EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco

ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

☐ Back Cover

Table of Contents

book24×7.com
Research. Anytime, Anyplace.

EJB™ & JSP™
Java™ On The Edge

Lou Marco

Visit us at mandtbooks.com

PROFESSIONAL MINDWARE™

M&T BOOKS

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Enterprise Java Beans and JavaServer Pages deliver the tools you need to develop state-of-the-art multi-tier applications for the Internet or an intranet. But how do you create robust components for these two APIs--and get them to work together with each other and the rest of the containers in Java 2 Enterprise Edition? This unique guide delivers the answers. With lucid explanations and lots of sample code illustrating the development of a hotel reservation system, Lou Marco shows you step by step how to harness the power of JSP and EJB-- and create cutting-edge J2EE applications.

Make JSP, EJB, and J2EE Work Together

- Get the lowdown on J2EE N-tier application development
- Work with JSP objects, standard actions, and Web sessions
- Use JavaBeans or JSP tags to access a database with JDBC
- Understand how JSP works with Java servlets
- Take control of JSP errors, exceptions, and debugging
- Master EJB basics, from clesses to session and entity Beans
- Harness EJB tools to secure your application
- Manage transactions using EJB with JDBC, JTS, and JTA
- Build Bean- or container-managed persistence in EJB components
- Learn the ins and outs of JSP and EJB as you create a fully functional hotel reservation system

About the Authors

Lou Marco is a consultant, writer, and the owner of Lou Marco and Associates, a firm that designs Web sites and writes custom software. An IT professional with more than two decades of experience, he contributes frequently to *Enterprise Systems Journal* and is the author of *ISPF/REXX Development for*

Experienced Programmers.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Part I: EJB & JSP—Java On the Edge

Chapter List

Top

Prev   Next

Chapter 1: Enterprise Computing Concepts

JavaServer Pages (JSPs) and Enterprise JavaBeans (EJBs) are part of a server-side application development specification called the Java 2 Platform, Enterprise Edition (J2EE). Before you jump into the specifics of JSPs or EJBs, some background on enterprise application development, J2EE, and how JSP and EJB fit into J2EE is in order.

This chapter sets the stage with information on the characteristics of a typical computing environment found in a modern corporation. Next, you read about two significant advancements in computer science that provide application developers with the means to satisfy their customers' demands for computing services. You get a high-level look at J2EE and see how J2EE addresses the needs of application developers. You read about the components of J2EE, which include JSP and EJB. The chapter closes with a short discussion on the roles that JSPs and EJBs play in developing enterprise applications with the J2EE specification.

The Enterprise Computing Environment

Today's corporate computing environment is a different animal from its ancestors. Typically, enterprise computing environments are:

- **Data-Obsessed:** These days, the modern company is addicted to its data. With storage costs low, companies are less likely to purge data stores today than in years past. Some industries, such as brokerage and insurance, keep decades' worth of data and subject their data to intense analysis. The astute corporate mavens realize that corporate data is an asset worth exploiting. Those in charge look to their computing professionals to provide tools that exploit this valuable asset.

- **Distributed:** Today's enterprise computing environment has grown beyond the scenario of a single machine in an air-conditioned room, with rows and rows of storage devices, serving hundreds or thousands of dumb green screens. The more likely scenario for today's environment is one of networked servers in diverse geographical locations that serve data to hundreds or thousands of comparatively smart GUI clients.

- **Secure:** A good deal of corporate data must be kept from the prying eyes of the pesky, prying employee itching to know who got the biggest raise in the department, the dementedly disgruntled employee looking to vend proprietary information, and the capriciously curious employee trying to learn about various systems and applications.

- **Scalable:** The environment that serves the needs of one hundred may be inadequate to serve the needs of one thousand. As the number of users increases, resources, such as bandwidth or database connections, have a bad habit of thinning out to unacceptable levels or simply running out.

- **Fault tolerant:** With the computing environment distributed among many parts, the possibility of any single part failing increases with the number of parts. The company cannot afford to have its systems crash and burn every time a server winks out or a data store goes offline.

- **Heterogeneous:** The days of a company using products from a single vendor are gone. More likely, a company uses a mix of hardware and software from several competing vendors. Today, everything from the physical disk

packs to the video card on the desktop may be purchased from different vendors.

The modern computing environment clearly shares the characteristics of today's diverse corporation doing business in today's diverse world.

The challenges facing systems professionals tasked with developing enterprise applications are legion. How have today's systems folk risen to the challenge? Two powerful technologies developed over the past few decades have proven instrumental in developing applications that allow the modern corporation to conduct its business. These technologies are client-server architectures and object technologies.

Client-server architectures describe how to partition the major functions of an application in layers. Object technologies deal with constructing software systems as groups of communicating objects; each object has a set of well-defined behaviors (called *methods*) and comes with its own data (called *properties*).

Developing Applications in Layers

In the days of bell-bottoms and disco music, companies used networks primarily to connect mainframes using dedicated hardware and proprietary software and protocols. In the 1980s, companies started to use UNIX servers and the TCP/IP protocol, which quickly became an industry standard. In response to servers' not adequately scaling to meet the needs of ever-increasing numbers of users, those in charge of the shop began to shift processing power from centralized servers to the network. The era of client-server computing had begun.

Developing client-server applications is different from developing applications that run on green-screen, glass house systems. The distributing of processing power between client and server demands that client-server software be developed to reflect this division.

One strategy devised to develop client-server applications is to write the software in layers. A *layer* is a logical level that deals with related application tasks. The basic idea is to develop the software to implement the layer's functions independently of features in other layers.

By partitioning software into layers, application developers could concentrate on the features and problems peculiar to a particular layer. Division of application features among layers meant division of development responsibility. In addition, the marketplace started to offer tools to support this software development strategy.

The layers commonly used to abstract a software system these days are a *presentation* layer, an *application logic* layer, and a *data* layer. Each layer is responsible for functions not found in the other layers:

- The presentation layer is responsible for user interface tasks. These tasks include accepting user input, performing various edit checks on input, and displaying relevant application output.

- The application logic layer is responsible for tasks that execute the algorithms that solve business problems. These tasks include performing calculations, handling security, and accessing data. The application logic layer contains most of the code for the application.

- The data layer is responsible for tasks that maintain permanent data stores in the form of one or more databases. These tasks include data locking, data integrity support, and transaction support.

