage()` method itself. An exception may or may not cause a rollback, depending on its type. More about exceptions and transactions later.

The thread of execution isn't the only factor that determines whether a EJB is included in the scope of a transaction; the EJB's transaction attributes also play a role. Determining whether a EJB participates in the transaction scope of any unit-of-work is accomplished either implicitly using EJB's transaction attributes or explicitly using the Java Transaction API (JTA).

Transaction Attributes

As an application developer, you do *not* normally need to control transactions explicitly when using an EJB server. EJB servers can manage transactions implicitly, based on the transaction attributes established for EJBs at deployment time. The ability to specify how business objects participate in transactions through attribute-based programming is a common characteristic of CTMs, and one of the most important features of the EJB component model.

When an EJB is deployed, you can set its runtime transaction attribute in the deployment descriptor to one of several values. The list below shows the XML attribute values used to specify these transaction attributes:

- `NotSupported`
- `Supports`
- `Required`
- `RequiresNew`
- `Mandatory`
- `Never`

Using transaction attributes simplifies building transactional applications by reducing the risks associated with improper use of transactional protocols like JTA (discussed later in this chapter). It's more efficient and easier to use transaction attributes than to control transactions explicitly.

It is possible to set a transaction attribute for the entire EJB (in which case, it applies to all methods) or to set different transaction attributes for individual methods. The former is much simpler and less error prone, but setting attributes at the method level offers more flexibility. The code fragments in the following sections show how the default transaction attribute of a EJB can be set in the EJB's deployment descriptor.

Setting a transaction attribute

In the XML deployment descriptor, a `<container-transaction>` element specifies the transaction attributes for the EJBs described in the deployment descriptor:

```
<ejb-jar>
  ...
  <assembly-descriptor>
     ...
     <container-transaction>
        <method>
            <ejb-name>TravelAgentEJB</ejb-name>
            <method-name> * </method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
     </container-transaction>
     <container-transaction>
        <method>
            <ejb-name>TravelAgentEJB</ejb-name>
            <method-name>listAvailableCabins</method-name>
        </method>
        <trans-attribute>Supports</trans-attribute>
     </container-transaction>
     ...
   </assembly-descriptor>
   ...
</ejb-jar>
```

This deployment descriptor specifies the transaction attributes for the TravelAgent EJB. The `<container-transaction>` element specifies a method and the transaction attribute that should be applied to that method. The first `<container-transaction>` element specifies that all methods by default have a transaction attribute of `Required`; the `*` is a wildcard that indicates all of the methods of the TravelAgent EJB. The second `<container-transaction>` element overrides the default setting to specify that the `listAvailableCabins()` method will have a `Supports` transaction attribute. Note that we have to specify which EJB we're referring to with the `<ejb-name>` element; an XML deployment descriptor can cover many EJBs.

Transaction Attributes Defined

Here are the definitions of the transaction attributes listed above. In a few of the definitions, we say that the client transaction is *suspended*. This means that the transaction is not propagated to the enterprise bean method being invoked; propagation of the transaction is temporarily halted until the enterprise bean method returns.

To make things easier, we'll talk about attribute types as if they were bean types: for example, we'll say a "`Required` EJB" as shorthand for "an enterprise bean with the `Required` transactional attribute".

NotSupported

Invoking a method on an EJB with this transaction attribute suspends the transaction until the method is completed. This means that the transaction scope is not propagated to the *NotSupported* EJB or any of the EJBs it calls. Once the method on the *NotSupported* EJB is done, the original transaction resumes its execution.

Figure 8-1 shows that a *Not Supported* EJB does not propagate the client transaction when one of its methods is invoked.

[FIGURE (modified 8-1)]

Figure 14-1: Not Supported attribute

Supports

This attribute means that the enterprise bean method will be included in the transaction scope if it is invoked within a transaction. In other words, if the EJB or client that invokes the *Supports* EJB is part of a transaction scope, the *Supports* EJB and all EJBs accessed by it become part of the original transaction. However, the *Supports* EJB doesn't have to be part of a transaction and can interact with clients and other EJBs that are not included in a transaction scope.

Figure 8-3(a) shows the *Supports* EJB being invoked by a transactional client and propagating the transaction. Figure 8-3(b) shows the *Supports* EJB being invoked from a non-transactional client.

[FIGURE (modifed 8-2)]

Figure 14-2: Supports attribute

Required

This attribute means that the enterprise bean method must be invoked within the scope of a transaction. If the calling client or EJB is part of a transaction, the *Required* EJB is automatically included in its transaction scope. If, however, the calling client or EJB is not involved in a transaction, the *Required* EJB starts its own new transaction. The new transaction's scope covers only the *Required* EJB and all other EJBs accessed by it. Once the method invoked on the *Required* EJB is done, the new transaction's scope ends.

Figure 8-5(a) shows the *Required* EJB being invoked by a transactional client and propagating the transaction. Figure 8-5(b) shows the *Required* EJB being invoked from a non-transactional client, which causes it to start its own transaction.

[FIGURE (modifed 8-3)]

Figure 14-3: Required attribute

Requires New

This attribute means that a new transaction is always started. Regardless of whether the calling client or EJB is part of a transaction, a method with the *RequiresNew* attribute begins a new transaction when invoked. If the calling client is already involved in a transaction, that transaction is suspended until the *RequiresNew* EJB's method call returns. The new transaction's scope only covers the *RequiresNew* EJB and all the EJBs accessed by it. Once the method invoked on the *RequiresNew* EJB is done, the new transaction's scope ends and the original transaction resumes.

Figure 8-7(a) shows the *RequiresNew* EJB being invoked by a transactional client. The client's transaction is suspended while the EJB executes under its own transaction. Figure 8-7(b) shows the *RequiresNew* EJB being invoked from a non-transactional client; the *RequiresNew* executes under its own transaction.

[FIGURE (modified 8-4)]

Figure 14-4: Requires New attribute

Mandatory

This attribute means that the enterprise bean method must always be made part of the transaction scope of the calling client. If the calling client or EJB is not part of a transaction, the invocation will fail, throwing a `javax.transaction.TransactionRequiredException` to remote clients or a `javax.ejb.TransactionRequiredLocalException` to local EJB 2.0 clients.

Figure 8-9(a) shows the *Mandatory* EJB being invoked by a transactional client and propagating the transaction. Figure 8-9(b) shows the *Mandatory* EJB being invoked from a non-transactional client; the method throws the `TransactionRequiredException` to remote clients or `TransactionRequredLocalException` to local EJB 2.0 clients, because there is no transaction scope.

[FIGURE (modifed 8-5)]

Figure 14-5: Mandatory attribute

Never

This attribute means that the enterprise bean method must never be invoked within the scope of a transaction. If the calling client or EJB is part of a transaction, the *Never* EJB will throw a `RemoteException` to remote clients or an `EJBException` to local EJB 2.0 clients. If, however, the calling client or EJB is not involved in a transaction, the *Never* EJB will execute normally without a transaction.

Figure 8-11(a) shows the *Never* EJB being invoked by a non-transactional client. Figure 8-11(b) shows the *Never* EJB being invoked by transactional client; the method throws the `RemoteException` to remote clients or

`EJBException` to local EJB 2.0 clients, because the method can never be invoked by a client or EJB that is included in a transaction.

[FIGURE (modifed 8-6)]

Figure 14-6: Never attribute

EJB 2.0: Message-driven beans and transaction attributes

Message-driven beans may *only* declare the *NotSupported* or *Required* transaction attributes. The other transaction attributes don't make sense in message-driven beans because they apply to client-initiated transactions. The *Supports*, *RequiresNew*, *Mandatory*, and *Never* attributes are all relative to the transaction context of the client. For example, the *Mandatory* attribute requires the client to have a transaction in progress before calling the enterprise bean. This is meaningless for a message-driven bean, which is uncoupled from the client.

The *NotSupported* transaction attribute indicates the message will be processed without a transaction. The *Required* transaction attribute indicates that the message will be processed with a container-initiated transaction.

Transaction Propagation

To illustrate the impact of transaction attributes on enterprise bean methods, we'll look once again at the `bookPassage()` method of the TravelAgent EJB created in Chapter 7 (see the listings at the earlier in the chapter):

In order for `bookPassage()` to execute as a successful transaction, both the creation of the Reservation EJB and the charge to the customer must be successful. This means that both operations must be included in the same transaction. If either operation fails, the entire transaction fails. We could have specified the *Required* transaction attribute as the default for all the EJB involved, because that attribute enforces our desired policy that all EJBs must execute within a transaction and thus ensures data consistency.

As a transaction monitor, an EJB server watches each method call in the transaction. If any of the updates fail, all the updates to all the EJBs will be reversed or *rolled back*. A rollback is like an *undo* command. If you have worked with relational databases, then the concept of a rollback should be familiar. Once an update is executed, you can either commit the update or roll it back. A commit makes the changes requested by the update permanent; a rollback aborts the update and leaves the database in its original state. Making EJBs transactional provides the same kind of rollback/commit control. For example, if the Reservation EJB cannot be created, the charge made by the ProcessPayment EJB is rolled back. Transactions make updates an all-or-nothing proposition. This ensures that the unit-of-work, like the `bookPassage()` method, executes as intended, and it prevents inconsistent data from being written to databases.

In cases where the container implicitly manages the transaction, the commit and rollback decisions are handled automatically. When transactions are managed explicitly within an enterprise bean or by the client, the responsibility falls on the enterprise bean or application developer to commit or roll back a transaction. Explicit demarcation of transactions is covered in detail later in this chapter.

Let's assume that the TravelAgent EJB is created and used on a client as follows:

```
TravelAgent agent = agentHome.create(customer);
agent.setCabinID(cabin_id);
agent.setCruiseID(cruise_id):
try {
  agent.bookPassage(card,price);
} catch(Exception e) {
  System.out.println("Transaction failed!");
}
```

Furthermore, let's assume that the `bookPassage()` method has been given the transaction attribute *RequiresNew*. In this case, the client that invokes the `bookPassage()` method is not itself part of a transaction. When `bookPassage()` is invoked on the TravelAgent EJB, a new transaction is created, as dictated by the *RequiresNew* attribute. This means that the TravelAgent EJB registers itself with the EJB server's transaction manager, which will manage the transaction automatically. The transaction manager coordinates transactions, propagating the transaction scope from one EJB to the next to ensure that all EJBs touched by a transaction are included in the transaction's unit-of-work. That way, the transaction manager can monitor the updates made by each enterprise bean and decide, based on the success of those updates, whether to commit all changes made by all enterprise beans to the database or roll them all back. If a *system exception* is thrown by the `bookPassage()` method, the transaction is automatically rolled back. We will talk more about exceptions later in this chapter.

When the `byCredit()` method is invoked within the `bookPassage()` method, the ProcessPayment EJB registers with the transaction manager under the transactional context that was created for the TravelAgent EJB; the transactional context is propagated to the ProcessPayment EJB. When the new Reservation EJB is created, it is also registered with the transaction manager under the same transaction. When all the EJBs are registered and their updates made, the transaction manager checks to ensure that their updates will work. If all the updates will work, then the transaction manager allows the changes to become permanent. If one of the EJBs reports an error or fails, any changes made by either the ProcessPayment or Reservation EJB are rolled back by the transaction manager. Figure 8-15 illustrates the propagation and management of the TravelAgent EJB's transactional context.

[FIGURE (modified 8-8)]

Figure 14-7: Managing the TravelAgent EJB's transactional context

In addition to managing transactions in its own environment, an EJB server can coordinate with other transactional systems. If, for example, the ProcessPayment EJB actually came from a different EJB server than the TravelAgent EJB, the two EJB servers would cooperate to manage the transaction as one unit-of-work. This is called a *distributed transaction*.[1]

A distributed transaction is a great deal more complicated, requiring what is called a *two- phase commit* (2-PC or TPC). 2-PC is a mechanism that allows transactions to be managed across different servers and resources (e.g. databases and JMS providers). The details of a 2-PC are beyond the scope of this book, but a system that supports it will not require any extra operations by a EJB or application developer. If distributed transactions are supported, the protocol for propagating transactions, as discussed earlier, will be supported. In other words, as an application or EJB developer, you should not notice a difference between local and distributed transactions.

Isolation and Database Locking

Transaction isolation (the "I" in ACID) is a critical part of any transactional system. This section explains isolation conditions, database locking, and transaction isolation levels. These concepts are important when deploying any transactional system.

Dirty, Repeatable, and Phantom Reads

Transaction isolation is defined in terms of isolation conditions called *dirty reads*, *repeatable reads*, and *phantom reads*. These conditions describe what can happen when two or more transactions operate on the same data.[2]

To illustrate these conditions, let's think about two separate client applications using their own instances of the TravelAgent EJB to access the same data—specifically, a cabin record with the primary key of 99. These examples revolve around the `RESERVATION` table, which is accessed by both the `bookPassage()` method (through the Reservation EJB) and the `listAvailableCabins()` method (through JDBC). It might be a good idea to go back to Chapter 12 and review how the `RESERVATION` table is accessed through these methods. This will help you to understand how two transactions

1 Not all EJB servers support distributed transactions.

2 Isolation conditions are covered in detail by the ANSI SQL-92 Specification, Document Number: ANSI X3. 135-1992 (R1998).

executed by two different clients can impact each other. Assume that both methods have a transaction attribute of *Required*.

Dirty reads

A dirty read occurs when the first transaction reads uncommitted changes made by a second transaction. If the second transaction is rolled back, the data read by the first transaction becomes invalid because the rollback undoes the changes. The first transaction won't be aware that the data it has read has become invalid. Here's a scenario showing how a dirty read can occur (illustrated in Figure 14-8):

1. Time 10:00:00: Client 1 executes the TravelAgent.bookPassage() method. Along with the Customer and Cruise EJBs, Client 1 had previously chosen Cabin 99 to be included in the reservation.
2. Time 10:00:01: Client 1's TravelAgent EJB creates a Reservation EJB within the bookPassage() method. The Reservation EJB's create() method inserts a record into the RESERVATION table, which reserves Cabin 99.
3. Time 10:00:02: Client 2 executes TravelAgent.listAvailableCabins(). Cabin 99 has been reserved by Client 1, so it is not in the list of available cabins that are returned from this method.
4. Time 10:00:03: Client 1's TravelAgent EJB executes the ProcessPayment.byCredit() method within the bookPassage() method. The byCredit() method throws an exception because the expiration date on the credit card has passed.
5. Time 10:00:04: The exception thrown by the ProcessPayment EJB causes the entire bookPassage() transaction to be rolled back. As a result, the record inserted into the RESERVATION table when the Reservation EJB was created is not made durable (it is removed). Cabin 99 is now available.

[FIGURE (use 8-9)]

Figure 14-8: A dirty read

Client 2 is now using an invalid list of available cabins because Cabin 99 is available but is not included in the list. This would be serious if Cabin 99 was the last available cabin because Client 2 would inaccurately report that the cruise was booked. The customer would presumably try to book a cruise on a competing cruise line.

Repeatable reads

A repeatable read is when the data read is guaranteed to look the same if read again during the same transaction. Repeatable reads are guaranteed in one of two ways: either the data read is locked against changes or the data read is a snapshot that doesn't reflect changes. If the data is locked, then it cannot be changed by any other transaction until this transaction ends. If the data is a snapshot, then other transactions can change the data, but these changes won't

be seen by this transaction if the read is repeated. Here's an example of a repeatable read (illustrated in Figure 14-9):

1. Time 10:00:00: Client 1 begins an explicit javax.transaction.UserTransaction.
2. Time 10:00:01: Client 1 executes TravelAgent.listAvailableCabins(2), asking for a list of available cabins that have two beds. Cabin 99 is in the list of available cabins.
3. Time 10:00:02: Client 2 is working with an interface that manages Cabin EJBs. Client 2 attempts to change the bed count on Cabin 99 from 2 to 3.
4. Time 10:00:03: Client 1 re-executes the TravelAgent.listAvailableCabins(2). Cabin 99 is still in the list of available cabins.