Code that implements tasks within a layer communicates with code in adjacent layers only. For example, code within the presentation layer communicates with code within the application logic layer but does not communicate with code within the data layer.

To implement a layered application, you need an architecture that describes the physical boundaries between the above layers. The components that reside within the physical boundaries of the layers are called *tiers.* A summary of two common client-server architectures, called *two-tier* and *three-tier* (or *n-tier*) architectures, follows.

Note   The term *architecture* as used throughout this chapter refers to a partitioning strategy and a coordination strategy. The partitioning strategy leads to dividing the entire system in discrete, non-overlapping parts or components. The coordination strategy leads to explicitly defined interfaces between those parts.

Two-Tier Architectures in Brief

Some two-tier architectures combine most of the application logic layer tasks with the presentation layer, while others combine most of the application logic layer with the data layer.

A two-tier architecture could have one tier consisting of client PCs containing application logic code and database access routines and the other tier consisting of one or more databases. This arrangement is often referred to as a *fat client.*

Another way to implement the two-tier architecture is placing the application logic layer with the data layer to form a tier and having the presentation layer in the other tier. Here, the database would rely on stored procedures and triggers to implement most of the application logic. This arrangement is often referred to as a *thin client.*

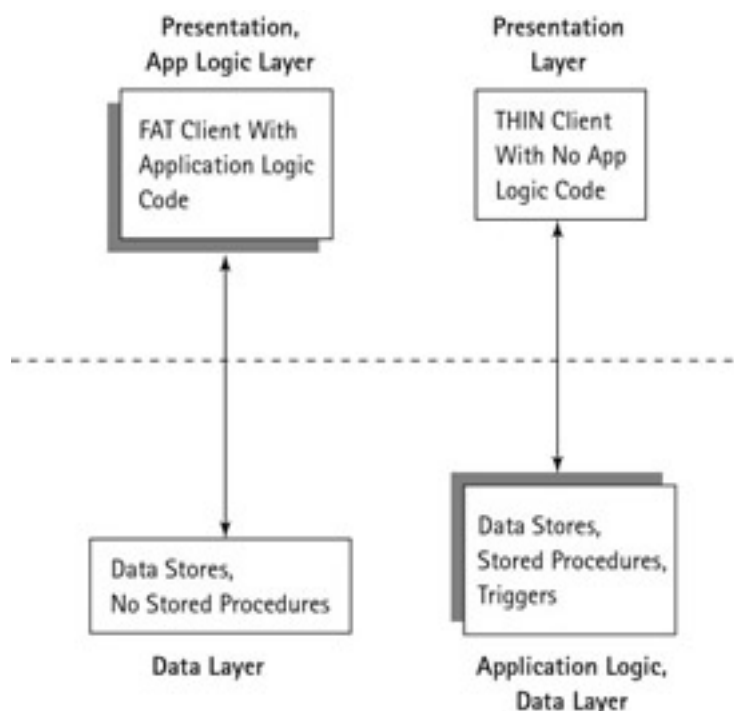Figure 1-1 shows the differences between fat and thin client arrangements.

Figure 1-1: Fat and thin clients compared and contrasted

As you can see in Figure 1-1, the fat client architecture containing application logic code is a combination of the functionality of the presentation and application logic layers. The thin client architecture has the application logic code buried within the DBMS in the form of stored procedures (code stored within the database that performs some application-specific task) and triggers (a feature of a DBMS that executes stored procedures based on one or more conditions). Most two-tier architectures fall somewhere in between these extremes. The dashed line represents the tier boundary.

Conventional wisdom these days is that two-tier architectures can satisfactorily handle a hundred or so users. For larger numbers of users, performance may start to degrade because of the client's need to maintain a connection to the server. These constant connections drain network bandwidth and use scarce database connections. This problem is more severe in the fat client than in the thin client scenario. For the fat client implementation, every request for data requires reaching across the network, dipping into the database, and returning data to the client. For the thin client implementation, one request for data can trigger a DBMS stored procedure that executes on the server. This stored procedure could return the same data that a fat client would need multiple requests for. Although using stored procedures helps alleviate the bandwidth problem, the thin client still requires the database connection.

More bad news for adopters of a two-tier architecture follows. In the fat client scenario, any change to the application logic (and you know that there will be changes) involves compiling and installing the changed code on all the clients — an expensive proposition. In the thin client scenario, the enterprise usually relies on vendor-specific databases and the vendor's implementation of triggers and stored procedures. Typically, proprietary implementations of DBMS features are not portable to different platforms and usually will not work with different vendor products.

Every strain of technology solves some old problems while introducing new ones. Two-tier architectures are certainly no exception; although applications developed with a two-tier architecture achieve some benefits by isolating tasks into separate tiers, the disadvantages of the architecture remain. A sensible question is: Are there ways of exploiting the advantages of these architectures while taking the sting out of their problems?

N-tier Architectures in Brief

Perhaps I can shed some light on a possible answer to the $64,000 question posed in the previous section by posing another question: What are the root causes of the deficiencies of the two-tier architectures? One cause is the architecture's failure to give the application logic layer its own tier. By trying to divvy up the functionality of the application logic layer, the resulting architecture ties applications to high-maintenance clients, proprietary and nonportable databases, and clogged networks. Why not give the application logic layer its very own tier?

You don't have to be a rocket scientist to guess what the architecture is called when the presentation, application logic, and data layers have their own tier. The "n" in n-tier means that a particular layer (the application logic layer, really) may have more than one physical tier. Whether you're talking about three-tier (a specific case of the more general n-tier) or n-tier, the basic concepts are the same — to encapsulate the application logic from the presentation and data layers.

What does this buy you? With the computations, business logic code, and other application logic layer tasks isolated in one or more separate tiers, these tasks do not reside in the client, nor do they reside in the database. Put another way, n-tier architectures typically deploy thin clients and DBMSs devoid of application code.

There are several paths to the road of three-tier architecture implementation. A popular implementation places the application logic layer on one or more *application servers.* These servers provide many essential services to a three-tier application, such as transaction management, resource pooling, and security.

Rather than allow a fat client or stored procedure–laden database to handle transactions (when to commit one or more transactions or when to rollback, for example), a three-tier architecture implementation delegates this vitally important function to the application server. Because business logic dictates what constitutes a transaction, support services dealing with transaction management belong on the application server because the business logic is implemented there.

As previously mentioned, a shortcoming of two-tier architectures is the consumption of resources, such as database connections, even when such resources are not needed. A characteristic of two-tier architectures is that each client needs a connection to the databases. Three- or n-tier architecture implementations allow a client to request data from one or more databases by communicating with code in the application logic layer tier. This code can dynamically connect to a database to fetch and return the requested data to the client. Also, this code can queue the data request until a database connection becomes available, and then fetch and return the requested data to the client.

Application servers — both hardware and software — are more secure than desktop client PCs. The hardware that houses the application server usually resides in a physically protected space. Rarely would you worry about stumbling over a power cord for the hardware that houses an application server! On the software side, most server software is built with security in mind unlike client desktop operating systems, such as Windows or Mac OS.

Do three-tier architectures solve the problems of two-tier architectures cited above? For the most part, they do. The problems caused by fat clients simply do not apply to n-tier architectures. Thin clients are relatively inexpensive to install and maintain. Application changes will not have much of an impact on thin clients; the application servers take the brunt of the changes.

Pulling application logic out of the DBMS by not using stored procedures places less reliance on proprietary stored procedure implementations. Three-tier implementations have a wider choice of DBMS products for use in the data

layer than two-tier, thin-client implementations.

In general, the isolation of functions in discrete layers, implemented in discrete tiers, means that each tier can be tweaked by using best-of-breed products without much impact on the remaining tiers.

As previously mentioned, any technology worth its salt solves old problems while introducing new ones. Some problems caused by implementing applications that follow the n-tier architecture are described below.

N-tier architectures are flexible. One result of this flexibility is that the three- or n-tier implementer has to cope with more hardware and software components than its two-tier counterpart. The addition of the application server opens up new system configuration possibilities. While selecting best-of-breed products to implement the system's layers is a good thing, the problems with having a multiple vendor environment, replete with finger pointing, persist. As you might imagine, maintenance costs for a large n-tier system are high.

Imagine a large n-tier application, such as a banking/ATM system, with thousands of clients dispersed all over the world securely reading and writing terabytes of data to multiple data stores. The activity between tiers necessary to get the job done must be staggering! The overhead produced by transmitting and receiving all this data across networks that connect hardware and software components that implement the multiple tiers can slow down things, to be sure.

The problems I've mentioned can be solved for the most part by spending more money for additional hardware — not exactly the favorite solution!

We've talked about the benefits of developing software in layers, or tiers. As we'll see here and throughout subsequent chapters, J2EE provides an architecture for constructing n-tier applications. Before we move on to discuss J2EE particulars, we need to take a look at another essential technology instrumental to J2EE application development that has proved its worth in theory and practice: object technology.
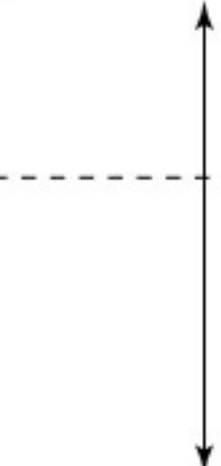
Presentation, App Logic Layer

FAT Client With Application Logic Code

Presentation Layer

THIN Client With No App Logic Code

Data Stores, No Stored Procedures

Data Layer

Data Stores, Stored Procedures, Triggers

Application Logic, Data Layer

Chapter 2: J2EE Component APIs

This chapter provides an overview of the J2EE component APIs. As mentioned in Chapter 1, J2EE is a collection of approximately 12 application programming interfaces (APIs) for developing enterprise applications. These APIs define a complete set of services that software engineers use to develop software components. J2EE simplifies the work of an application development team by providing a rich set of services that manage many application details without programming.

J2EE API Classifications

The J2EE APIs provide numerous services to n-tier application developers. We may group the J2EE APIs into three classifications corresponding to the category of service, or function, the APIs provide to the application development team. The classifications are as follows:

- **Application components:** These include *applets,* which are Java programs that execute in the client browser; *servlets,* which execute on the server; and *JSP pages,* which provide dynamic content to Web pages. J2EE also enables clients to run applications that can access data (by using a database API) without going to a Web server.

- **Resource managers:** These enable customer components to connect to an external component. These external components can be another piece of J2EE, such as JavaMail (for mail messaging) or an IBM mainframe transaction processor (such as IMS or CICS).

- **Database access:** J2EE database access relies on the Java Database Connectivity API or JDBC, which enables a customer container to issue industry-standard SQL. Relational database access in Java also relies heavily on Java Transaction Services, or JTS, and the Java Transaction API.

The J2EE APIs work in concert to provide the services mentioned in the aforementioned classifications. For example, a developer would use an application component API, such as JSP, to create a Web interface for an application that accesses data from a relational database using JDBC. In the following section, we'll take a look at J2EE APIs that fall within the preceding classifications.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

J2EE APIs

Sun Microsystems provides a list of technologies that developers use in creating J2EE applications. Most of these
technologies have an associated API. A few, notably XML, are used in several J2EE APIs. Here is a list of the J2EE
APIs with a brief description:

- **JavaServer Pages (JSP):** Enables developers to dynamically generate Web pages with HTML, XML, and Java
  code. JSP pages execute on the Web server.

- **Java Servlets:** Enables developers to dynamically create Web content as well as provide additional functionality to
  a Web server. Java servlets execute on the Web server.

- **Enterprise JavaBeans (EJB):** Defines an architecture that enables developers to create reusable, server-side
  components called enterprise beans.

- **Java Messaging Services (JMS):** A set of APIs that invoke asynchronous messaging services such as broadcast
  and point-to-point (client-to-client) messages.

- **Java Transaction API (JTA):** Provides developers with a mechanism for handling the commit and the rollback of
  transactions as well as ensuring the ACID (Atomicity, Consistency, Isolation, and Durability) properties of a
  transaction.

- **Java Transaction Services (JTS):** Provides developers with a means of communicating with transaction monitors
  and other transaction-oriented resources.

- **JavaMail:** Enables a J2EE application to send and receive e-mail.

- **Java Naming and Directory Interface (JNDI):** Provides an interface for accessing name and directory services,
  such as LDAP directory services and Domain Name Service (DNS).

- **Java Database Connectivity (JDBC):** Provides the J2EE application with a standard interface to databases
  (usually relational databases).

- **Remote Method Invocation (RMI/IIOP):** Enables a Java application to invoke methods on different Java Virtual
  Machines.