[FIGURE (use 8-10)]

Figure 14-9: Repeatable read

This example is somewhat unusual because it uses `javax.transaction.UserTransaction`. This class is covered in more detail later in this chapter; essentially, it allows a client application to control the scope of a transaction explicitly. In this case, Client 1 places transaction boundaries around both calls to `listAvailableCabins()`, so that they are a part of the same transaction. If Client 1 didn't do this, the two `listAvailableCabins()` methods would have executed as separate transactions and our repeatable read condition would not have occurred.

Although Client 2 attempted to change the bed count for Cabin 99 to 3, Cabin 99 still shows up in the Client 1 call to `listAvailableCabins()` when a bed count of 2 is requested. This is because either Client 2 was prevented from making the change (because of a lock), or Client 2 was able to make the change, but Client 1 is working with a snapshot of the data that doesn't reflect that change.

A *nonrepeatable read* is when the data retrieved in a subsequent read within the same transaction can return different results. In other words, the subsequent read can see the changes made by other transactions.

Phantom reads

Phantom reads occur when new records added to the database are detectable by transactions that started prior to the insert. Queries will include records added by other transactions after their transaction has started. Here's a scenario that includes a phantom read (illustrated in Figure 14-10):

1. Time 10:00:00: Client 1 begins an explicit javax.transaction.UserTransaction.
2. Time 10:00:01: Client 1 executes TravelAgent.listAvailableCabins(2), asking for a list of available cabins that have two beds. Cabin 99 is in the list of available cabins.

3. Time 10:00:02: Client 2 executes bookPassage() and creates a Reservation EJB. The reservation inserts a new record into the RESERVATION table, reserving cabin 99.

4. Time 10:00:03: Client 1 re-executes the TravelAgent.listAvailableCabins(2). Cabin 99 is no longer in the list of available cabins.

[FIGURE (use 8-11)]

Figure 14-10: Phantom read

Client 1 places transaction boundaries around both calls to `listAvailableCabins()`, so that they are a part of the same transaction. In this case, the reservation was made between the `listAvailableCabins()` queries in the same transaction. Therefore, the record inserted in the `RESERVATION` table didn't exist when the first `listAvailableCabins()` method is invoked, but it does exist and is visible when the second `listAvailableCabins()` method is invoked. The record inserted is a *phantom record*.

Database Locks

Databases, especially relational databases, normally use several different locking techniques. The most common are *read locks*, *write locks*, and *exclusive write locks*. (I've taken the liberty of adding "snapshots," although this isn't a formal term.) These locking mechanisms control how transactions access data concurrently. Locking mechanisms impact the read conditions that were just described. These types of locks are simple concepts that are not directly addressed in the EJB specification. Database vendors implement these locks differently, so you should understand how your database addresses these locking mechanisms to best predict how the isolation levels described in this section will work.

Read locks
: Read locks prevent other transactions from changing data read during a transaction until the transaction ends, thus preventing nonrepeatable reads. Other transactions can read the data but not write it. The current transaction is also prohibited from making changes. Whether a read lock locks only the records read, a block of records, or a whole table depends on the database being used.

Write locks
: Write locks are used for updates. A write lock prevents other transactions from changing the data until the current transaction is complete. A write lock allows dirty reads, by other transactions and by the current transaction itself. In other words, the transaction can read its own uncommitted changes.

Exclusive write locks
: Exclusive write locks are used for updates. An exclusive write lock prevents other transactions from reading or changing data until the current

transaction is complete. An exclusive write lock prevents dirty reads by other transactions. Other transactions are not allowed to read the data while it is exclusively locked. Some databases do not allow transactions to read their own data while it is exclusively locked.

Snapshots

Some databases get around locking by providing every transaction with its own snapshot of the data. A snapshot is a frozen view of the data that is taken when the transaction begins. Snapshots can prevent dirty reads, nonrepeatable reads, and phantom reads. Snapshots can be problematic because the data is not real-time; it is old the instant the snapshot is taken.

Transaction Isolation Levels

Transaction isolation is defined in terms of the isolation conditions (*dirty reads*, *repeatable reads*, and *phantom reads*). Isolation levels are commonly used in database systems to describe how locking is applied to data within a transaction.[3] The following terms are usually used to discuss isolation levels:

Read Uncommitted

The transaction can read uncommitted data (data changed by a different transaction that is still in progress).

Dirty reads, nonrepeatable reads, and phantom reads can occur. Bean methods with this isolation level can read uncommitted change.

Read Committed

The transaction cannot read uncommitted data; data that is being changed by a different transaction cannot be read.

Dirty reads are prevented; nonrepeatable reads and phantom reads can occur. Bean methods with this isolation level cannot read uncommitted data.

Repeatable Read

The transaction cannot change data that is being read by a different transaction.

Dirty reads and nonrepeatable reads are prevented; phantom reads can occur. Bean methods with this isolation level have the same restrictions as Read Committed and can only execute repeatable reads.

Serializable

The transaction has exclusive read and update privileges to data; different transactions can neither read nor write the same data.

Dirty reads, nonrepeatable reads, and phantom reads are prevented. This isolation level is the most restrictive.

---

3 Isolation conditions are covered in detail by ANSI SQL-92 Specification, Document Number: ANSI X3.135- 1992 (R1998).

These isolation levels are the same as those defined for JDBC. Specifically, they map to the static final variables in the `java.sql.Connection` class. The behavior modeled by the isolation levels in the connection class is the same as the behavior described here.

The exact behavior of these isolation levels depends largely on the locking mechanism used by the underlying database or resource. How the isolation levels work depends in large part on how your database supports them.

In EJB, the deployer sets transaction isolation levels in a vendor specific way if the container manages the transaction. The EJB developer sets the transaction isolation level if the enterprise bean manages its own transactions. Up to this point we have only discussed container-managed transactions; bean-managed transactions are discussed later in this chapter.

Balancing Performance Against Consistency

Generally speaking, as the isolation levels become more restrictive, the performance of the system decreases because more restrictive isolation levels prevent transactions from accessing the same data. If isolation levels are very restrictive, like *Serializable*, then all transactions, even simple reads, must wait in line to execute. This can result in a system that is very slow. EJB systems that process a large number of concurrent transactions and need to be very fast will therefore avoid the *Serializable* isolation level where it is not necessary, since it will be prohibitively slow.

Isolation levels, however, also enforce consistency of data. More restrictive isolation levels help ensure that invalid data is not used for performing updates. The old adage "garbage in, garbage out" applies here. The *Serializable* isolation level ensures that data is never accessed concurrently by transactions, thus ensuring that the data is always consistent.

Choosing the correct isolation level requires some research about the database you are using and how it handles locking. You must also balance the performance needs of your system against consistency. This is not a cut-and-dried process, because different applications use data differently.

Although there are only three ships in Titan's system, the entity beans that represent them are included in most of Titan's transactions. This means that many, possibly hundreds, of transactions will be accessing these Ship EJBs at the same time. Access to Ship EJBs needs to be fast or it becomes a bottleneck, so we do not want to use very restrictive isolation levels. At the same time, the ship data also needs to be consistent; otherwise, hundreds of transactions will be using invalid data. Therefore, we need to use a strong isolation level when making changes to ship information. To accommodate these conflicting requirements, we can apply different isolation levels to different methods.

Most transactions use the Ship EJB's get methods to obtain information. This is *read- only* behavior, so the isolation level for the get methods can be very low, such as *Read Uncommitted*. The set methods of the Ship EJB are almost never used; the name of the ship probably wouldn't change for years. However, the data changed by the set methods must be isolated to prevent dirty reads by other transactions, so we will use the most restrictive isolation level, *Serializable*, on the ship's set methods. By using different isolation levels on different business methods, we can balance consistency against performance.

Controlling isolation levels

Different EJB servers allow different levels of granularity for setting isolation levels; some servers defer this responsibility to the database. Most EJB servers control the isolation level through the resource access API (e.g. JDBC and JMS) and may allow different resources to have different isolation levels, but will generally require that access to the same resource within a single transaction use a consistent isolation level. You will need to consult your vendor's documentation to find out the level of control your server offers.

Bean-managed transactions in session beans (stateful and stateless) and message-driven beans (EJB 2.0), however, allow the EJB developer to specify the transaction isolation level using the API of the resource providing persis tent storage. The JDBC API, for example, provides a mechanism for specifying the isolation level of the database connection. The following code shows how this is done. Bean- managed transactions are covered in more detail later in this chapter.

```
...
DataSource source = (javax.sql.DataSource)
    jndiCntxt.lookup("java:comp/env/jdbc/titanDB");

Connection con = source.getConnection();
con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
...
```

You can set the isolation level to be different for different resources within the same transaction, but all enterprise beans that use the same resource in a transaction should use the same isolation level.

Non-Transactional Beans

Beans that reside outside a transaction scope normally provide some kind of stateless service that doesn't directly manipulate data in a data store. While these types of enterprise beans may be necessary as utilities during a transaction, they do not need to meet the stringent ACID requirements of a transaction.

Consider a non-transactional stateless session bean, the QuoteBean, that provides live stock quotes. This EJB may respond to a request from a

transactional EJB involved in a stock purchase transaction. The success or failure of the stock purchase, as a transaction, will not impact the state or operations of the QuoteBean, so it doesn't need to be part of the transaction. Beans that are involved in transactions are subjected to the isolated ACID property, which means that their services *cannot* be shared during the life of the transaction. Making an enterprise bean transactional can be an expensive runtime activity. Declaring an EJB to be non-transactional (i.e., *Not Supported*) leaves it out of the transaction scope, which may improve the performance and availability of that service.

Explicit Transaction Management

> Although this section covers JTA, it is strongly recommended that you do not attempt to manage transactions explicitly. Through transaction attributes, EJB provides a comprehensive and simple mechanism for delimiting transactions at the method level and propagating transactions automatically. Only developers with a thorough understanding of transactional systems should attempt to use JTA with EJB.

In EJB, implicit transaction management is provided on the enterprise bean method level so that we can define transactions that are delimited by the scope of the method being executed. This is one of the primary advantages of EJB over cruder distributed object implementations: it reduces complexity and therefore programmer error. In addition, declarative transaction demarcation, as used in EJB, separates the transactional behavior from the business logic; a change to transactional behavior does not require changes to the business logic. In rare situations, however, it may be necessary to take control of transactions explicitly. To do this, it is necessary to have a much more complete understanding of transactions.

Explicit management of transactions is complex and is normally accomplished using the OMG's OTS (Object Transaction Service) or the Java implementation of OTS, JTS (Java Transaction Service). OTS and JTS provide APIs that allow developers to work with transaction managers and resources (e.g. databases and JMS providers) directly. While the JTS implementation of OTS is robust and complete, it is not the easiest API to work with; it requires clean and intentional control over the bounds of enrollment in transactions.

Enterprise JavaBeans supports a much simpler API, the Java Transaction API (JTA), for working with transactions. This API is implemented by the `javax.transaction` package. JTA actually consists of two components: a high-level transactional client interface and a low-level X/Open XA interface. We are concerned with the high-level client interface since that is the one accessible to the enterprise beans and is the recommended transactional interface for client

applications. The low-level XA interface is used by the EJB server and container to automatically coordinate transactions with resources like databases.

As an application and EJB developer, your use of explicit transaction management will focus on one very simple interface: `javax.transaction.UserTransaction`. `UserTransaction` provides an interface to the transaction manager that allows the application developer to manage the scope of a transaction explicitly. Here is an example of how explicit demarcation might be used in a EJB or client application:

```
Object ref = getInitialContext().lookup("TravelAgentHome");
TravelAgentHome home = (TravelAgentHome)
    PortableRemoteObject.narrow(ref, TravelAgentHome.class);

TravelAgent tr1 = home.create(customer);
tr1.setCruiseID(cruiseID);
tr1.setCabinID(cabin_1);
TravelAgent tr2 = home.create(customer);
tr2.setCruiseID(cruiseID);
tr2.setCabinID(cabin_2);

javax.transaction.UserTransaction tran = ...; // Get the UserTransaction.
tran.begin();
tr1.bookPassage(visaCard,price);
tr2.bookPassage(visaCard,price);
tran.commit();
```

The client application needs to book two cabins for the same customer—in this case, the customer is purchasing a cabin for himself and his children. The customer doesn't want to book either cabin unless he can get both, so the client application is designed to include both bookings in the same transaction. Explicitly marking the transaction's boundaries through the use of the `javax.transaction.UserTransaction` object does this. Each enterprise bean method invoked by the current thread between the `UserTransaction.begin()` and `UserTransaction.commit()` method is included in the same transaction scope, according to transaction attribute of the enterprise bean methods invoked.

Obviously this example is contrived, but the point it makes is clear. Transactions can be controlled directly, instead of depending on method scope to delimit them. The advantage of using explicit transaction demarcation is that it gives the client control over the bounds of a transaction. The client, in this case, may be a client application or another enterprise bean.[4] In either case, the same `javax.transaction.UserTransaction` is used, but it is obtained from

4 Only beans declared as managing their own transactions (bean-managed transaction beans) can use the `UserTransaction` interface.

different sources depending on whether it is needed on the client or in an enterprise bean.

Java 2 Enterprise Edition (J2EE) specifies how a client application can obtain a UserTransaction object using JNDI. Here's how a client obtains a UserTransaction object if the EJB container is part of a J2EE system (J2EE and its relationship with EJB is covered in more detail in Chapter 17):

```
...
Context jndiCntx = new InitialContext();
UserTransaction tran =
    (UserTransaction)jndiCntx.lookup("java:comp/UserTransaction");

utx.begin();
...
utx.commit();
...
```

Enterprise beans can also manage transactions explicitly. Only session beans and message-driven beans (EJB 2.0) with the `<transaction-type>` value of "`Bean`" can be managed their own transactions. Enterprise beans that manage their own transactions are frequently referred to as bean-managed transaction (BMT) beans. Entity beans can never be BMT beans. BMT beans do not declare transaction attributes for their methods. Here's how a session bean declares that it will manage transactions explicitly:

```
<ejb-jar>
 <enterprise-beans>
 ...
   <session>
   ...
   <transaction-type>Bean</transaction-type>
   ...
```

To manage its own transaction, an enterprise bean needs to obtain a `UserTransaction` object. An enterprise bean obtains a reference to the `UserTransaction` from the `EJBContext`, as shown below:

```
public class HypotheticalBean extends SessionBean {
  SessionContext ejbContext;

  public void someMethod() {
   try {
    UserTransaction ut = ejbContext.getUserTransaction();
    ut.begin();

    // Do some work.

    ut.commit();
    } catch(IllegalStateException ise) {...}
      catch(SystemException se) {...}
      catch(TransactionRolledbackException tre) {...}
```

```
    catch(HeuristicRollbackException hre) {...}
    catch(HeuristicMixedException hme) {...}
```

An enterprise bean can also access the `UserTransaction` from the JNDI ENC as shown in the following example. Both methods are legal and proper. The enterprise bean performs the lookup using the "`java:comp/env/UserTransaction`" context:

```
InitialContext jndiCntx = new InitialContext();
UserTransaction tran = (UserTransaction)
    jndiCntx.lookup("java:comp/env/UserTransaction");
```

Transaction Propagation in Bean-Managed Transactions

With stateless session beans, transactions that are managed using the `UserTransaction` must be started and completed within the same method. In other words, `UserTransaction` transactions cannot be started in one method and ended in another. This makes sense because stateless session bean instances are shared across many clients. So while one stateless instance may service a client's first request, a completely different instance may service the same client subsequent request. With stateful session beans, however, a transaction can begin in one method and be committed in another because a stateful session bean is only used by one client. This allows a stateful session bean to associate itself with a transaction across several different client- invoked methods. As an example, imagine the TravelAgent EJB as a BMT bean. In the following code, the transaction is started in the `setCruiseID()` method and completed in the `bookPassage()` method. This allows the TravelAgent EJB's methods to be associated with the same transaction.