- **Interface Definition Language (IDL):** Enables J2EE-based applications to use CORBA objects.

In the following sections of this chapter, we explore the APIs in the preceding list in greater detail.

CORBA at a Glance

CORBA, the Common Object Request Broker Architecture, defines a standard for creating distributed object request
systems. The CORBA standard is the result of the collaboration of well over a hundred companies. The end result is a
standard that is language, platform, and vendor neutral.

CORBA enables the enterprise to use existing software by providing features that developers can use to wrap existing software as CORBA objects. With CORBA, applications written in several languages can happily coexist and communicate with each other.

A great deal of Enterprise JavaBeans was derived from CORBA. Indeed, a cursory look at EJB could lead one to think that EJB is a slimmed-down, Java-centric version of CORBA. EJB and CORBA can be used together, specifically when an enterprise bean needs access to code written in another language, or code written in another language needs access to an enterprise bean.

Because CORBA is the brainchild of numerous companies, no single company controls CORBA. A committee (the Object Management Group, or OMG) must agree upon changes made to the CORBA specification, which has both positive and negative consequences. On the plus side, you are fairly assured that you are not tied to a single vendor, product, or architecture when using a CORBA implementation. On the minus side, you may have to wait years for the OMG to make decisions on CORBA-related issues.

The OMG Interface Definition Language (IDL) defines the interface to objects in the CORBA universe. Although IDL is a language, you, the application programmer, do not necessarily execute IDL code. Rather, you write IDL code and use a code generator to transform IDL into a specific programming language. Java programmers use an IDL-to-Java translator to generate a representation of their IDL as Java. If you're curious, you can take a look at how IDL translates to Java by examining `ftp://www.omg.org/pub/docs/format.98-02-29.pdf`.

JavaServer Pages

You've already read some of the skinny on JavaServer Pages (JSP). Some call JSP the front door to enterprise applications, and with good reason. JSPs enable the enterprise application developer to separate presentation code from business logic code on the server, thereby providing the application with a robust presentation layer.

Java Servlets

As with JSP, servlets enable developers to dynamically create Web content as well as provide additional functionality to a Web server.

If a JSP gets translated into a servlet, why are JSPs important in the J2EE arena? JSP pages are easier to code and maintain than servlets because servlets require the Java programmer to explicitly write out HTML statements to a response object, whereas the Web page developer using JSP merely codes HTML.

cross-reference   Please refer to Chapter 3, "A First Look at JavaServer Pages" and Chapter 8, "JSP Pages and Servlets Revisited," for more detailed discussions of servlets and their relationship to JSP pages.

For example, assuming `you` is the current Web page viewer below, the following code is a JSP that generates an HTML page that displays the string `Yes, it's` concatenated with the current user.

Listing 2-1: Your first JSP page

```
<html>
<body>
<%@ page language="java" %>
<p> Yes, it's,
<% String you = (String) session.getAttribute('you');
out.println(you); %>
</p>
</body>
```

```
</html>
```

The code in Listing 2-1 is the functional equivalent to the servlet code shown in Listing 2-2.

Note   Recall that JSP pages get translated into servlets. However, the servlet code shown in Listing 2-2 is not the result of translating the JSP in Listing 2-1 into a servlet. The JSP translator generates a servlet that performs the same function as the servlet shown in Listing 2-2 but with different Java code .

Listing 2-2: A servlet functionally equivalent to the JSP page in listing 2-1

```
import java.io.*;

import javax.servlet.*;
public class HeyItsYou extends HttpServlet {
 public void doGet(HttpServletRequest req,
  HttpServletResponse res) throws ServletException, IOException {
  res.setContentType("text/html");
      HttpSession session = req.getSession( false ) ;
  PrintWriter out = res.getWriter();
  out.println("<html>");
  out.println("<body>");
  out.println ("<p> Hey, it'sey, it's,");
  out.print("String you = ");
  out.println((String) session.getAttribute('you'));
  out.println(user);
  out.println("</p>");
  out.println("</body>");
  out.println("</html>");
 }
}
```

The JSP page is smaller than the servlet, and most users agree that the JSP is easier to understand and maintain. Many others also agree that writing out HTML (or XML, of course) by way of `out.println()` statements is a major drag because a large page can have hundreds of `out.println()` statements.

Hence, the bottom line is that, while JSPs and servlets often accomplish the same task, you'll still need servlets from time to time to do what JSPs cannot.

Enterprise JavaBeans

Enterprise JavaBeans (EJB) define an architecture that enables developers to create reusable, server-side components called enterprise beans. Enterprise beans typically reside on the application server or may have their own dedicated server. Of course, you can read much more about EJB in the following chapters.

Please note that enterprise beans are not JavaBeans! One difference is that calling a JavaBean (from a servlet or JSP page) involves *intra-process* communication, whereas calling an EJB (from a servlet or JSP page) involves *inter-process* communication. You can read about other differences in the following chapters.

Java Messaging Services

Java Messaging Services (JMS) is an API that invokes asynchronous messaging services such as broadcast and point-to-point (client-to-client) messages.

JMS is an API for using networked messaging services. A messaging system accepts messages from "producer" clients and delivers them to "consumer" clients. Data sent in a message is often intended as a sort of event notification (for example, an e-mail-handling process may need to be notified when a request is queued). Another common use for messaging (thus, JMS) is for interfacing with remote legacy applications. It can be complex and sometimes risky to use Remote Procedure Call (RPC) or a Java variant such as Remote Method Invocation (RMI) to directly invoke remote applications while a messaging solution can provide an easier and more reliable interconnection. In short, why write remote procedure calls when you have access to an API specifically designed for sending messages across a network from one object to another?

JMS calls frequently rely on the Java Naming and Directory Interface (JNDI) to locate message recipients. JNDI is discussed later in this chapter.

Java Transaction API

Java Transaction API (JTA) provides developers with a mechanism for handling the commit and the rollback of transactions as well as ensuring the ACID (Atomicity, Consistency, Isolation, and Durability) properties of a transaction.

JTA is used for managing distributed transactions (e.g., updates to multiple databases that must be handled in a single transaction). JTA is a low-level API and associated coding is complex and error-prone — not in the spirit of J2EE!

Fortunately, EJB containers or application servers generally provide support for distributed transactions using JTA. For this reason, the EJB developer is able to gain the benefit of distributed transactions, leaving the complex implementation details to the provider of the EJB container. Now, that's more in the J2EE spirit!