EJB 2.0: TravelAgentBean

```
import com.titan.reservation.*;

import java.sql.*;
import javax.sql.DataSource;
import java.util.Vector;
import java.rmi.RemoteException;
import javax.naming.NamingException;
import javax.ejb.EJBException;

public class TravelAgentBean implements javax.ejb.SessionBean {

    ...
    public void setCruiseID(Integer cruiseID)
        throws javax.ejb.FinderException {
        try {
           ejbContext.getUserTransaction().begin();
           CruiseHomeLocal home = (CruiseHomeLocal)
           jndiContext.lookup("java:comp/env/ejb/CruiseHome");
```

```
            cruise = home.findByPrimaryKey(cruiseID);
        } catch(RemoteException re) {
            throw new EJBException(re);
        }

    }
    public TicketDO bookPassage(CreditCardDO card, double price)
        throws IncompleteConversationalState {

        try {
            if (ejbContext.getUserTransaction().getStatus() !=
                javax.transaction.Status.STATUS_ACTIVE) {

                throw new EJBException("Transaction is not active");
            }
        } catch(javax.transaction.SystemException se) {
             throw new EJBException(se);
        }


        if (customer == null || cruise == null || cabin == null)
        {
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeLocal resHome =
               (ReservationHomeLocal) jndiContext.lookup(
                    "java:comp/env/ejb/ReservationHome");

            ReservationLocal reservation =
                resHome.create(customer, cruise, cabin, price);

            Object ref = jndiContext.lookup
                       ("java:comp/env/ejb/ProcessPaymentHome");

            ProcessPaymentHomeRemote ppHome =
                (ProcessPaymentHomeRemote) PortableRemoteObject.narrow(
                     ref, ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket =
                   new TicketDO(customer,cruise,cabin,price);

            ejbContext.getUserTransaction().commit();

            return ticket;
        } catch(Exception e) {
```

```
            throw new EJBException(e);
        }
    }
    ...
}
```

EJB 1.1: TravelAgentBean

```
public class TravelAgentBean implements javax.ejb.SessionBean {

    ...
    public void setCruiseID(Integer cruiseID)
        throws javax.ejb.FinderException {
        try {
           ejbContext.getUserTransaction().begin();
           CruiseHomeRemote home = (CruiseHomeRemote)
           getHome("CruiseHome", CruiseHomeRemote. class);
           cruise = home.findByPrimaryKey(cruiseID);
        } catch(RemoteException re) {
            throw new EJBException(re);
        }

    }
    public TicketDO bookPassage(CreditCardDO card, double price)
        throws IncompleteConversationalState {

        try {
            if (ejbContext.getUserTransaction().getStatus() !=
                javax.transaction.Status.STATUS_ACTIVE) {

                throw new EJBException("Transaction is not active");
            }
        } catch(javax.transaction.SystemException se) {
             throw new EJBException(se);
        }


        if (customer == null || cruise == null || cabin == null) {
            throw new IncompleteConversationalState();
        }
        try {
            ReservationHomeRemote resHome =
                (ReservationHomeRemote)getHome("ReservationHome",
                                ReservationHomeRemote.class);
            ReservationRemote reservation =
                resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
                getHome("ProcessPaymentHome", ProcessPaymentHomeRemote.class);
            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);
```

```
            TicketDO ticket =
                  new TicketDO(customer,cruise,cabin,price);

            ejbContext.getUserTransaction().commit();

            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    ...
}
```

Repeated calls to the `EJBContext.getUserTransaction()` method return a reference to the same `UserTransaction` object. The container is required to retain the association between the transaction and the stateful bean instance across multiple client calls until the transaction terminates.

In the `bookPassage()` method, we can check the status of the transaction to ensure that it's still active. If the transaction is no longer active, we throw an exception. The use of the `getStatus()` method is covered in more detail later in this chapter.

When a bean-managed transaction method is invoked by a client that is already involved in a transaction, the client's transaction is suspended until the method returns. This suspension occurs whether the BMT bean explicitly starts its own transaction within the method or the transaction was started in a previous method invocation. The client transaction is always suspended until the bean-managed transaction method returns.

> Transaction control across methods is strongly discouraged because it can result in improperly managed transactions and long-lived transactions that lock up resources.

EJB 2.0: Message-driven beans and bean-managed transactions

Message-driven beans also have the option of managing their own transactions. In the case of MDBs, the scope of the transaction must begin and end within the `onMessage()` method—it's not possible for a bean-managed transaction to span `onMessage()` calls.

The ReservationProcessor EJB can be transformed to be a BMT bean, simply by changing its `<transaction-type>` value to "`Bean`".

```
<ejb-jar>
 <enterprise-beans>
 ...
   <message-driven>
```

```
...
<transaction-type>Bean</transaction-type>
...
```

In this case, the `ReservationProcessorBean` class would be modified to use the `javax.transaction.UserTransaction` to mark the beginning and end of the transaction in `onMessage()`:

```
public class ReservationProcessorBean
implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener {

    MessageDrivenContext ejbContext;
    Context jndiContext;

    public void onMessage(Message message) {
        try {

            ejbContext.getUserTransaction().begin();

            MapMessage reservationMsg = (MapMessage)message;

            Integer customerPk = (Integer)
                    reservationMsg.getObject("CustomerID");
            Integer cruisePk =   (Integer)
                    reservationMsg.getObject("CruiseID");
            Integer cabinPk =    (Integer)
                    reservationMsg.getObject("CabinID");

            double price = reservationMsg.getDouble(“Price”);

            CreditCardDO card = (CreditCardDO)
                    reservationMsg.getObject(“CreditCard”);

            CustomerLocal customer = getCustomer(customerPk);
            CruiseLocal cruise = getCruise(cruisePk);
            CabinLocal cabin = getCabin(cabinPk);

            ReservationHomeLocal resHome = (ReservationHomeLocal)
                jndiContext.lookup("java:comp/env/ejb/ReservationHome");

            ReservationLocal reservation =
                resHome.create(customer, cruise, cabin, price);

            Object ref = jndiContext.lookup
                        ("java:comp/env/ejb/ProcessPaymentHome");

            ProcessPaymentHomeRemote ppHome =
            (ProcessPaymentHomeRemote)
            PortableRemoteObject.narrow
                        (ref, ProcessPaymentHomeRemote.class);
```

```
            ProcessPaymentLocal process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);


            deliverTicket(reservationMsg, ticket);

            ejbContext.getUserTransaction().commit();

        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
    ...
}
```

It is important to understand that in BMT, the message consumed by the MDB is not a part of the transaction. When an MDB uses container-managed transactions, the message it is handling is a part of the transaction, so if the transaction is rolled back, the consumption of the message is also rolled back, forcing the JMS provider to redeliver the message. But with bean-managed transactions, the message is not a part of the transaction, so if the BMT transaction is rolled back, the JMS provider will not be aware of transaction the failure. However, all is not lost because the JMS provider can still rely on message acknowledgment to determine if the message was successfully delivered.

The EJB container will acknowledge the message if the `onMessage()` method returns successfully. If, however, a RuntimeException is thrown by the onMessage() method, the container will not acknowledge the message and the JMS provider will suspect a problem and will probably attempt to redeliver the message. If redlivery of a message is important when a transaction fails in BMT, you're best course of action is to ensure that the onMessage( ) method throws an EJBException so that the container will not acknowledge the message received from the JMS provider.

> Vendors will use proprietary (declarative) mechanisms to specify the number of times to redeliver messages to BMT/NotSupported MDBs which 'fail' to acknowledge. The JMS provider may provide a "dead message" area into which such messages would be placed if they cannot be successfully processed according to the retry count. The "dead message" area can monitored by administrators and delivered messages can be detected and handled manually."

While the message is not a part of the transaction, everything else between the `UserTransaction.begin()` and `UserTransaction.commit()` is a

part of the same transaction. This includes creating a new Reservation EJB and processing the credit card using the ProcessPayment EJB. If a transaction failure occurs, these operations will be rolled back. The transaction also includes the use of the JMS API in the `deliverTicket()` method to send the ticket message. If a transaction failure occurs, the ticket message will not be sent.

Heuristic Decisions

Transactions are normally controlled by a *transaction manager* (often the EJB server) that manages the ACID characteristics across several enterprise beans, databases, and servers. This transaction manager uses a *two-phase commit* (2-PC) to manage transactions. 2-PC is a protocol for managing transactions that commits updates in two stages. 2-PC is complex and outside the scope of this book, but basically it requires that servers and databases cooperate through an intermediary, the transaction manager, to ensure that all the data is made durable together. Some EJB servers support 2-PC while others don't, and the value of this transaction mechanism is a source of some debate. The important point to remember is that a transaction manager controls the transaction; based on the results of a poll against the resources (databases, JMS providers, and other resources), it decides whether all the updates should be committed or rolled back. A *heuristic decision* is when one of the resources makes a unilateral decision to commit or roll back without permission from the transaction manager. Once a heuristic decision has been made, the atomicity of the transaction is lost and possible data integrity errors can occur.

`UserTransaction`, discussed in the next section, throws a couple of different exceptions related to heuristic decisions; these are included in the following discussion.

UserTransaction

`UserTransaction` is a Java interface that is defined in the following code. EJB servers are not required to support the rest of JTA, nor are they required to use JTS for their transaction service. The `UserTransaction` is defined as follows:

```
public interface javax.transaction.UserTransaction
{
    public abstract void begin()
       throws IllegalStateException, SystemException;
    public abstract void commit()
        throws IllegalStateException, SystemException,
               TransactionRolledbackException,
               HeuristicRollbackException, HeuristicMixedException;
    public abstract int getStatus();
    public abstract void rollback()
       throws IllegalStateException, SecurityException, SystemException;
    public abstract void setRollbackOnly()
```

```
        throws IllegalStateException, SystemException;
    public abstract void setTransactionTimeout(int seconds)
        throws SystemException;
}
```

Here's what the methods defined in this interface do:

`begin()`

Invoking the `begin()` method creates a new transaction. The thread that executes the `begin()` method is immediately associated with the new transaction. The transaction is propagated to any EJB that supports existing transactions. The `begin()` method can throw one of two checked exceptions. IllegalStateException is thrown when `begin()` is called by a thread that is already associated with a transaction. You must complete any transactions associated with that thread before beginning a new transaction. `SystemException` is thrown if the transaction manager (the EJB server) encounters an unexpected error condition.

`commit()`

The `commit()` method completes the transaction that is associated with the current thread. When `commit()` is executed, the current thread is no longer associated with a transaction. This method can throw several checked exceptions. `IllegalStateException` is thrown if the current thread is not associated with a transaction. `SystemException` is thrown if the transaction manager (the EJB server) encounters an unexpected error condition. `TransactionRolledbackException` is thrown when the entire transaction is rolled back instead of committed; this can happen if one of the resources was unable to perform an update or if the `UserTransaction.rollBackOnly()` method was called. `HeuristicRollbackException` indicates that heuristic decisions were made by one or more resources to roll back the transaction. `HeuristicMixedException` indicates that heuristic decisions were made by resources to both roll back and commit the transaction; some resources decided to roll back while others decided to commit.

`rollback()`

The `rollback()` method is invoked to roll back the transaction and undo updates. The `rollback()` method can throw one of three different checked exceptions. `SecurityException` is thrown if the thread using the `UserTransaction` object is not allowed to roll back the transaction. `IllegalStateException` is thrown if the current thread is not associated with a transaction. `SystemException` is thrown if the transaction manager (the EJB server) encounters an unexpected error condition.

`setRollBackOnly()`

This method is invoked to mark the transaction for rollback. This means that, whether or not the updates executed within the transaction succeed, the transaction must be rolled back when completed. This method can be

invoked by any `TX_BEAN_MANAGED` EJB participating in the transaction or by the client application. The `setRollBackOnly()` method can throw one of two different checked exceptions. `IllegalStateException` is thrown if the current thread is not associated with a transaction. `SystemException` is thrown if the transaction manager (the EJB server) encounters an unexpected error condition.

`setTransactionTimeout(int seconds)`

This method sets the life span of a transaction: how long it will live before timing out. The transaction must complete before the transaction timeout is reached. If this method is not called, the transaction manager (EJB server) automatically sets the timeout. If this method is invoked with a value of 0 seconds, the default timeout of the transaction manager will be used. This method must be invoked after the `begin()` method. `SystemException` is thrown if the transaction manager (EJB server) encounters an unexpected error condition.

`getStatus()`

The `getStatus()` method returns an integer that can be compared to constants defined in the `javax.transaction.Status` interface. This method can be used by a sophisticated programmer to determine the status of a transaction associated with a `UserTransaction` object. `SystemException` is thrown if the transaction manager (EJB server) encounters an unexpected error condition.

Status

`Status` is a simple interface that contains no methods, only constants. Its sole purpose is to provide a set of constants that describe the current status of a transactional object— in this case, the `UserTransaction`:

```
interface javax.transaction.Status
{
   public final static int STATUS_ACTIVE;
   public final static int STATUS_COMMITTED;
   public final static int STATUS_COMMITTING;
   public final static int STATUS_MARKED_ROLLBACK;
   public final static int STATUS_NO_TRANSACTION;
   public final static int STATUS_PREPARED;
   public final static int STATUS_PREPARING;
   public final static int STATUS_ROLLEDBACK;
   public final static int STATUS_ROLLING_BACK;
   public final static int STATUS_UNKNOWN;
}
```

The value returned by `getStatus()` tells the client using the `UserTransaction` the status of a transaction. Here's what the constants mean:

`STATUS_ACTIVE`

An active transaction is associated with the `UserTransaction` object. This status is returned after a transaction has been started and prior to a transaction manager beginning a 2-PC commit. (Transactions that have been suspended are still considered active.)

`STATUS_COMMITTED`

A transaction is associated with the `UserTransaction` object; the transaction has been committed. It is likely that heuristic decisions have been made; otherwise, the transaction would have been destroyed and the `STATUS_NO_TRANSACTION` constant would have been returned instead.

`STATUS_COMMITTING`

A transaction is associated with the `UserTransaction` object; the transaction is in the process of committing. The `UserTransaction` object returns this status if the transaction manager has decided to commit but has not yet completed the process.

`STATUS_MARKED_ROLLBACK`

A transaction is associated with the `UserTransaction` object; the transaction has been marked for rollback, perhaps as a result of a `UserTransaction.setRollbackOnly()` operation invoked somewhere else in the application.

`STATUS_NO_TRANSACTION`

No transaction is currently associated with the `UserTransaction` object. This occurs after a transaction has completed or if no transaction has been created. This value is returned rather than throwing an `IllegalStateException`.

`STATUS_PREPARED`

A transaction is associated with the `UserTransaction` object. The transaction has been prepared, which means that the first phase of the two-phase commit process has completed.

`STATUS_PREPARING`

A transaction is associated with the `UserTransaction` object; the transaction is in the process of preparing, which means that the transaction manager is in the middle of executing the first phase of the two-phase commit.

`STATUS_ROLLEDBACK`

A transaction is associated with the `UserTransaction` object; the outcome of the transaction has been identified as a rollback. It is likely that heuristic decisions have been made; otherwise, the transaction would have been destroyed and the `STATUS_NO_TRANSACTION` constant would have been returned.

`STATUS_ROLLING_BACK`

A transaction is associated with the `UserTransaction` object; the transaction is in the process of rolling back.