Java Transaction Services

The Java Transaction Service (JTS) provides developers with a means of communicating with transaction monitors and other transaction-oriented resources. Of course, JTS provides high-level support for JTA as well as other transaction services.

The Java Transaction Service plays the role of an intermediary for all the constituent components of the EJB architecture. In JTS terminology, the director is called the *transaction manager.* The participants in the transaction that implement transaction-protected resources such as relational databases are called *resource managers.* When an application begins a transaction, it creates a transaction object that represents the transaction. You would use JNDI (Java Naming and Directory Interface) to access this transaction object. The application invokes the resource managers to perform the work of the transaction. As the transaction progresses, the transaction manager keeps track of each of the resource managers enlisted in the transaction. Often, JTS assists in managing the activities involved in a two-phase commit.

JavaMail

The JavaMail API offers a standard Java extension API to talk to all your favorite standard Internet mail protocols. The API provides a platform-independent and protocol-independent framework to build Java technology–based mail and messaging applications. Put differently, JavaMail represents a standardized, extensible platform for communicating, presenting, and manipulating all current and future Multipurpose Internet Mail Extension (MIME) types. The JavaMail API is implemented as a Java platform standard extension.

Say goodbye to writing your own classes for talking to mail protocols! Say goodbye to learning yet another unique third-party or in-house class library for dealing with e-mail or newsgroups! JavaMail was designed to communicate with popular protocols and MIME types.

Java Naming and Directory Interface

Java Naming and Directory Interface (JNDI) provides an interface for accessing name and directory services, such as LDAP directory services and Domain Name Service (DNS). JNDI enables Java programs to use name servers and directory servers to look up objects or data by name. This important feature enables a client object to locate a remote server object or data.

JNDI is a generic API that can work with any name or directory server. As such, JNDI was not designed to replace existing technology, but instead it provides a common interface to existing naming services. For example, JNDI provides methods to bind a name to an object, enabling that object to be located, regardless of its location on the network.

Server providers have been implemented for many common protocols (e.g., NIS, LDAP, and NDS) and for CORBA object registries. Of particular interest to users of J2EE, JNDI is used to locate Enterprise JavaBean (EJB) components on the network.

Again, the thrust of J2EE technology is to provide enterprise application developers with much-needed services in the distributed realm. It's hard to think of a more valuable service than a naming service. JNDI provides the Java application developer with this much-needed service.

Java Database Connectivity

Java Database Connectivity (JDBC) provides the J2EE application a standard interface to databases (usually relational databases). In principle, JDBC serves the same purpose as Open Database Connectivity (ODBC). JDBC provides a database-independent protocol for accessing relational databases from Java. JDBC supports Data Manipulation Language (DML) statements such as `insert`, `update`, `delete`, and `select`. It also includes Data Definition Language (DDL) statements such as `Create Table`, `Alter Table`, and so on.

Database vendors usually provide a JDBC *driver* that enables a Java program to access the vendor's RDBMS product. As of this writing, Sun has *154* JDBC drivers listed in its driver database

Note   See <u>http://industry.java.sun.com/products/jdbc/drivers</u> for a listing of available drivers for use with JDBC.

JDBC was included in core Java starting with version 1.1. With JDBC, the SQL is always dynamically generated at runtime and sent to the database. Many have griped about the inefficiencies of applying dynamically created SQL against databases. In response, another standard for Java database access has emerged and is called SQLJ. SQLJ enables static SQL to be used and it requires less cumbersome syntax than JDBC. One SQLJ advantage over JDBC is better code quality because SQL is checked at compile-time. Also, SQLJ usually shows better performance than JDBC because access paths to the database are generated once and reused for subsequent executions of the static SQL.

We speak of *levels* for JDBC drivers; the slowest are level 1 drivers and the quickest are level 4 drivers. In addition, some drivers serve as a bridge between JDBC and ODBC, mostly as an easy way to access ODBC databases (MS-something or other databases, usually).

A type 1 driver provides JDBC access using a JDBC-ODBC bridge. This bridge provides JDBC access to most ODBC drivers. Disadvantages of this type of JDBC driver include additional performance overhead of the ODBC layer, and the requirement to load client code on each client machine.

A type 2 driver is a partial Java driver that converts JDBC calls into the native client database API. As with the type 1 driver, this driver requires some client code to be loaded on each client machine.

A type 3 driver is a pure Java driver that translates JDBC calls into a database-independent network protocol. The database-independent protocol is implemented using a middleware server. The middleware server translates the database-independent protocol into the native database server protocol. Middleware vendors typically offer a type 3

driver. Because the driver is written purely in Java, it requires no configuration on the client machine other than telling the application the location of the driver.

A type 4 driver is a pure Java driver that uses a native protocol to convert JDBC calls into the database server network protocol. Using this type of driver, the application can make direct calls from a Java client to the database. A type 4 driver, such as Informix JDBC Driver, is typically offered by the database vendor. Because the driver is written purely in Java, it requires no configuration on the client machine other than telling the application where to find the driver.

As you might imagine, JDBC relies on a host of other J2EE API sets, such as JTA and JTS, to get the job done.

Remote Method Invocation and IIOP

Remote Method Invocation (RMI) enables a Java application to invoke methods on different Java Virtual Machines. RMI is an important API used for supporting distributed computing and has been supported in core Java since version 1.1. RMI enables a Java client application to communicate with a Java server application by invoking methods on that remote object. With RMI, the client gets a reference to a server object, and then it can invoke methods on that object as if it were a local object within the same virtual machine.

For server objects developed in other languages, you must employ other techniques such as using Java IDL with CORBA or RMI/IIOP (the Internet Inter-ORB Protocol) to access the server object.

Java Interface Definition Language

By using the Java Interface Definition Language (IDL), the Java programmer has access to CORBA objects. As previously mentioned, the Java programmer uses the "IDL to Java" compiler, called *idlj,* to generate Java code to interact with CORBA objects.

Listing 2-3 is an example of CORBA IDL taken from the CORBA Document Object Model specification.

Note   The Document Object Model (DOM) is the recommendation of the Worldwide Web Consortium (W3C) for expressing a document as a set of related nodes. A common use of DOM is to model XML documents. See Appendix D for an overview on XML. Refer to http://www.w3c.org/DOM for the definitive specification of the Document Object Model.

Listing 2-3: Example IDL code from the W3C DOM

```
interface Element : Node {
 readonly attribute DOMString       tagName;
 DOMString      getAttribute(in DOMString name);
 void          setAttribute(in DOMString name,
                 in DOMString value)
                 raises(DOMException);
 void          removeAttribute(in DOMString name)
                  raises(DOMException);
 Attr          getAttributeNode(in DOMString name);
 Attr          setAttributeNode(in Attr newAttr)
                 raises(DOMException);
 Attr          removeAttributeNode(in Attr oldAttr)
                 raises(DOMException);
 NodeList       getElementsByTagName(in DOMString name);
 void          normalize();
};
```

The idlj compiler produces Listing 2-4, the Java language binding for the IDL shown above.

Listing 2-4: Java code from the W3C DOM

```java
public interface Element extends Node {
 public String        getTagName();
 public String        getAttribute(String name);
 public void          setAttribute(String name,
                        String value)
                        throws DOMException;
 public void          removeAttribute(String name)
                        throws DOMException;
 public Attr          getAttributeNode(String name);
 public Attr          setAttributeNode(Attr newAttr)
                        throws DOMException;
 public Attr          removeAttributeNode(Attr oldAttr)
                        throws DOMException;
 public NodeList       getElementsByTagName(String name);
 public void          normalize();
}
```

J2EE Connector

The J2EE Connector provides a Java solution to the problem of connectivity among the many application servers and Enterprise Information Systems (EIS) already in existence. By using the J2EE Connector architecture, EIS vendors no longer need to customize their product for each application server. Application server vendors who conform to the J2EE Connector architecture do not need to add custom code whenever they want to add connectivity to a new EIS.

Before the J2EE Connector architecture was defined, no specification for the Java platform addressed the problem of providing a standard architecture for integrating heterogeneous EISs. Most EIS vendors and application server vendors use nonstandard vendor-specific architectures to provide connectivity between application servers and enterprise information systems.

Chapter 3: A First Look at JavaServer Pages

This chapter provides you with a bird's eye view of JavaServer Pages (JSP). You can read how to execute JSP and how JSP is intimately related to Java servlets. You can also see a couple of simple JSPs and read about what happens during the execution of these JSPs. This chapter continues with a brief discussion on the advantages and disadvantages of using JSP over several competing technologies and is followed by a recap of the material presented.

JavaServer Pages

JavaServer Pages (JSP) is one solution to providing dynamic Web content. The days of displaying the same old HTML page to all customers, or to the same customer, who has visited the site several times, is rapidly becoming a thing of the past. Today, Web pages need to display different content customized according to user input or relevant events.

Customers want and expect some sort of personalization from sites. A return customer does not want to be forced to reenter the same information when revisiting the site. Also, a Web page displaying data relevant to your inputs may need to differ from pages displayed for other users. Imagine an online banking site where you enter your password and see not only information on your accounts, but information for other bank customers as well!

Sites that change based on relevant events also provide a good example of dynamic content. Sites with stock market quotes or weather information need to be refreshed at regular intervals to be useful. News sites must also refresh content to stay on top of what's happening in the world. Stores that have online catalogs that often change inventory and prices should not contain static content. Today's Internet-related technologies, such as JavaServer Pages, give the Web application developer the means to create pages with dynamic content.

JSP combines static text with special JSP tags. The static text represents invariant parts of the Web page, typically but not necessarily HTML.

Note   JSP pages mostly use HTML and XML for the static, template portion. Rather than constantly writing "HTML or XML," I've taken the liberty of writing "HTML" in this chapter and trusting you to know if "HTML or XML" or "HTML" applies.

The JSP tags represent parts of the page that can change depending on the factors the page designer deems appropriate. The basic mechanics are that the static text and the JSP tags are eventually sent to a Java-enabled server that generates HTML from both the static part and the JSP tag. Once done, the server sends the generated HTML back to the browser for display and continued user interaction.

This approach of mixing static text with tags is not unique to JSP. Indeed, several competing technologies employ this approach. However, JSP enables you to leverage the full power of the Java programming language to make your pages very flexible. The pros and cons of JSP are discussed later in this chapter, in the section "JSP Versus the Competition."

Creating and Using JSP Pages

A special IDE is not required to create JSP pages. You don't develop JSP pages as you would a Java application or servlet. You don't have to wrap JSP pages in packages or deal with system settings (such as `CLASSPATH`). You don't even have to (but you could) compile JSP pages! All you need is a good Web page editor that enables you to easily enter the various JSP tags.

A site development team using JSP pages can have part of the team develop the static HTML portion, while others develop the dynamic portion. The HTML developers need not know how to code JSP pages, or know anything about programming in Java. But, as you might imagine, the JSP developer needs to be adept in coding HTML. When you recall that the end result of a JSP is a Web page containing generated HTML, how could any self-respecting JSP developer not be HTML-fluent?

It's simple to use a JSP page. The JSP page user must have access to a server that understands JSP tags, or a JSP-enabled server. To use a JSP page with such a server, you enter the name of the page as you would any Web page in the location area of your browser. A file representing a JSP page has a `.jsp` extension, which a JSP-enabled server recognizes as a JSP page and, in turn, processes the special tags as JSP tags.

Note   The term "JSP page," although redundant, enjoys widespread use among the JSP development community. Hence, the term is used throughout this book.

For Web pages that submit a JSP page to the server with a `GET` or `POST` service, the customer may never realize that JSP pages are in play on the site. The `ACTION` attribute of an HTML form may specify that the action upon submitting the form is to send the name of a JSP page with one or more parameters to the server. Again, the JSP-enabled server recognizes the `.jsp` extension and takes appropriate action.

The Relationship Between JSP Pages and Java Servlets

The simple mechanics of creating and using JSP pages masks the complexity of the under-the-covers activity. JSP pages actually are compiled into Java servlets. All those environment issues dealing with compiling and executing servlets come into play. Whereas you don't compile JSPs, your Java-enabled server performs the compilation from JSP page into a Java servlet for you. Although you, the JSP developer, need not care about `CLASSPATH` and other settings, your server needs to know these settings. Your server needs access to the Java compiler and various classes required for servlet and JSP compilation.