`STATUS_UNKNOWN`

A transaction is associated with the `UserTransaction` object; its current status cannot be determined. This is a transient condition and subsequent invocations will ultimately return a different status.

EJBContext Rollback Methods

Only BMT beans have access to the `UserTransaction` from the `EJBContext` and JNDI ENC. Enterprise beans that manager their own transactions, container-managed transaction (CMT) beans, can not use the `UserTransaction`. CMT beans use the `setRollbackOnly()` and `getRollbackOnly()`methods of the `EJBContext` to interact with the current transaction.

The `setRollbackOnly()` method gives an enterprise bean the power to veto a transaction. This power can be used if the enterprise bean detects a condition that would cause inconsistent data to be committed when the transaction completes. Once an enterprise bean invokes the `setRollback-Only()` method, the current transaction is marked for rollback and cannot be committed by any other participant in the transaction—including the container.

The `getRollbackOnly()` method returns `true` if the current transaction has been marked for rollback. This can be used to avoid executing work that wouldn't be committed anyway. If, for example, an exception is thrown and captured within an enterprise bean method, `getRollbackOnly()`can be used to determine whether the exception caused the current transaction to be rolled back. If it did, there is no sense in continuing the processing. If it didn't, the EJB has an opportunity to correct the problem and retry the task that failed. Only expert EJB developers should attempt to retry tasks within a transaction. Alternatively, if the exception didn't cause a rollback (`getRollbackOnly()` returns `false`), a rollback can be forced using the `setRollbackOnly()` method.

BMT beans must *not* use the `setRollbackOnly()` and `getRollbackOnly()` methods of the `EJBContext`. BMT beans should use the `getStatus()` and `rollback()` methods on the `UserTransaction` object to check for rollback and force a rollback respectively.

Exceptions and Transactions

Application Exceptions Versus System Exceptions

An application exception is any exception that does *not* extend `java.lang.RuntimeException` or the

`java.rmi.RemoteException`. System exceptions are `java.lang.RuntimeException` and its subtypes, including `EJBException`.

> An application exception must never extend either the `RuntimeException`, the `RemoteException`, or one of their subtypes.

Transactions are *automatically* rolled back if a system exception is thrown from an enterprise bean method. Transactions are *not automatically* rolled back if an application exception is thrown. If you remember these two rules, you will be well prepared to deal with exceptions and transactions in EJB.

The `bookPassage()` method provides a good illustration of an application exception and how it's used. The following code shows the `bookPassage()` method:

EJB 2.0: bookPassage() method

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome =
        (ReservationHomeLocal)
        jndiContext.lookup
                    ("java:comp/env/ejb/ReservationHome");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);

        Object ref = jndiContext.lookup
                    ("java:comp/env/ejb/ProcessPaymentHome");

        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
        PortableRemoteObject.narrow
                    (ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
```

```
        throw new EJBException(e);
    }
}
```

EJB 1.1: bookPassage() method

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {


    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeRemote resHome =
        (ReservationHomeRemote)getHome("ReservationHome",
                                ReservationHomeRemote.class);
        ReservationRemote reservation =
            resHome.create(customer, cruise, cabin, price);
        ProcessPaymentHomeRemote ppHome =
        (ProcessPaymentHomeRemote)
                    getHome("ProcessPaymentHome",
                            ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket =
                new TicketDO(customer,cruise,cabin,price);
        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

System exceptions

System exceptions are the `RuntimeException` and its subclasses. The `EJBException` is a subclass of the `RuntimeException`, so it's considered a system exception.

System exceptions always cause a transaction to roll back when thrown from a enterprise bean method. Any `RuntimeException` (`EJBException`, `NullPointerException`, `IndexOutOfBoundsException`, etc.) thrown within the `bookPassage()` method is handled by the container automatically, and also results in a transaction rollback. In Java, `RuntimeException` types do not need to be declared in the `throws` clause of the method signature or handled using `try`/`catch` blocks; they are automatically thrown from the method.

`RuntimeException` types thrown from within enterprise beans always cause the current transaction to roll back. If the method in which the exception occurs started the transaction, the transaction is rolled back. If the transaction started from a client that invoked the method, the client's transaction is marked for rollback and cannot be committed.

System exceptions are handled automatically by the container, which will always:

- Roll back the transaction
- Log the exception to alert the system administrator
- Discard the EJB instance

`RuntimeExceptions` thrown from the callback methods (`ejbLoad()`, `ejbActivate()`, etc.) are treated the same as exceptions thrown from business methods.

While EJB requires that system exceptions be logged, it does not specify how exceptions should be logged or the format of the log file. The exact mechanism for recording the exception and reporting it to the system administrator is left to the vendor.

When a system exception occurs, the EJB instance is discarded, which means that it's dereferenced and garbage collected. The container assumes that the EJB instance may have corrupt variables or otherwise be unstable, and is therefore unsafe to use.

The impact of discarding an EJB instance depends on the enterprise beans's type. In the case of stateless session beans and entity beans, the client does not notice that the instance was discarded. These types are not dedicated to a particular client; they are swapped in and out of an instance pool, so any instance can service a new request. With stateful session beans, however, the impact on the client is severe. Stateful session beans are dedicated to a single client and maintain conversational state. Discarding a stateful bean instance destroys the instance's conversation state and invalidates the client's reference to the EJB. When stateful session instances are discarded, subsequent invocations of the EJB's methods by the client result in a `NoSuchObjectException`, a subclass of the `RemoteException`.[5]

With message-driven beans a system exception thrown by the `onMessage()` method or one of the callback methods (`ejbCreate()` or `ejbRemove()`) will cause the bean instance to be discarded. If the MDB was BMT bean, then the message it was handling may or may not be redelivered depending on when the EJB container acknowledges delivery. In the case of container-managed

5 Although the instance is always discarded with a `RuntimeException`, the impact on the remote reference may vary depending on the vendor.

transactions, the container will rollback the transaction, so the message will not be acknowledged and may be redelivered.

In session and entity beans, a system exception occurs and the instance is discarded, a `RemoteException` is always thrown to remote clients; clients using the beans remote component interfaces. If the client started the transaction, which was then propagated to the EJB, a system exception (thrown by the enterprise bean method) will be caught by the container and rethrown as a `javax.transaction.TransactionRolledbackException`. The `TransactionRolledbackException` is a subtype of the `RemoteException`; it's a more explicit indication to the client that a rollback occurred. In all other cases, whether the EJB is container-managed or bean-managed, a `RuntimeException` thrown from within the enterprise bean method will be caught by the container and rethrown as a `EJBException`. A system exception always results in a rollback of the transaction.

In EJB 2.0 session and entity beans, when a system exception occurs and the instance is discarded, an `EJBException` is always thrown to any local enterprise bean clients (clients using the enterprise bean's local component interfaces). If the client started the transaction, which was then propagated to the EJB, a system exception (thrown by the enterprise bean method) will be caught by the container and rethrown as a `javax.ejb.TransactionRolledbackLocalException`. The `TransactionRolledbackLocalException` is a subtype of the `EJBException`; it's a more explicit indication to the client that a rollback occurred. In all other cases, whether the EJB is container-managed or bean-managed, a `RuntimeException` thrown from within the enterprise bean method will be caught by the container and rethrown as an `EJBException`. A system exception always results in a rollback of the transaction.

An `EJBException` should be thrown, in most cases, when a subsystem throws an exception such as JDBC throwing a `SQLException` or JMS throwing a `JMSException`. In some cases the bean developer may attempt to handle the exception and retry an operation rather then throw an `EJBException`. This should only be done when the exceptions thrown by the subsystem and their repurcussions on the transaction are well understood. As a rule of thumb, throw subsystem exceptions as `EJBExceptions` and allow the EJB container to rollback the transaction and discard the bean instance.

The callback methods defined in the `javax.ejb.EntityBean` and `javax.ejb.SessionBean` interfaces declare the `java.rmi.RemoteException` in their `throws` clause. This is left over from EJB 1.0, which has been deprecated since EJB 1.1. You should never throw `RemoteException`s from callback methods, or any other bean class methods.

Application exceptions

An application exception is normally thrown in response to a business logic error, as opposed to a system error. They are always delivered directly to the client, without being repackaged as `RemoteException` or `EJBException` (EJB 2.0) types. They do not typically cause transactions to roll back; the client usually has an opportunity to recover after an application exception is thrown. For example, the `bookPassage()` method throws an application exception called `IncompleteConversationalState`; this is an application exception because it does not extend `RuntimeException` or `RemoteException`. The `IncompleteConversationalState` exception is thrown if one of the arguments passed into the `bookPassage()` method is `null`. (Application errors are frequently used to report validation errors like this.) In this case, the exception is thrown before tasks are started, and is clearly not the result of a subsystem (JDBC, JMS, Java RMI, JNDI, etc.) failure.

Because it is an application exception, throwing `IncompleteConversationalState` does not result in a transaction rollback. The exception is thrown before any work is done, avoiding unnecessary processing by the `bookPassage()` method and providing the client (the enterprise bean or application that invoked the `bookPassage()` method) with an opportunity to recover and possibly retry the method call with valid arguments.

Business methods defined in the remote and local interfaces can throw any kind of application exception. These application exceptions must be declared in the method signatures of the remote and local interfaces and in the corresponding method in the Enterprise EJB class.

The EJB create, find, and remove methods can also throw several exceptions defined in the `javax.ejb` package: `CreateException`, `DuplicateKeyException`, `FinderException`, `ObjectNotFoundException`, and `RemoveException`. These exceptions are also considered application exceptions: they are delivered to the client as is, without being repackaged as `RemoteExceptions`. Furthermore, these exceptions don't necessarily cause a transaction to roll back, giving the client the opportunity to retry the operation. These exceptions may be thrown by the EJBs themselves; in the case of container-managed persistence (CMP), the container can also throw any of these exceptions while handling the EJB's create, find, or remove methods (`ejbCreate()`, `ejbFind...()`, and `ejbRemove()`). The container might, for example, throw a `CreateException` if the container encounters a bad argument while attempting to insert a record for a container-managed EJB. You can always choose to throw a standard application exception from the appropriate method regardless of how persistence is managed.

Here is a detailed explanation of the five standard application exceptions and the situations in which they are thrown:

`CreateException`

The `CreateException` is thrown by the `create()` method in the remote interface. This exception can be thrown by the container if the container is managing persistence, or it can be thrown explicitly by the EJB developer in the `ejbCreate()` or `ejbPostCreate()` methods. This exception indicates that an application error has occurred (invalid arguments, etc.) while the EJB was being created. If the container throws this exception, it may or may not roll back the transaction. Explicit transaction methods must be used to determine the outcome. Bean developers should roll back the transaction before throwing this exception only if data integrity is a concern.

`DuplicateKeyException`

The `DuplicateKeyException` is an subtype of the `CreateException`; it is thrown by the `create()` method in the remote interface. This exception can be thrown by the container, if the container is managing persistence, or it can be thrown explicitly by the EJB developer in the `ejbCreate()` method. This exception indicates that an EJB with the same primary key already exists in the database. The transaction is typically *not* rolled back by the EJB provider or container before throwing this exception.

`FinderException`

The `FinderException` is thrown by the find methods in the home interface. This exception can be thrown by the container, if the container is managing persistence, or it can be thrown explicitly by the EJB developer in the `ejbFind...()` methods. This exception indicates that an application error occurred (invalid arguments, etc.) while the container attempted to find the EJBs. Do not use this method to indicate that entities were not found. Multi-entity find methods return an empty collection if no entities were found; single-entity find methods throw an `ObjectNotFoundException` to indicate that no object was found. The transaction is typically not rolled back by the EJB provider or container before throwing this exception.

`ObjectNotFoundException`

The `ObjectNotFoundException` is thrown from a single-entity find method to indicate that the container couldn't find the requested entity. This exception can be thrown by the container if the container is managing persistence, or it can be thrown explicitly by the EJB developer in the `ejbFind...()` methods. This exception should not be thrown to indicate a business logic error (invalid arguments, etc.). Use the `FinderException` to indicate business logic errors in single-entity find methods. The `ObjectNotFoundException` is only thrown by single-entity find methods to indicate that the entity requested was not found. Find methods that return multiple entities should return an empty collection if nothing is found. The transaction is typically not rolled back by the EJB provider or container before throwing this exception.

RemoveException

The RemoveException is thrown from the remove() methods in the remote and home interfaces. This exception can be thrown by the container, if the container is managing persistence, or it can be thrown explicitly by the EJB developer in the ejbRemove() method. This exception indicates that an application error has occurred while the EJB was being removed. The transaction may or may not have been rolled back by the container before throwing this exception. Explicit transaction methods must be used to determine the outcome. Bean developers should roll back the transaction before throwing the exception only if data integrity is a concern.

Table 14-1 summarizes the interactions between different types of exceptions and transactions in session and entity beams.

Table 14-1: Exception Summary for Session and Entity beans

Transaction Scope	Transaction Type Attributes	Exception Thrown	Container's Action	Clier
Client Initiated Transaction Transaction is started by the client (application or EJB) and is propagated to the enterprise bean method.	transaction-type = Container transaction-attribute = Required | Mandatory | Supports |	Application Exception	If the EJB invoked setRollbackOnly(), then mark the client's transaction for rollback. Rethrow the Application Exception.	Rece Excep transa have
		System Exception	Mark the client's transaction for rollback. Log the error. Discard the instance. Rethrow the JTA TransactionRollbackException to remote clients or the javax.ejb.TransactionRollbackLocalException to EJB 2.0 local clients.	Remo JTA Roll local jav ion cep The c been
Container Initiated Transaction The transaction started when the EJB's method was invoked and will end when method completes.	transaction-type = Container transaction-attribute = Required | RequiresNew	Application Exception	If the EJB method called setRollbackOnly(), then roll back the transaction and rethrow the Application Exception. If the EJB didn't explicitly roll back the transaction, then attempt to commit the transaction and rethrow the Application Exception.	Rece Excep transa have client affect

		System Exception	Roll back the transaction. Log the error. Discard the instance. Rethrow `RemoteException` to remote clients or the `EJBException` to EJB 2.0 local clients.	Remo Remo EJB EJB The rollec The affec
Bean is not part of a transaction The EJB was invoked but does not propagate the client's transaction and does not start its own transac tion.	transaction-type = Container transaction-attribute = Never | NotSupported | Supports |	Application Exception	Rethrow the Application Exception.	Rece Excep The affec
		System Exception	Log the error. Discard the instance. Rethrow `RemoteException` to remote clients or the `EJBException` to EJB 2.0 local clients.	Remo Remo EJB EJB The affec
Bean Managed Transaction. The stateful or stateless session EJB uses the `EJBContext` to explicitly manage its own transac tion	transaction-type = Bean transaction-attribute = Bean-managed transaction EJBs do not use transaction attributes.	Application Exception	Rethrow the Application Exception.	Rece Excep The affec
		System Exception	Roll back the transaction. Log the error. Discard the instance. Rethrow `RemoteException` to remote clients or the `EJBException` to EJB 2.0 local clients.	Remo Remo EJB EJB The affec

Table 14-2 summarizes the interactions between different types of exceptions and transactions in message-driven beans.

Table 14-2: Exception Summary for Message-Driven beans

Transaction Scope	Transaction Type Attributes	Exception Thrown	Container's Action

Container Initiated Transaction The transaction started before the `onMessage()` method was invoked and will end when method completes.	transaction-type = Container transaction-attribute = Required	System Exception	Roll back the transaction. Log the error. Discard the instance.
Container Initiated Transaction No-transaction was started.	transaction-type = Container transaction-attribute = NotSupported	System Exception	Log the error. Discard the instance.
Bean Managed Transaction. The message-driven bean uses the `EJBContext` to explicitly manage its own transaction	transaction-type = Bean transaction-attribute = Bean-managed transaction EJBs do not use transaction attributes.	System Exception	Roll back the transaction. Log the error. Discard the instance.

Transactional Stateful Session Beans

As you saw in Chapter 12, session beans can interact directly with the database as easily as they can manage the workflow of other enterprise beans. The ProcessPayment EJB, for example, makes inserts into the `PAYMENT` table when the `byCredit()` method is invoked. The TravelAgent EJB queries the database directly when the `listAvailableCabins()` method is invoked. With stateless session beans like ProcessPayment, there is no conversational state, so each method invocation must make changes to the database immediately. With stateful session beans, however, we may not want to make changes to the database until the transaction is complete. Remember, a stateful session bean can be just one participant out of many in a transaction, so it may be advisable to postpone database updates until the entire transaction is committed or to avoid updates if it's rolled back.

There are several different scenarios in which a stateful session bean would want to cache changes before applying them to the database. For example, think of a shopping cart implemented by a stateful session bean that accumulates several items for purchase. If the stateful bean implements `SessionSynchronization`, it can cache the items and only write them to the database when the transaction is complete.

The `javax.ejb.SessionSynchronization` interface allows a session bean to receive additional notification of the session's involvement in transactions. The addition of these transaction callback methods by the `SessionSynchronization` interface expands the EJB's awareness of its life cycle to include a new state, the *Transactional Method-Ready state*. This third state, although not discussed in Chapter 12, is always a part of the life cycle of a transactional stateful session bean. Implementing the `SessionSynchronization` interface simply makes it visible to the EJB. Figure 14-11 shows the stateful session bean with the additional state in EJB.

[FIGURE (use modifed 8-12)]

Figure 14-11: Life cycle of a stateful session bean

The `SessionSynchronization` interface has the following definition:

```
package javax.ejb;

public interface javax.ejb.SessionSynchronization {
    public abstract void afterBegin() throws RemoteException;
    public abstract void beforeCompletion() throws RemoteException;
    public abstract void afterCompletion(boolean committed)
        throws RemoteException;
}
```

When a method of the SessionSynchronization bean is invoked outside of a transaction scope, the method executes in the Method-Ready state as discussed in Chapter 12. However, when a method is invoked within a transaction scope (or creates a new transaction), the EJB moves into the Transactional Method-Ready state.

The Transactional Method-Ready State

Transitioning into the Transactional Method-Ready state

When a transactional method is invoked on a `SessionSynchronization` bean, the stateful bean becomes part of the transaction. This causes the `afterBegin()` callback method defined in the `SessionSynchronization` interface to be invoked. This method should take care of reading any data from the database and storing the data in the bean's instance fields. The `afterBegin()` method is called before the EJB object delegates the business method invocation to the EJB instance.

Life in the Transactional Method-Ready state

When the `afterBegin()` callback method is done, the business method originally invoked by the client is executed on the EJB instance. Any subsequent business methods invoked within the same transaction will be delegated directly to the EJB instance.

Once a stateful session bean is a part of a transaction—whether it implements `SessionSynchronization` or not—it cannot be accessed by any other transactional context. This is true regardless of whether the client tries to access the EJB with a different context or the EJB's own method creates a new context. If, for example, a method with a transaction attribute of *RequiresNew* is invoked, the new transactional context causes an error to be thrown. Since the attributes *NotSupported* and *Never* simply a different transactional context (no context), invoking a method with these attributes also causes an error. A stateful session bean cannot be removed while it is involved in a transaction. This means that invoking `ejbRemove()` while the SessionSynchronization bean is in the middle of a transaction will cause an error to be thrown.

At some point, the transaction in which the SessionSynchronization bean has been enrolled will come to an end. If the transaction is committed, the SessionSynchronization bean will be notified through its `beforeCompletion()` method. At this time, the EJB should write its cached data to the database. If the transaction is rolled back, the `beforeCompletion()` method will not be invoked, avoiding the pointless effort of writing changes that won't be committed to the database.

The `afterCompletion()` method is always invoked, whether the transaction ended successfully with a commit or unsuccessfully with a rollback. If the transaction was a success—which means that `beforeCompletion()` was invoked—the committed parameter of the `afterCompletion()` method will be `true`. If the transaction was unsuccessful, `committed` will be `false`.

It may be desirable to reset the stateful session bean's instance variables to some initial state if the `afterCompletion()` method indicates that the transaction was rolled back.

15

Design Strategies

The previous fourteen chapters have presented the core EJB technology. What's left is a grab bag of miscellaneous issues: how do you solve particular design problems, how do you work with particular kinds of databases, and topics of that nature.

Hash Codes in Compound Primary Keys

Chapter 11 discusses the necessity of overriding the `Object.hashCode()` and `Object.equals()` methods in the primary key class of entity beans. With complex primary keys that have several fields, overriding the `Object.equals()` method is fairly trivial. However, the `Object.hashCode()` method is more complicated because an integer value that can serve as a suitable hash code must be created from several fields.

One solution is to concatenate all the values into a `String` and use the `String` object's `hashCode()` method to create a hash code value for the whole primary key. The `String` class has a decent hash code algorithm that generates a fairly well distributed and repeatable hash code value from any set of characters. The following code shows how to create such a hash code for a hypothetical primary key:

```
public class HypotheticalPrimaryKey implements java.io.Serializable {
    public int primary_id;
    public short secondary_id;
    public java.util.Date date;
    public String desc;
```



```
    public int hashCode() {

        StringBuffer strBuff = new StringBuffer();
        strBuff.append(primary_id);
        strBuff.append(secondary_id);
        strBuff.append(date);
        strBuff.append(desc);
        String str = strBuff.toString();
        int hashCode = str.hashCode();
        return hashCode;
    }
    // the constructor, equals, and toString methods follow
}
```

A `StringBuffer` cuts down on the number of objects created, since `String` concatenation is expensive. The code could be improved by saving the hash code in a private variable and returning that value in subsequent method calls; this way, the hash code is only calculated once in the life of the instance.

Well-Distributed Versus Unique Hash Codes

A `Hashtable` is designed to provide fast lookups by binding an object to a key. Given any object's key, looking the object up in a hash table is a very quick operation. For the lookup, the key is converted to an integer value using the key's `hashCode()` method.

Hash codes do not need to be unique, only well-distributed. By "well-distributed," we mean that given any two keys, the chances are very good that the hash codes for the keys will be different. A well-distributed hash code algorithm reduces, but does not eliminate, the possibility that different keys evaluate to the same hash code. When keys evaluate to the same hash code, they are stored together and uniquely identified by their `equals()` method. If you look up an object using a key that evaluates to a hash code that is shared by several other keys, the `Hashtable` locates the group of objects that have been stored with the same hash code; then it uses the key's `equals()` method to determine which key (and hence, which object) you want. (That's why you have to override the `equals()` method in primary keys, as well as the `hashCode()` method.) Therefore, the emphasis in designing a good hash code algorithm is on producing codes that are well-distributed rather than unique. This allows you to design an index for associating keys with objects that is easy to compute, and therefore fast.

Passing Objects by Value

Passing objects by value is tricky with Enterprise JavaBeans. Two simple rules will keep you out of most problem areas: objects that are passed by value should be fine-grained Dependent Objects or wrappers used in bulk accessors, and dependent objects should be immutable.

EJB 1.1: Dependent Objects

The concept of dependent objects was addressed in Chapter 6, which describes the use of dependent objects in EJB 2.0. But for EJB 1.1, dependent objects are a new concept. EJB 2.0 and EJB 1.1 use dependent objects differently, because EJB 2.0 can accommodate much finer-grained entity beans than EJB 1.1.

Dependent objects are objects that only have meaning within the context of another business object. They typically represent fairly fine-grained business concepts, like an address, phone number, or order item. For example, an address has little meaning when it is not associated with a business object like `Person` or `Organization`. It depends on the context of the business object to give it meaning. Such an object can be thought of as a wrapper for related data. The fields that make up an address (street, city, state, and Zip) should be packaged together in a single object called `AddressDO`. In turn, the `AddressDO` object is usually an attribute or property of another business object; in EJB, we would typically see an `AddressDO` or some other dependent object as a property of an entity bean.

Here's a typical implementation of an `AddressDO`:

```
public class AddressDO implements java.io.Serializable {

    private String street;
    private String city;
    private String state;
    private String zip;

    public Address(String str, String cty, String st, String zp) {
        street = str;
        city = cty;
        state = st;
        zip = zp;
    }
    public String getStreet() {return street;}
    public String getCity() {return city;}
    public String getState() {return state;}
    public String getZip() {return zip;}
}
```


We want to make sure that clients don't change an `AddressDO`'s fields. The reason is quite simple: the `AddressDO` object is a copy, not a remote reference. Changes to `AddressDO` objects are not reflected in the entity from which it originated. If the client were to change the `AddressDO` object, those changes would not be reflected in the database. Making the `AddressDO` immutable helps to ensure that clients do not mistake this fine-grained object for a remote reference, thinking that a change to an address property is reflected on the server.

To change an address, the client is required to remove the `AddressDO` object and add a new one with the changes. This enforces the idea that the dependent object is not a remote object and that changes to its state are not reflected on the server. Here is the remote interface to a hypothetical Employee bean that aggregates address information:

```
public interface Employee extends javax.ejb.EJBObject {
    public AddressDO [] getAddresses() throws RemoteException;
    public void removeAddress(AddressDO adrs) throws RemoteException;
    public void addAddress(AddressDO adrs) throws RemoteException;
    // ... Other business methods follow.
}
```

In this interface, the `Employee` can have many addresses, which are obtained as a collection of pass-by-value `AddressDO` objects. To remove an address, the target `AddressDO` is passed back to the bean in the `removeAddress()` method. The bean class then removes the matching `AddressDO` object from its persistent fields. To add an address, an `AddressDO` object is passed to the bean by value.

Dependent Objects may be persistent fields, or they may be properties that are created as needed. The following code demonstrates both strategies using the `AddressDO` object. In the first listing, the `AddressDO` object is a persistent field, while in the second the `AddressDO` object is a property that doesn't correspond to any single field; we create the `AddressDO` object as needed but don't save it as part of the bean. Instead, the `AddressDO` object corresponds to four persistent fields: `street`, `city`, `state`, and `zip`.

```
// Address as a persistent field
public class Person extends javax.ejb.EntityBean {
   public AddressDO address;
   public AddressDO getAddress(){
       return address;
   }
   public void setAddress(AddressDO addr){
       address = addr;
   }
  ....
}
```



```
// Address as a property
public class Person extends javax.ejb.EntityBean {

   public String street;
   public String city;
   public String state;
   public String zip;

   public AddressDO getAddress(){
       return new AddressDO(street, city, state, zip);
   }
   public void setAddress(AddressDO addr){
       street = addr.street;
       city = addr.city;
       state = addr.state;
       zip = addr.zip;
   }
  ....
}
```

When a dependent object is used as a property, it can be synchronized with the persistent fields in the accessor methods themselves or in the `ejbLoad()` and `ejbStore()` methods. Both strategies are acceptable.

This discussion of dependent objects has been full of generalizations, and thus may not be applicable to all situations. That said, it is recommended that only very fine-grained, dependent, immutable objects should be passed by value. All other business concepts should be represented as beans—entity or session. A very fine-grained object is one that has very little behavior, consisting mostly of get and set methods. A dependent object is one that has little meaning outside the context of its aggregator. An immutable object is one that provides only get methods and thus cannot be modified once created.

Validation Rules in Dependent Objects

Dependent Objects make excellent homes for format validation rules. Format validation ensures that a simple data construct adheres to a predetermined structure or form. As an example, a Zip Code always has a certain format. It must be composed of digits; it must be five or nine digits in length; and if it has nine digits, it must use a hyphen as a separator between the fifth and sixth digits. Checking to see that a Zip Code follows these rules is format validation.

One problem that all developers face is deciding where to put validation code. Should data be validated at the user interface (UI), or should it be done by the bean that uses the data? Validating the data at the UI has the advantage of conserving network resources and improving performance. Validating data in the bean, on the middle tier, ensures that the logic is reusable across user interfaces. Dependent objects provide a logical compromise that allows data to be validated on the client, but remain independent of the UI. By placing the validation logic in

the constructor of a dependent object, the object automatically validates data when it is created. When data is entered at the UI (GUI, Servlet, JSP, or whatever) it can be validated by the UI using its corresponding dependent object. If the data is valid, the dependent object is created; if the data is invalid, the constructor throws an exception.

The following code shows a dependent object that represents a Zip Code. It adheres to the rules for a dependent object as I have defined them, and also includes format validation rules in the constructor.

```
public class ZipCodeDO implements java.io.Serializable {

   private String code;
   private String boxNumber;

   public ZipCode(String zipcode) throws ValidationException
{
      if (zipcode == null)
          throw new ValidationException("Zip code cannot be null");
      else if (zipcode.length()==5 && ! isDigits(zipcode))
          throw new ValidationException("Zip code must be all digits");
      else if (zipcode.length()==10 )
          if (zipcode.charAt(5) == '-' ) {
            code = zipcode.substring(0,5);
if (isDigits( code )){
              boxNumber = zipcode.substring(6);
                if (isDigits( boxNumber ))
                    return;
             }
          }
          throw new ValidationException("Zip code must be of form #####- ####");
   }
   private boolean isDigits(String str) {
      for (int i = 0; i < str.length(); i++){
          char chr = str.charAt(i);
          if ( ! Character.isDigit(chr)) {
              return false;
          }
      }
      return true;
   }
   public String getCode() { return code; }

   public String getBoxNumber() { return boxNumber; }

   public String toString() {
      return code+'-'+boxNumber;
   }
}
```


This simple example illustrates that format validation can be performed by dependent objects when the object is constructed at the user interface or client. Any format validation errors are reported immediately, without requiring any interaction with the middle tier of the application. In addition, any business object that uses `ZipCodeDO` automatically gains the benefit of the validation code, making the validation rules reusable (and consistent) across beans. Placing format validation in the dependent object is also a good coding practice because it makes the dependent object responsible for its own validation; responsibility is a key concept in object-oriented programming. Of course, dependent objects are only useful for validation if the Enterprise JavaBeans implementation supports pass-by-value.

As an alternative to using Dependent Objects, format validation can be performed by the accessors of enterprise beans. If, for example, a customer bean has accessors for setting and obtaining the Zip Code, the accessors could incorporate the validation code. While this is more efficient from a network perspective—passing a `String` value is more efficient than passing a dependent object by value—it is less reusable than housing format validation rules in dependent objects.

Bulk Accessors

Most entity beans have several persistent fields that are manipulated through accessor methods. Unfortunately, the one-to-one nature of the accessor idiom can result in many invocations when accessing an entity, which translates into a lot of network traffic even for simple edits. Every field you want to modify requires a method invocation, which in turn requires you to go out to the network. One way to reduce network traffic when editing entities is to use bulk accessors. This strategy packages access to several persistent fields into one bulk accessor. Bulk accessors provide get and set methods that work with structures or simple pass-by-value objects. The following code shows how a bulk accessor could be implemented for the Cabin bean:

```
// CabinData DataObject
public class CabinData {
    public String name;
    public int deckLevel;
    public int bedCount;
    public CabinData() {
    }
    public CabinData(String name, int deckLevel, int bedCount) {
        this.name = name;
        this.deckLevel = deckLevel;
        this.bedCount = bedCount;
    }
}