The first time you request a JSP page, the server translates the page into a Java class. Recall from [Chapter 1](#) the concept of J2EE *containers.* The JSP-enabled server has a *JSP container* that provides the environment necessary for this translation. Sometimes, the JSP container is called the *JSP engine;* both terms are used interchangeably in this book.

The server compiles the class generated by the JSP engine into a servlet. This servlet contains Java `println` statements that write the static text to the output stream, and Java code that implements the functionality of your JSP tags. Depending on the amount of Java code generated by the JSP and the speed of the server, you may notice a slight delay during the JSP-to-servlet compilation. However, subsequent requests of the JSP page do not cause a page retranslation and recompilation. The JSP request accesses the already compiled servlet in memory.

As an aside, some servers enable you to establish file aliases. You can avoid the delay caused by the first-time JSP translation and compilation by requesting your JSP page (causing translation and servlet generation), followed by creating an alias of your JSP page to the generated servlet. Now, when your customer requests your JSP page, the server references the previously generated servlet, which is already compiled and in memory.

JSP Versus the Competition

As previously mentioned, JSP is not the only technology available to the Web application developer that generates
dynamic Web output. As with any technology, JSP has its advantages and disadvantages. This section describes
what JSPs can offer that competing technologies cannot.

Separating Logic from Presentation

As you read previously, coding business logic apart from presentation is a good design feature. Do you recall reading
about multitier architectures from Chapter 1? When properly separated, the code that implements the business logic
can be changed without affecting the code that implements the presentation, and vice-versa. JavaServer Pages give
the Web developer the ability to *cleanly* separate the logic from look-and-feel.

JSP enables Web developers to encapsulate the business logic in custom JSP tags (discussed in Chapter 7, "JSP
Tag Extensions") and Java software components, such as JavaBeans and Enterprise JavaBeans. The code,
implementing the logic, is tied together with JSP scriptlets, expressions, and other JSP tags, which haven't been
discussed yet, and is sent to the Web server for execution.

This separation enables developers to practice their particular specialty; the skilled HTML author has no need to learn
JSP and the JSP author doesn't need to be an HTML maven. The HTML author can concentrate on coding HTML
(presentation) tags and the JSP developer can concentrate on coding JSP (logic) tags.

The Strength of Java

Because JSP pages eventually are translated and compiled into Java servlets, you can use JSP pages on any server
that supports Java. You are not tied to any particular vendor or platform when you go the JSP route.

Of course, you have full command and control of the Java programming language when you use JSP. JSP make
extensive use of Java Beans and can communicate with other J2EE technologies, such as JDBC and, of course,
Enterprise JavaBeans.

JSP Versus Java Servlets

Before the advent of JSP, the most-used Java technology that could generate dynamic Web page content was Java
servlets. Because JSPs eventually are compiled into Java servlets, you can do as much with JSPs as you can do with
Java servlets. However, coding JSPs is easier than coding Java servlets. With JSPs, you place static text by coding
HTML tags as opposed to Java servlets, in which you place static text by coding a plentitude of `println` statements.
With JSPs, you change static text by changing HTML; and with Java servlets, you change static text by modifying a
Java servlet (don't forget the compile/debug cycle!).

JSP Versus Active Server Pages

Active Server Pages (ASP) is the Microsoft solution for providing dynamic Web content. Actually, ASP looks very similar to JSP; both use custom tags to implement business logic and text (HTML) for invariant Web page parts. However, the devil is in the details, as described in the following:

- ASP uses VBScript or JScript, a Microsoft flavor of JavaScript, as its scripting language, whereas JSP uses Java, a more powerful language than VBScript or JScript.

- The ASP developer typically uses a Microsoft Web server platform or requires a third-party product that permits ASP execution on non-Microsoft platforms. The JSP developer has a wide variety of Web server platforms available for use.

Note   These third parties must port Microsoft software components, such as ActiveX, to different platforms in order for ASP to be used on these platforms.

- An ASP is interpreted every time the page is invoked, whereas a JSP is interpreted only the first time the page is invoked (or when the page is changed).

However, Microsoft has overcome the previously mentioned limitations of ASP with its release of ASP.NET. ASP.NET, formerly ASP+, promises to be a serious contender against JSP. As of this writing, you may download the ASP.NET Beta-2 release from http://www.asp.net/.

JSPs Versus Client-Side Scripting

Client-side scripting with JavaScript or VBScript is certainly handy and useful, but it does present several problems, including the following:

- You must count on the customer's browser to have scripting enabled, which, of course, you can't.

- Different customers may use different browsers. And coding client-side scripts that work on different browsers can be a headache.

- Scripting languages used on the client side cannot match the strength and versatility of Java.

- Client-side scripting languages have very limited access to server-side resources, such as databases. JavaServer pages have access to all server-side resources within the well-defined architecture of J2EE.

- You have the usual problems of maintaining software on the client that caused your organization to thin the client in the first place.

In short, the advantages of using JSP over competing technologies are as follows:

- JSP enables a clean separation of business logic from presentation.

- JSP, by using Java as the scripting language, is not limited to a specific vendor's platform.

- JSP, as an integral part of the J2EE architecture, has full access to server-side resources. Because JSP pages execute on the server, you need not require the client to use a particular browser or have a fixed configuration

Disadvantages of Using JSP

What technology doesn't have problems? Certainly, JSP technology has room for improvement. That said, what one Web application developer views as a weakness, another may view as a strength (remember "bug" versus "features"?). Here is a (short) list of real or perceived shortcomings of JavaServer Pages:

- JSP implementations typically issue poor diagnostics. Because JSP pages are translated, and then compiled into Java servlets, errors that creep in your pages are rarely seen as errors arising from the coding of JSP pages. Instead, such errors are seen as either Java servlet errors or HTML errors. You could look at this as an example of a perceived strength of JSP — that of not needing to compile them — as opposed to a weakness. For example, a JSP developer coding a scriptlet where a JSP declaration is called for would have to interpret a Java compile error. The JSP developer would need access to the generated source to properly diagnose the error. Of course, generated code is rarely a thing of beauty, and often, not easily understood.

- The JSP developer needs to know Java. Again, one developer's asset is another's liability. Whereas Java is certainly more full-featured and flexible than other page scripting languages, no one can argue that the learning curve for Java is far steeper than other scripting languages. If you already know Java (you do, right?), this is not an issue. However, if a corporation is short on Java mavens but wants to use a dynamic Web technology, JSP may not be the route to go. (Another way to look at the need to know Java is that if you had to train a rookie in using either JSP or, say, ASP, and you had two days to produce half a dozen pages, which technology would you opt for?)