// CabinBean using bulk accessors
```



```
public class CabinBean implements javax.ejb.EntityBean {
     public int id;
     public String name;
     public int deckLevel;
     public int ship;
     public int bedCount;
     // bulk accessors
     public CabinData getData() {
         return new CabinData(name,deckLevel,bedCount);
     }
     public void setData(CabinData data) {
         name = data.name;
         deckLevel = data.deckLevel;
         bedCount = data.bedCount;
     }
     // simple accessors and entity methods
     public String getName() {
         return name;
     }
     public void setName(String str) {
         name = str;
     }
     // more methods follow
 }
```

The `getData()` and `setData()` methods allow several fields to be packaged into a simple object and passed between the client and bean in one method call. This is much more efficient than requiring three separate calls to set the name, deck level, and bed count.

Rules-of-thumb for bulk accessors

Here are some guidelines for creating bulk accessors:

Data objects are not dependent objects

Data objects and dependent objects serve clearly different purposes, but they may appear at first to be the same. Where dependent objects represent business concepts, data objects do not; they are simply an efficient way of packaging an entity's fields for access by clients. Data objects may package dependent objects along with more primitive attributes, but they are not dependent objects themselves.

Data objects are simple structures

Keep the data objects as simple as possible; ideally, they should be similar to a simple struct in C. In other words, the data object should not have any business logic at all; it should only have fields. All the business logic should remain in the entity bean, where it is centralized and easily maintained.

In order to keep the semantics of a C struct, data objects should not have accessor (get and set) methods for reading and writing their fields. The

CabinData class doesn't have accessor methods; it only has fields and a couple of constructors. The lack of accessors reinforces the idea that the data object exists only to bundle fields together, not to "behave" in a particular manner. As a design concept, we want the data object to be a simple structure devoid of behavior; it's a matter of form following function. The exception is the multi-argument constructor, which is left as a convenience for the developer.

Bulk accessors bundle related fields

The bulk accessors can pass a subset of the entity's data. Some fields may have different security or transaction needs, which require that they be accessed separately. In the CabinBean, only a subset of the fields (`name`, `deckLevel`, `bedCount`) is passed in the data object. The `id` field is not included for several reasons: it doesn't describe the business concept, it's already found in the primary key, and the client should not edit it. The `ship` field is not passed because it should only be updated by certain individuals; the identities authorized to change this field are different from the identities allowed to change the other fields. Similarly, access to the ship may fall under a different transaction isolation level than the other fields (e.g., `Serializable` versus `Read Committed`).

In addition, it's more efficient to design bulk accessors that pass logically related fields. In entity beans with many fields, it is possible to group certain fields that are normally edited together. An employee bean, for example, might have several fields that are demographic in nature (`address`, `phone`, `email`) that can be logically separated from fields that are specific to benefits (`compensation`, `401K`, `health`, `vacation`). Logically related fields can have their own bulk accessor; you might even want several bulk accessors in the same bean:

```
public interface Employee extends javax.ejb.EJBObject {

   public EmployeeBenefitsData getBenefitsData()
      throws RemoteException;

   public void setBenefitsData(EmployeeBenefitsData data)
      throws RemoteException;

   public EmployeeDemographicData getDemographicData()
      throws RemoteException;

   public void setDemographicData(EmployeeDemographicData data)
      throws RemoteException;

   // more simple accessors and other business methods follow

}
```

Retain simple accessors

Simple accessors (get and set methods for single fields) should not be abandoned when using bulk accessors. It is still important to allow editing of

single fields. It's just as wasteful to use a bulk accessor to change one field as it is to change several fields using simple accessors.

Local references in EJB 2.0 container-managed persistence are very efficient, so the performance benefits of bulk accessors are minimal. Therefore, if you're using EJB 2.0, use bulk accessors with remote interfaces whenever it makes sense according to the guidelines given here, but use them sparingly with local interfaces.

Entity Objects

The pass-by-value section earlier gave you some good ground rules for when and how to use pass-by-value in EJB. Business concepts that do not meet the dependent object criteria should be modeled as either session or entity beans. It's easy to mistakenly adopt a strategy of passing business objects that would normally qualify as entity beans (Customer, Ship, and City) by value to the clients. Overzealous use of bulk accessors that pass data objects loaded with business behavior is bad design. The belief is that passing the entity objects to the client avoids unnecessary network traffic by keeping the set and get methods local. The problem with this approach is object equivalence. Entities are supposed to represent the actual data on the database, which means that they are shared and always reflect the current state of the data. Once an object is resident on the client, it is no longer representative of the data. It is easy for a client to end up with many dirty copies of the same entity, resulting in inconsistent processing and representation of data.

While it's true that the set and get methods of entity objects can introduce a lot of network traffic, implementing pass-by-value objects instead of using entity beans is not the answer. The network problem can be avoided if you stick to the design strategy elaborated throughout this book: remote clients interact primarily with session beans, not entity beans. You can also reduce network traffic significantly by using bulk accessors, provided that these accessors only transfer structures with no business logic. Finally, try to keep the entity beans on the server encapsulated in workflow defined by session beans. This eliminates the network traffic associated with entities, while ensuring that they always represent the correct data.

Improved Performance with Session Beans

In addition to defining the interactions among entity beans and other resources (workflow), session beans have another substantial benefit: they improve performance. The performance gains from using session beans are related to the concept of *granularity*. Granularity describes the scope of a business component, or how much business territory the component covers. As you

learned previously, very fine-grained dependent business objects are usually modeled as pass-by-value objects. At a small granularity, you are dealing with entity beans like Ship or Cabin. These have a scope limited to a single concept and can only impact the data associated with that concept. Session beans represent large, coarse-grained components with a scope that covers several business concepts—all the business concepts or processes that the bean needs in order to accomplish a task. In distributed business computing, you rely on fine-grained components like entity beans to ensure simple, uniform, reusable, and safe access to data. Coarse-grained business components like session beans capture the interactions of entities or business processes that span multiple entities so that they can be reused; in doing so, they also improve performance on both the client and the server. As a rule of thumb, client applications should do most of their work with coarse-grained components like session beans, and with limited direct interaction with entity beans.

To understand how session beans improve performance, we have to address the most common problems cited with distributed component systems: network traffic, latency, and resource consumption.

Network Traffic and Latency

One of the biggest problems of distributed component systems is that they generate a lot of network traffic. This is especially true of component systems that rely solely on entity- type business components, such as EJB's `EntityBean` component. Every method call on a remote reference begins a remote method invocation loop, which sends information from the stub to the server and back to the stub. The loop requires data to be streamed to and from the client, consuming bandwidth. If we built a reservation system for Titan Cruise Lines, we would probably use several entity beans like Ship, Cabin, Cruise, and Customer. As we navigate through these fine-grained beans, requesting information, updating their states, and creating new beans, we generate network traffic. One client probably doesn't generate very much traffic, but multiply that by thousands of clients and we start to develop problems. Eventually, thousands of clients will produce so much network traffic that the system as a whole will suffer.

Another aspect of network communications is *latency*. Latency is the delay between the time we execute a command and the time it completes. With enterprise beans there is always a bit of latency due to the time it takes to communicate requests via the network. Each method invocation requires a RMI loop that takes time to travel from the client to the server and back to the client. A client that uses many beans will suffer from a time delay with each method invocation. Collectively, the latency delays can result in very slow clients that take several seconds to respond to each user action.

Accessing coarse-grained session beans from the client instead of fine-grained entity beans can substantially reduce problems with network bandwidth and

latency. In Chapter 12, we developed the `bookPassage()` method on the TravelAgent bean. The `bookPassage()` method encapsulates the interactions of entity beans that would otherwise have resided on the client. For the network cost of one method invocation on the client (`bookPassage()`), several tasks are performed on the EJB server. Using session beans to encapsulate several tasks reduces the number of remote method invocations needed to accomplish a task, which reduces the amount of network traffic and latency encountered while performing these tasks.

In EJB 2.0, a good design is to use remote component interfaces on the session bean that manages the workflow, and local compenent interfaces on the enterprise beans (both entity and session) that it manages. This ensures the best performance.

Striking a Balance

We don't want to abandon the use of entity business components, because they provide several advantages over traditional two-tier computing. They allow us to encapsulate the business logic and data of a business concept so that it can be used consistently and reused safely across applications. In short, entity business components are better for accessing business state because they simplify data access.

At the same time, we don't want to overuse entity beans on the client. Instead, we want the client to interact with coarse-grained session beans that encapsulate the interactions of small-grained entity beans. There are situations where the client application should interact with entity beans directly. If a client application needs to edit a specific entity— change the address of a customer, for example—exposing the client to the entity bean is more practical than using a session bean. If, however, a task needs to be performed that involves the interactions of more than one entity bean—transferring money from account to another, for example—then a session bean should be used.

When a client application needs to perform a very specific operation on an entity, like an update, it makes sense to make the entity available to client directly. If the client is performing a task that spans business concepts or otherwise involves more then one entity, that task should be modeled in a session bean as a workflow. A good design will emphasize the use of coarse-grained session beans as workflow and limit the number of activities that require direct client access to entity beans.

In EJB 2.0, entity beans that are accessed by both remote clients and local enterprise beans can accommodate both by implementing both remote and local component interfaces. The methods defined in remote and local component interfaces do not need to be identical; each should define methods appropriate to the clients that will use them. For example, the remote interfaces might make more use of bulk accessors then the local interface.

Listing Behavior

Make decisions about whether to access data directly or through entity beans with care. Listing behavior that is specific to a workflow can be provided by direct data access from a session bean. Methods like `listAvailableCabins()` in the TravelAgent bean use direct data access because it is less expensive than creating a find method in the Cabin bean that returns a list of Cabin beans. Every bean that the system has to deal with requires resources; by avoiding the use of components where their benefit is questionable, we can improve the performance of the whole system. A CTM is like a powerful truck, and each business component it manages is like a small weight. A truck is much better at hauling around a bunch of weights than an lightweight vehicle like a bicycle, but piling too many weights on the truck will make it just as ineffective as the bicycle. If neither vehicle can move, which one is better?

Chapter 12 spends some time discussing the TravelAgent bean's `listAvailableCabins()` method as an example of a method that returns a list of tabular data. This section provides several different strategies for implementing listing behavior in your beans.

Tabular data is data that is arranged into rows and columns. Tabular data is often used to let application users select or inspect data in the system. Enterprise JavaBeans lets you use find methods to list entity beans, but this mechanism is not a silver bullet. In many circumstances, find methods that return remote references are a heavyweight solution to a lightweight problem. For example, Table 9-1 shows the schedule for a cruise.

Table 9-2: Hypothetical Cruise Schedule (continued)

Cruise ID	Port-of-Call	Arrive	Depart
233	San Juan	June 4, 1999	June 5, 1999
233	Aruba	June 7, 1999	June 8, 1999
233	Cartagena	June 9, 1999	June 10, 1999
233	San Blas Islands	June 11, 1999	June 12, 1999

It would be possible to create a Port-Of-Call entity object that represents every destination, and then obtain a list of destinations using a find method, but this would be overkill. Recognizing that the data is not shared and only useful in this one circumstance, we would rather present the data as a simple tabular listing.

In this case, we will present the data to the bean client as an array of `String` objects, with the values separated by a character delimiter. Here is the method signature used to obtain the data:.

```
public interface Schedule implements javax.ejb.EJBObject {
    public String [] getSchedule(int ID) throws RemoteException;
}
```


And here is the structure of the `String` values returned by the `getSchedule()` method:

```
233; San Juan; June 4, 1999; June 5, 1999
233; Aruba; June 7, 1999; June 8, 1999
233; Cartegena; June 9, 1999; June 10, 1999
233; San Blas Islands; June 11, 1999; June 12, 1999
```

The data could also be returned as a multidimensional array of strings, in which each column represents one field. This would certainly make it easier to reference each data item, but would also complicate navigation.

One disadvantage to using the simple array strategy is that Java is limited to single type arrays. In other words, all the elements in the array must be of the same type. We use an array of `Strings` here because it has the most flexibility for representing other data types. We could also have used an array of `Objects` or even a `Vector`. The problem with using an `Object` array or a `Vector` is that there is no typing information at runtime or development time.

Implementing lists as arrays of structures

Instead of returning a simple array, a method that implements some sort of listing behavior can also return an array of structures. For example, to return the cruise ship schedule data illustrated in Table 9-1, you could return an array of schedule structures. The structures are simple Java objects with no behavior (i.e., no methods) that are passed in an array. The definition of the structure and the bean interface that would be used are:

```
// Definition of the bean that uses the Structure
public interface Schedule implements javax.ejb.EJBObject {
  public CruiseScheduleItem [] getSchedule(int ID) throws RemoteException;
}

// Definition of the Structure
public class CruiseScheduleItem {
    public int cruiseID;
    public String portName;
    public java.util.Date arrival;
    public java.util.Date departure;
}
```

Using structures allows the data elements to be of different types. In addition, the structures are self-describing: it is easy to determine the structure of the data in the tabular set based on its class definition.

Implementing lists as ResultSets

A more sophisticated and flexible way to implement a list is to provide a pass-by-value implementation of the `java.sql.ResultSet` interface. Although it is defined in the JDBC package (`java.sql`) the `ResultSet` interface is

semantically independent of relational databases; it can be used to represent any set of tabular data. Since the `ResultSet` interface is familiar to most enterprise Java developers, it is an excellent construct for use in listing behavior. Using the `ResultSet` strategy, the signature of the `getSchedule()` method would be:

```
public interface Schedule implements javax.ejb.EJBObject {
    public ResultSet getSchedule(int cruiseID) throws RemoteException;
}
```

In some cases, the tabular data displayed at the client may be generated using standard SQL through a JDBC driver. If the circumstances permit, you may choose to perform the query in a session bean and return the result set directly to the client through a listing method. However, there are many cases in which you don't want to return a `ResultSet` that comes directly from JDBC drivers. A `ResultSet` from a JDBC 1.x driver is normally connected directly to the database, which increases network overhead and exposes your data source to the client. In these cases, you can implement your own `ResultSet` object that uses arrays or vectors to cache the data. JDBC 2.0 provides a cached `javax.sql.RowSet` that looks like a `ResultSet`, but is passed by value and provides features like reverse scrolling. You can use the `RowSet`, but don't expose behavior that allows the result set to be updated. Data updates should only be performed by bean methods.

In some cases, the tabular data comes from several data sources or nonrelational databases. In these cases, you can query the data using the appropriate mechanisms within the listing bean, and then reformat the data into your `ResultSet` implementation. Regardless of the source of data, you still want to present it as tabular data using a custom implementation of the `ResultSet` interface.

Using a `ResultSet` has a number of advantages and disadvantages. First, the advantages:

Consistent interface for developers
: The `ResultSet` interface provides a consistent interface that developers are familiar with and that is consistent across different listing behaviors. Developers don't need to learn several different constructs for working with tabular data; they use the same `ResultSet` interface for all listing methods.

Consistent interface for automation
: The `ResultSet` interface provides a consistent interface that allows software algorithms to operate on data independent of its content. A builder can be created that constructs an HTML or GUI table based on any set of results that implements the `ResultSet`.

Metadata operations
: The `ResultSet` interface defines several metadata methods that provide developers with runtime information describing the result set they are working with.

Flexibility
: The `ResultSet` interface is independent of the data content, which allows tabular sets to change their schema independent of the interfaces. A change in schema does not require a change to the method signatures of the listing operations.

And now, the disadvantages of using a `ResultSet`:

Complexity
: The `ResultSet` interface strategy is much more complex than returning a simple array or an array of structures. It normally requires you to develop a custom implementation of the `ResultSet` interface. If properly designed, the custom implementation can be reused across all your listing methods, but it's still a significant development effort.

Hidden structure at development time
: Although the `ResultSet` can describe itself through metadata at runtime, it cannot describe itself at development time. Unlike a simple array or an array of structures, the `ResultSet` interface provides no clues at development time about the structure of the underlying data. At runtime, metadata is available, but at development time, good documentation is required to express the structure of the data explicitly.

Bean Adapters

One of the most awkward aspects of the EJB bean interface types is that, in some cases, the callback methods are never used or are not relevant to the bean at all. A simple container-managed entity bean might have empty implementations for its `ejbLoad()`, `ejbStore()`, `ejbActivate()`, `ejbPassivate()`, or even its `setEntityContext()` methods. Stateless session beans provide an even better example of unnecessary callback methods: they must implement the `ejbActivate()` and `ejbPassivate()` methods even though these methods are never invoked!

To simplify the appearance of the bean class definitions, we can introduce *adapter classes* that hide callback methods that are never used or that have minimal implementations. Here is an adapter for the entity bean that provides empty implementations of all the `EntityBean` methods:

```
public class EntityAdapter implements javax.ejb.EntityBean {
    public EntityContext ejbContext;

    public void ejbActivate(){}
    public void ejbPassivate(){}
```



```
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbRemove(){}

    public void setEntityContext(EntityContext ctx) {
        ejbContext = ctx;
    }
    public void unsetEntityContext() {
        ejbContext = null;
    }
    public EntityContext getEJBContext() {
        return ejbContext;
    }
}
```

We took care of capturing the `EntityContext` for use by the subclass. We can do this because most entity beans implement the context methods in exactly this way. We simply leverage the adapter class to manage this logic for our subclasses.

If a callback method is deemed necessary, it can simply be overridden by a method in the bean class.

A similar `Adapter` class can be created for stateless session beans:

```
public class SessionAdapter implements javax.ejb.SessionBean {
    public SessionContext ejbContext;

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}

    public void setSessionContext(SessionContext ctx) {
        ejbContext = ctx;
    }
    public SessionContext getEJBContext() {
        return ejbContext;
    }
}
```

Don’t use these adapter classes when you need to override more than one or two of their methods. If you need to implement several of the callback methods, your code will be clearer if you don’t use the adapter class. The adapter class also impacts the inheritance hierarchy of the bean class. If later you would like to implement a different superclass, one that captures business logic, the class inheritance would need to be modified.

Implementing a Common Interface

This book discourages implementing the remote interface in the bean class. This makes it a little more difficult to enforce consistency between the business methods defined in the remote interface and the corresponding methods on the bean class. There are good reasons for not implementing the remote interface in the bean class, but there is also a need for a common interface to ensure that the bean class and remote interface define the same business methods. This section describes a design alternative that allows you to use a common interface to ensure consistency between the bean class and the remote interface.

Why the Bean Class Shouldn't Implement the Remote Interface

There should be no difference, other than the missing `java.rmi.RemoteException`, between the business methods defined in the `ShipBean` and their corresponding business methods defined in the `ShipRemote` interface. EJB requires you to match the method signatures so that the remote interface can accurately represent the bean class on the client. Why not implement the remote interface `com.titan.ShipRemote` in the `ShipBean` class to ensure that these methods are matched correctly?

EJB allows a bean class to implement its remote interface, but this practice is discouraged for a couple of very good reasons. First, the remote interface is actually an extension of the `javax.ejb.EJBObject` interface, which you learned about in Chapter 5. This interface defines several methods that are implemented by the EJB container when the bean is deployed. Here is the definition of the `javax.ejb.EJBObject` interface:

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {
    public abstract EJBHome getEJBHome();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey();
    public abstract boolean isIdentical(EJBObject obj);
    public abstract void remove();
}
```

The methods defined here are implemented and supported by the EJB object for use by client software and are not implemented by the `javax.ejb.EntityBean` class. In other words, these methods are intended for the remote interface's implementation, not the bean instance's. The bean instance implements the business methods defined in the remote interface, but it does so indirectly. The EJB object receives all the method invocations made on the remote interface; those that are business methods (like the `getName` or `setCapacity` methods in `Ship`) are delegated to the bean instance. The other methods, defined by the `EJBObject`, are handled by the container and are never delegated to the bean instance.

Just for kicks, change the `ShipBean` definition so that it implements the `Ship` interface as show here:

```
public class ShipBean implements ShipRemote {
```

When you recompile the `ShipBean`, you should have five errors stating that the `ShipBean` must be declared abstract because it doesn't implement the methods from the `javax.ejb.EJBObject`. EJB allows you to implement the remote interface, but in so doing you clutter the bean class's definition with a bunch of methods that have nothing to do with its functionality. You can hide these methods in an adapter class; however, using an adapter for methods that have empty implementations is one thing, but using an adapter for methods that shouldn't be in the class at all is decidedly bad practice.

Another reason that beans should not implement the remote interface is that a client can be an application on a remote computer or it can be another bean. Beans as clients are very common. When calling a method on an object, the caller sometimes passes itself as one of the parameters.[1] In normal Java programming, an object passes a reference to itself using the `this` keyword. In EJB, however, clients, even bean clients, are only allowed to interact with the remote interfaces of beans. When one bean calls a method on another bean, it is not allowed to pass the `this` reference; it must obtain its own remote reference from its context and pass that instead. The fact that a bean class doesn't implement its remote interface prevents you from passing the `this` reference and forces you to get a reference to the interface from the context. The bean class won't compile if you attempt to use `this` as a remote reference. For example, assume that the `ShipBean` needs to call `someMethod(ShipRemote ship)`. It can't simply call `someMethod(this)` because `ShipBean` doesn't implement `ShipRemote`. If, however, the bean instance implements the remote interface, you could mistakenly pass the bean instance reference using the `this` keyword to another bean.

Beans should always interact with the remote references of other beans so that method invocations are intercepted by the EJB objects. Remember that the EJB objects apply security, transaction, concurrency, and other system-level constraints to method calls before they are delegated to the bean instance; the EJB object works with the container to manage the bean at runtime.

The proper way to obtain a bean's remote reference, within the bean class, is to use the `EJBContext`. Here is an example of how this works:

```
public class HypotheticalBean extends EntityBean {
    public EntityContext ejbContext;
    public void someMethod() throws RemoteException {
```

1 This is frequently done in loopbacks where the invokee will need information about the invoker. Loopbacks are discouraged in EJB because they require reentrant programming, which should be avoided.


```
            Hypothetical mySelf = (Hypothetical) ejbContext.getEJBObject();

            // Do something interesting with the remote reference.
        }
        // More methods follow.
}
```

EJB 2.0: Why the Bean Class Shouldn't Implement the Local Interface

In EJB 2.0, the bean class should not implement the local interface for the exact same reasons that it shouldn't implement the remote interface: You would have to support the methods of the `javax.ejb.EJBLocalObject`, which are not germane to the bean class.

EJB 1.1: The Business Interface Alternative

Although it is undesirable for the bean class to implement its remote interface, we can define an intermediate interface that is used by both the bean class and the remote interface to ensure consistent business method definitions. We will call this intermediate interface the *business interface*.

The following code contains an example of a business interface defined for the Ship bean, called `ShipBusiness`. All the business methods formerly defined in the `ShipRemote` interface are now defined in the `ShipBusiness` interface. The business interface defines all the business methods, including every exception that will be thrown from the remote interface when used at runtime:

```
package com.titan.ship;
import java.rmi.RemoteException;

public interface ShipBusiness {
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public void setCapacity(int cap) throws RemoteException;
    public int getCapacity() throws RemoteException;
    public double getTonnage() throws RemoteException;
    public void setTonnage(double tons) throws RemoteException;
}
```

Once the business interface is defined, it can be extended by the remote interface. The remote interface extends both the `ShipBusiness` and the `EJBObject` interfaces, giving it all the business methods and the `EJBObject` methods that the container will implement at deployment time:

```
package com.titan.ship;
```



```
import javax.ejb.EJBObject;

public interface ShipRemote extends ShipBusiness, javax.ejb.EJBObject {
}
```

Finally, we can implement the business interface in the bean class as we would any other interface:

```
public class ShipBean implements ShipBusiness, javax.ejb.EntityBean {
    public int id;
    public String name;
    public int capacity;
    public double tonnage;

    public String getName() {
        return name;
    }
    public void setName(String n) {
        name = n;
    }
    public void setCapacity(int cap) {
        capacity = cap;
    }
    public int getCapacity() {
        return capacity;
    }
    public double getTonnage() {
        return tonnage;
    }
    public void setTonnage(double tons) {
        tonnage = tons;
    }

    // More methods follow...
}
```

In the case of the `ShipBean` class, we choose not to throw the `RemoteException`. Classes that implement interfaces can choose not to throw exceptions defined in the interface. They cannot, however, add exceptions. This is why the business interface must declare that its methods throw the `RemoteException` and all application exceptions. The remote interface should not modify the business interface definition. The bean class can choose not to throw the `RemoteException`, but it must throw all the application-specific exceptions.

The business interface is an easily implemented design strategy that will make it easier to develop beans. This book recommends that you use the business interface strategy in your own implementations. Remember not to pass the business interface in method calls; always use the bean's remote interface in method parameters and as return types.

Entity Beans Without Create Methods

If an entity bean is never meant to be created by a client, you can simply not implement a `create()` method on the home interface. This means that the entity in question can only be obtained using the `find()` methods on the home interface. Titan might implement this strategy with their Ship beans, so that new ships must be created by directly inserting a record into the database—a privilege that might be reserved for the database administrator. They wouldn't want some crazed travel agent inserting random ships into their cruise line.

EJB 1.1: Object-to-Relational Mapping Tools

Some EJB vendors provide object-to-relational mapping tools that, using wizards, can create object representations of relational databases, generate tables from objects, or map existing objects to existing tables. These tools are outside the scope of this book because they are proprietary in nature and cannot generally be used to produce beans that can be used across EJB servers. In other words, in many cases, once you have begun to rely on a mapping tool to define a bean's persistence, you might not be able to migrate your beans to a different EJB server; the bean definition is bound to the mapping tool.

Mapping tools can make bean developers much more productive, but you should consider the implementation-specific details of your tool before using it. If you will need to migrate your application to a bigger, faster EJB server in the future, make sure that the mapping tool you use is supported in other EJB servers.

Some products that perform object-to-relational mapping use JDBC. The Object People's TOPLink and Watershed's ROF are examples of this type of product. These products provide more flexibility for mapping objects to a relational database and are not as dependent on the EJB server. However, EJB servers must support these products in order for them to be used, so again let caution guide your decisions about using these products.

Avoid Emulating Entity Beans with Session Beans

Session beans that implement the `SessionSynchronization` interface (discussed in Chapter 8) can emulate some of the functionality of bean-managed entity beans. This approach provides a couple of advantages. First, these session beans can represent entity business concepts like entity beans; second, dependency on vendor-specific object-to- relational mapping tools is avoided.

Unfortunately, session beans were never designed to represent data directly in the database, so using them as a replacement for entity beans is problematic. Entity beans fulfill this duty nicely because they are transactional objects. When the attributes of a bean are changed, the changes are reflected in the database automatically in a transactionally safe manner. This cannot be duplicated in stateful session beans because they are transactionally aware but are not transactional objects. The difference is subtle but important. Stateful session beans are not shared like entity beans. There is no concurrency control when two clients attempt to access the same bean at the same time. In the case of the stateful session beans, each client gets its own instance, so many copies of the same session bean representing the same entity data can be in use concurrently. Database isolation can prevent some problems, but the danger of obtaining and using dirty data is high.

Other problems include the fact that session beans emulating entity beans cannot have `find()` methods in their home interfaces. Entity beans support `find()` methods as a convenient way to locate data. Find methods could be placed in the session bean's remote interface, but this would be inconsistent with the EJB component model. Also, a stateful session bean must use the `SessionSynchronization` interface to be transactionally safe, which requires that it only be used in the scope of the client's transaction. This is because methods like `ejbCreate()` and `ejbRemove()` are not transactional. In addition, `ejbRemove()` has a significantly different function in session beans than in entity beans. Should `ejbRemove()` end the conversation, delete data, or both?

Weighing all the benefits against the problems and risks of data inconsistency, it is recommended that you do not use stateful session beans to emulate entity beans.

Limiting Session Beans to Workflow

Direct database access with JDBC

Perhaps the most straightforward and most portable option for using a server that only supports session beans is direct database access. We did some of this with the ProcessPayment bean and the TravelAgent bean in Chapter 12. When entity beans are not an option, we simply take this a step further. The following code is an example of the TravelAgent bean's `bookPassage()` method, coded with direct JDBC data access instead of using entity beans:

```
public Ticket bookPassage(CreditCard card, double price)
        throws RemoteException, IncompleteConversationalState {
    if (customerID == 0 || cruiseID == 0 || cabinID == 0) {
        throw new IncompleteConversationalState();
    }
```



```
    Connection con = null;
    PreparedStatement ps = null;;
    try {
        con = getConnection();

        // Insert reservation.
        ps = con.prepareStatement("insert into RESERVATION "+
            "(CUSTOMER_ID, CRUISE_ID, CABIN_ID, PRICE) values (?,?,?,?)");
        ps.setInt(1, customerID);
        ps.setInt(2, cruiseID);
        ps.setInt(3, cabinID);
        ps.setDouble(4, price);
        if (ps.executeUpdate() != 1) {
        throw new RemoteException (
            "Failed to add Reservation to database");
        }
        // Insert payment.
        ps = con.prepareStatement("insert into PAYMENT "+
            "(CUSTOMER_ID, AMOUNT, TYPE, CREDIT_NUMBER, CREDIT_EXP_DATE) "+
            "values(?,?,?,?,?)");
        ps.setInt(1, customerID);
        ps.setDouble(2, price);
        ps.setString(3, card.type);
        ps.setLong(4, card.number);
        ps.setDate(5, new java.sql.Date(card.experation.getTime()));
        if (ps.executeUpdate() != 1) {
        throw new RemoteException (
            "Failed to add Reservation to database");
        }
        Ticket ticket = new Ticket(customerID,cruiseID,cabinID,price);
        return ticket;

    } catch (SQLException se) {
      throw new RemoteException (se.getMessage());
    }
    finally {
      try {
        if (ps != null) ps.close();
        if (con!= null) con.close();
      } catch(SQLException se){
        se.printStackTrace();
      }
    }
}
```

No mystery here: we have simply redefined the TravelAgent bean so that it works directly with the data through JDBC rather than using entity beans. This method is transactional safe because an exception thrown anywhere within the method will cause all the database inserts to be rolled back. Very clean and simple.

The idea behind this strategy is to continue to model workflow or processes with session beans. The TravelAgent bean models the process of making a reservation. Its conversational state can be changed over the course of a conversation, and safe database changes can be made based on the conversational state.

EJB 1.1: Direct access with object-to-relational mapping tools

Object-to-relational mapping provides another mechanism for "direct" access to data in a stateful session bean. The advantage of object-to-relational mapping tools is that data can be encapsulated as object-like entity beans. So, for example, an object-to-relational mapping approach could end up looking very similar to our entity bean design. The problem with object-to-relational mapping is that most tools are proprietary and may not be reusable across EJB servers. In other words, the object-to-relational tool may bind you to one brand of EJB server. Object-to-relational mapping tools are, however, a much more expedient, safe, and productive mechanism to obtaining direct database access when entity beans are not available.

Avoid Chaining Stateful Session Beans

In developing session-only systems you will be tempted to use stateful session beans from inside other stateful session beans. While this appears to be a good modeling approach, it's problematic. Chaining stateful session beans can lead to problems when beans time out or throw exceptions that cause them to become invalid. Figure 9-1 shows a chain of stateful session beans, each of which maintains conversational state that other beans depend on to complete an operation encapsulated by bean A.

[FIGURE (use figure 9-1)]

Figure 9-1: Chain of stateful session beans

If any one of the beans in this chain times out, say bean B, the conversational state trailing that bean is lost. If this conversational state was built up over a long time, considerable work can be lost. The chain of stateful session beans is only as strong as its weakest link. If one bean times out or becomes invalid, the entire conversational state on which bean A depends becomes invalid. Avoid chaining stateful session beans.

Using stateless session beans from within stateful session beans is not a problem, because a stateless session bean does not maintain any conversational state. Use stateless session beans from within stateful session beans as much as you need.

Using a stateful session bean from within a stateless session bean is almost nonsensical because the benefit of the stateful session bean's conversational state cannot be leveraged beyond the scope of the stateless session bean's method.

17

Java 2, Enterprise Edition

The specification for the Java 2, Enterprise Edition (J2EE) defines a platform for developing web-enabled applications that includes Enterprise JavaBeans, Servlets, and Java Server Pages (JSP). J2EE products are application servers that provide a complete implementation of the EJB, Servlet, and JSP technologies. In addition, the J2EE outlines how these technologies work together to provide a complete solution. To understand what J2EE is, it's important that we introduce Servlets and JSP and explain the synergy between these technologies and Enterprise JavaBeans.

At risk of spoiling the story, J2EE provides two kinds of "glue" to make it easier for components to interact. We've already seen both types of glue. The JNDI Enterprise Naming Context (ENC) is used to standardize the way components look up resources that they need. We've seen the ENC in the context of enterprise beans; in this chapter, we'll look briefly at how servlets, JSPs, and even some clients can use the ENC to find resources. Second, the idea of deployment descriptors—in particular, the use of XML to define a language for deployment descriptors is also used with servlets and JSP. Java servlets and server pages can be packaged with deployment descriptors that define their relationship to their environment. Deployment descriptors are also used to define entire assemblies of many components into applications.

Servlets

The Servlet specification defines a server-side component model that can be implemented by web server vendors. Servlets provide a simple but powerful API for generating web pages dynamically. (Although servlets can be used for many

different request- response protocols, they are predominantly used to process HTTP requests for web pages.)

Servlets are developed in the same fashion as enterprise beans; they are Java classes that extend a base component class and have a deployment descriptor. Once a servlet is developed and packaged in a JAR file, it can be deployed in a web server. When a servlet is deployed, it is assigned to handle requests for a specific web page or assist other servlets in handling page requests. The following servlet, for example, might be assigned to handle any request for the *helloworld.html* page on a web server:

```
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

  protected void doGet(HttpServletRequest req,
    HttpServletResponse response)
    throws ServletException,java.io.IOException {

    try {
     ServletOutputStream writer = response.getWriter();
     writer.println("<HTML><BODY>");
     writer.println("<h1>Hello World!!</h1>");
     writer.println("</BODY></HTML>");
  } catch(Exception e) {
  // handle exception
  }
  ...
}
```

When a browser sends a request for the page to the web server, the server delegates the request to the appropriate servlet instance by invoking the servlet's `doGet()` method.[1] The servlet is provided information about the request in the `HttpServletRequest` object, and can use the `HttpServletResponse` object to reply to the request. This simple servlet sends a short HTML document including the text "Hello World" back to the browser, which displays it. Figure 17-1 illustrates how a request is sent by a browser and serviced by a servlet running in a web server.

[FIGURE (use figure 11-1)]

Figure 17-1: Servlet servicing an HTTP request

Servlets are similar to session beans because they both perform a service and can directly access backend resources like a database through JDBC, but they do not represent persistent data. Servlets do not, however, have support for transactions and are *not* composed of business methods. Servlets respond to

1 HttpServlets also have a `doPost()` method which handles requests for forms.

very specific requests, usually HTTP requests, and respond by writing to an output stream.

The Servlet specification is extensive and robust but also simple and elegant. It's a powerful server-side component model. You can learn more about servlets by reading *Java™ Servlet Programming, 2nd Edition* by Jason Hunter and William Crawford (O'Reilly).

Java Server Pages

Java Server Pages (JSP) is an extension of the servlet component model that simplifies the process of generating HTML dynamically. JSP essentially allows you to incorporate Java directly into an HTML page as a scripting language. In J2EE, the Java code in a JSP page can access the JNDI ENC, just like the code in a servlet. In fact, JSP pages (text documents) are translated and compiled into Java servlets, which are then run in a web server just like any other servlet—some servers do the compilation automatically at runtime. JSP can also be used to generate XML documents dynamically.

You can learn more about servlets by reading *Java™ Server Pages* by Hans Bergsten (O'Reilly).

Web Components and EJB

Together Servlets and JSP provide a powerful platform for generating web pages dynamically. Servlets and JSP, which are collectively called *web components*, can access resources like JDBC and enterprise beans. Because web components can access databases using JDBC, they can provide a powerful platform for e-commerce by allowing an enterprise to expose its business systems to the web through an HTML interface. HTML has several advantages over more conventional client applications, in Java or any other language. The most important advantages have to do with distribution and firewalls. Conventional clients need to be distributed and installed on client machines, which is their biggest limitation: they require additional work for deployment and maintenance. Applets, which are dynamically downloaded, can be used to eliminate the headache of installation, but applets have other limitations like sandbox restrictions and heavyweight downloads. In contrast, HTML is extremely lightweight, doesn't require prior installation, and doesn't suffer from security restrictions. In addition, HTML interfaces can be modified and enhanced at their source without having to update the clients.

Firewalls present another significant problem in e-commerce. HTTP, the protocol over which web pages are requested and delivered, can pass through most firewalls without a problem, but other protocols like IIOP or JRMP cannot. This has proven to be a significant barrier to the success of distributed object

systems that must support access from anonymous clients. This means that distributed object applications generally cannot be created for a client base that may have arbitrary firewall configurations. HTTP does not have this limitation, since practically all firewalls allow HTTP to pass unhindered.

The problems with distribution and firewalls have led the EJB industry to adopt, in large part, an architecture based on the collaborative use of web components (Servlets/JSP) and Enterprise JavaBeans. While web components provide the presentation logic for generating web pages, Enterprise JavaBeans provides a robust transactional middle tier for business logic. Web components access enterprise beans using the same API used by application clients. Each technology is doing what it does best: Servlets and JSP are excellent components for generating dynamic HTML, while Enterprise JavaBeans is an excellent platform for transactional business logic. Figure 17-2 illustrates how this architecture works.

[FIGURE (use figure 11-3)]

Figure 17-2: Using Servlets/JSP and EJB together

This web component–EJB architecture is so widely accepted that it begs the question, "Should there be a united platform?" This is the question that the J2EE specification is designed to answer. The J2EE specification defines a single application server platform that focuses on the interaction between these Servlets, JSP, and EJB. J2EE is important because it provides a specification for the interaction of web components with enterprise beans, making solutions more portable across vendors that support both component models.

J2EE Fills in the Gaps

The J2EE specification attempts to fill the gaps between the web components and Enterprise JavaBeans by defining how these technologies come together to form a complete platform.

One of the ways in which J2EE adds value is by creating a consistent programming model across web components and enterprise beans through the use of the JNDI ENC and XML deployment descriptors. A servlet in J2EE can access JDBC `DataSource` objects, environment entries, and references to enterprise beans through a JNDI ENC in exactly the same way that enterprise beans use the JNDI ENC. To support the JNDI ENC, web components have their own XML deployment descriptor that declares elements for the JNDI ENC (`<ejb-ref>`, `<resource-ref>`, `<env-entry>`) as well security roles and other elements specific to web components. In J2EE, web components (Servlets and JSP pages) along with their XML deployment descriptors, are packaged and deployed in JAR files with the extension *.war*, which stands for *w*eb *ar*chive. The use of the JNDI ENC, deployment descriptors, and JAR files in web components

makes them consistent with the EJB programming model and unifies the entire J2EE platform.

Use of the JNDI ENC makes it much simpler for web components to access Enterprise JavaBeans. The web component developer doesn't need to be concerned with the network location of enterprise beans; the server will map the `ejb-ref` elements listed in the deployment descriptor to the enterprise beans at deployment time. The JNDI ENC also supports access to a `javax.jta.UserTransaction` object, as is the case in EJB. The `UserTransaction` object allows the web component to manage transactions explicitly. The transaction context must be propagated to any enterprise beans accessed within the scope of the transaction (according to the transaction attribute of the enterprise bean method). A *.war* file can contain several servlets and JSP documents, which share an XML deployment descriptor.

J2EE also defines an *.ear* (Enterprise archive) file, which is a JAR file for packaging Enterprise JavaBean JAR files and web component JAR files (*war* files) together into one complete deployment called a J2EE Application. A J2EE Application has its own XML deployment descriptor that points to the EJB and web component JAR files (called modules) as well as other elements like icons, descriptions, and the like. When a J2EE Application is created, interdependencies like `ejb-ref` elements can be resolved and security roles can be edited to provide a unified view of the entire web application.

The J2EE Enterprise Archive (*.ear*) file would contain the EJB JAR files and the web component *.war* files. Figure 17-3 illustrates the file structure inside a J2EE archive file.

[FIGURE (figure 11-3)]

Figure 17-3: Contents of a J2EE EAR file

J2EE Application Client Components

In addition to integrating web and enterprise bean components, J2EE introduces a completely new component model: the application client component. An application client component is a Java application that resides on a client machine and accesses enterprise bean components on the J2EE server. Client components also have access to a JNDI ENC that operates the same way as the JNDI ENC for web and enterprise bean components. The client component also includes an XML deployment descriptor that declares the `env-entry`, `ejb-ref`, and `resource-ref` elements of the JNDI ENC in addition to a `description`, `display-name`, and `icon` that can be used to represent the client component in a deployment tool.

A client component is simply a Java program that uses the JNDI ENC to access environment properties, enterprise beans, and resources (JDBC, JavaMail, etc.)

made available by the J2EE server. Client components reside on the client machine, not the J2EE server. Here is an extremely simple component:

```
public class MyJ2eeClient {

    public static void main(String [] args) {

        InitialContext jndiCntx = new InitialContext();

        Object ref = jndiCntx.lookup("java:comp/env/ejb/ShipBean");
        ShipHome home = (ShipHome)
            PortableRemoteObject.narrow(ref,ShipHome.class);

        Ship ship = home.findByPrimaryKey(new ShipPK(1));
        String name = ship.getName();
        System.out.println(name);
    }
}
```

`MyJ2eeClient` illustrates how a client component is written. Notice that the client component did not need to use a network-specific JNDI `InitialContext`. In other words, we did not have to specify the service provider in order to connect to the J2EE server. This is the real power of the J2EE Application client component: location transparency. The client component does not need to know the exact location of the Ship EJB or choose a specific JNDI service provider; the JNDI ENC takes this care of locating the enterprise bean.

When application components are developed, an XML deployment descriptor is created that specifies the JNDI ENC entries. At deployment time, a vendor-specific J2EE tool generates the class files needed to deploy the component on client machines.

A client component is packaged into a JAR file with its XML deployment descriptor and can be included in a J2EE Application. Once a client component is included in the J2EE Application deployment descriptor, it can be packaged in the EAR file with the other components, as Figure 17-4 illustrates.

[FIGURE (use figure 11-4)]

Figure 17-4: Contents of a J2EE EAR file with Application component

Guaranteed Services

The J2EE 1.3 specification requires application servers to support a specific set of protocols and Java enterprise extensions. This ensures a consistent platform for deploying J2EE applications. J2EE application servers must provide the following “standard” services:

Enterprise JavaBeans 2.0
: J2EE products must support the complete specification.

Servlets 2.3
: J2EE products must support the complete specification.

Java Sever Pages 1.2
: J2EE products must support the complete specification.

HTTP and HTTPS
: Web components in a J2EE server service both HTTP and HTTPS requests. The J2EE product must be capable of advertising HTTP 1.0 and HTTPS (HTTP 1.0 over SSL 3.0) on ports 80 and 443 respectively.

Java RMI-IIOP
: As was the case with EJB 2.0, only the semantics of Java RMI-IIOP are required; the underlying protocol need not be IIOP. Therefore, components must use return and parameter types that are compatible with IIOP, and must use the `PortableRemoteObject.narrow()` method.

Java RMI-JRMP
: J2EE components can be Java RMI-JRMP clients.

JavaIDL
: Web components and enterprise beans must be able to access CORBA services hosted outside the J2EE environment using JavaIDL, a standard part of the Java 2 platform.

JDBC 2.0
: J2EE requires support for the JDBC Core (JDK 1.3) and some parts of the JDBC 2.0 Extension including connection naming and pooling, and distributed transaction support.

Java Naming and Directory Interface (JNDI) 1.2
: Web and enterprise bean components must have access to the JNDI ENC, which make available EJBHome objects, JTA UserTransaction objects, JDBC DataSource objects, and optionally Java Messaging Service connection factory objects.

JavaMail 1.2 and JAF 1.0
: J2EE products must support sending basic Internet mail messages (the protocol is not specified) using the JavaMail API from web and enterprise bean components. The JavaMail implementation must support MIME message types. JAF is the Java Activation Framework, which is needed to support different MIME types and is required for support of JavaMail functionality.

Java Message Service (JMS) 1.0.2
: J2EE products must provide support for both point-to-point (p2p) and publish-and-subscribe (pub/sub) messaging models. Support for the optional application integration interfaces is not required.

Java API for XML Parsing (JAXP) 1.1
: J2EE products must support JAXP and provide must at least one SAX 2 parser, at least one DOM 2 parser, and at least one XSLT transform engine.

J2EE™ Connector Architecture (JCA) 1.0

J2EE must support the JCA API from all components and provide full support for resource adapters and transaction capabilities as defined by the JCA.

Java™ Authentication and Authorization Service (JAAS) 1.0

J2EE products must support the use of JAAS as described in the JCA specification. In addition, application client containers must support the authentication facilities defined in the JAAS specification.

Java Transaction API 1.0.1

Web and enterprise bean components must have access to JTA UserTransaction objects via the JNDI ENC under the `"java:comp/UserTransaction"` context. The `UserTransaction` interface is used for explicit transaction control.

Fitting the Pieces Together

To illustrate how a J2EE platform would be used, imagine using a J2EE server in Titan's reservation system. To build this system, we would use the TravelAgent, Cabin, ProcessPayment, Customer, and other enterprise beans we defined in this book, along with web components that would provide a HTML interface.

The web components would access the enterprise beans in the same way that any Java client would, by using the enterprise beans' remote and home interfaces. The web components would generate HTML to represent the reservation system.

Figure 17-5 shows a web page generated by a servlet or JSP page for the Titan reservation system. This web page was generated by web components on the J2EE server. The person using the reservation system would have been guided through a login page, a customer selection page, and cruise selection page, and would be about to choose an available cabin for the customer.

[FIGURE (use figure 11-5)]

Figure 17-5: HTML interface to the Titan reservation system

The list of available cabins was obtained from the TravelAgent EJB, whose `listAvailableCabins()` method was invoked by the servlet that generated the web page. The list of cabins was used to create a HTML list box in a web page that was loaded into the user's browser. When the user chooses a cabin and submits the selection, an HTTP request is sent to the J2EE server. The J2EE server receives the request and delegates it to the `ReservationServlet`, which invokes the `TravelAgent.bookPassage()` method to do the actual reservation. The Ticket information returned by the `bookPassage()` method is then used to create another web page that is sent back to the user's browser. Figure 17-6 shows how the different components work together to process this request.

[FIGURE (use 11-6)]

Figure 17-6: J2EE Titan Reservation System

Future Enhancements

There are several areas that are targeted for improvement in the next major release of the J2EE specification. Support for “web services” is expected to be a larger part of a future J2EE specification, including support for Java API for XML messaging (JAXM), Java API for XML registries (JAXR), and Java API for XML RPC (JAX-RPC). Support for the XML Data Binding API may be required in a future version of the specification, which considered easier to use than JAXP.

In addition, J2EE may be expanded to require support for JDBC rowsets, SQLJ, management and deployment APIs, and possibly a J2EE SPI that would build on the advancements made with the JCA specification.