- JSP pages require about double the disk space to hold the page. Because JSP pages are translated into class files, the server has to store the resultant class files with the JSP pages.

- JSP pages must be compiled on the server when first accessed. This initial compilation produces a noticeable delay when accessing the JSP page for the first time. The developer may compile JSP pages and place them on the server in compiled form (as one or more class files) to speed up the initial page access. The JSP developer may need to bring down the server to make the changed classes corresponding to the changed JSP page.

All in all, it's a pretty short list.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Chapter 7: JSP Tag Extensions

Overview

At this point, you've been introduced to JSPs containing Java code in the form of scriptlets and expressions, custom JSP constructs such as directives, and JSP action tags, which include other JSP pages or which enable your JSPs to access JavaBeans. It certainly looks as if JSPs have covered all the bases for enabling JSP authors to generate dynamic Web content.

You may recall that one goal of using JSPs is to clearly delineate presentation details from business logic. As you write more scriptlet code in your JSPs, the delineation begins to blur. If you're not careful, your JSPs may contain more business logic code than presentation details. This needs to be avoided for two reasons. First, JSP authors may not be Java experts and hence won't be able to maintain the Java code contained in these constructs. Second, the coupling of business code and presentation code makes it harder to change either independently.

JSP tag extensions allow you to add functionality to JSP pages without having to add many Java scriptlets to your pages. In this chapter you will learn what JSP tag extensions are, what you can do with them, and how to create them.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

In this chapter, you examined the code for a simple JavaBean and saw how this bean is used in some JSP pages. In
addition, you also explored several JSP statements that you can use to transfer data between beans and JSP pages,
and you learned that you can create and share beans among several JSPs.

By now, you have a good understanding of a powerful feature of JavaServer Pages — the feature of using software
components within your pages.

Bean Usage Odds and Ends

A few points about using beans in JSP pages haven't been discussed yet. You probably surmised that you can use more than one bean instance from the same or different bean classes in your JSPs. All you need to do is code an appropriate jsp:useBean action for each bean instance.

When the JSP engine encounters a `jsp:useBean` action, the JSP engine searches for a bean with the same `id` and `scope` as the bean cited in the `jsp:useBean` action. If such a bean is *not* found, the JSP engine generates code to create the bean. If such a bean *is* found, the JSP engine makes that bean available to the page containing the `jsp:useBean` action. Beans may have the same `id` but be instantiated from different classes. If so, the JSP engine generates code to do a cast. If the cast is illegal, the generated servlet throws a `ClassCastException`.

As a result of this bean usage, multiple visits to the same page containing a `jsp:useBean` action during the same session will not create multiple beans. Another consequence of this use involves conditionally executing JSP commands, as explained in the following.

The examples of `jsp:useBean` you've seen use the empty tag XML syntax form. However, you can code the `jsp:useBean` construct by using separate opening and closing tags as follows:

```
<jsp:useBean id=someBeanName… >
  Static HTML, JSP commands, whatever…
</jsp:useBean>
```

The virtue of using separate opening and closing tags is that the code sandwiched between the tags is executed *only* if the bean instance does not exist. If you want to share a bean among several pages, you can place the same code in every page, knowing that the code gets executed once, depending on where the bean gets created. Remember, JSP knows the bean by the values of the `jsp:useBean` attributes `id` and `class`. Different beans (objects) may be the "same" bean to JSP.

Using JavaBeans in Multiple JSPs

The syntax of the `jsp:useBean` action has a couple of additional parameters, as follows, which have not been
mentioned:

```
<jsp:useBean    id="beanInstanceName"
                class="className"
                scope="beanScope"
                type="classType" />
```

The `scope` attribute is the focus of our discussion. The value of the `scope` attribute governs the bean's visibility.
Different values for the `scope` attribute place instantiated beans within different contexts. Refer to Table 4-3 in
Chapter 4 for a list of scope attribute values and the relevant contexts.

The default `scope` attribute value is *page*. Page scope means that bean instances are accessible on the existing JSP,
or the JSP containing the `jsp:useBean` action. Stated differently, other JSPs within your application cannot use
instantiated bean objects without containing a `jsp:useBean` action.

Trying with page Scope

To bring the point home, here's the JSP page `calculate.jsp` with a few changes. The following lines of code are
the substantial change:

```
<jsp:useBean id="CalcBean"  class="cbean.CalcBean" scope="page"/>
```

Because `scope="page"` is the default, this code really doesn't change the JSP's behavior. The next line invokes
another JSP, coded remarkably similar to `calculate.jsp`.

```
<a href=calculate2.jsp>Click Here for Next JSP Page</a>
```

Figure 6-3 shows `calcpage.html`, the page that calls `calculate.jsp`, and the revised `calculate.jsp`:
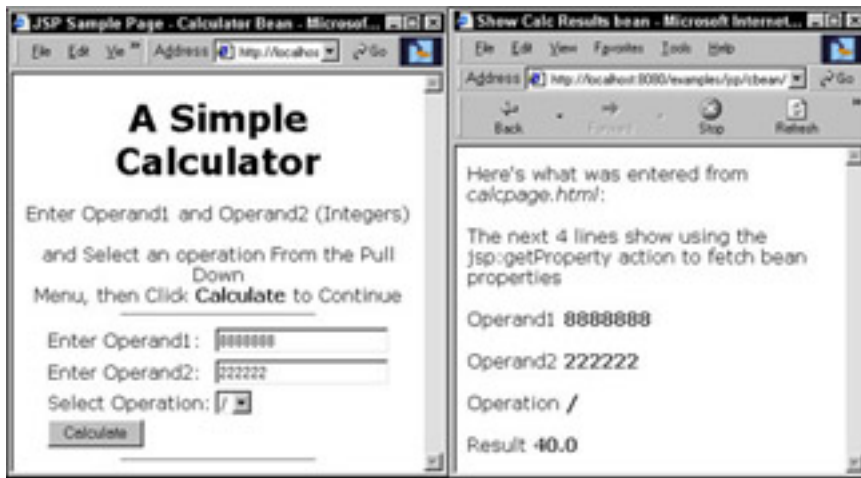
Figure 6-3: The calcpage.html and the revised calculate.jsp that contain a link to calculate2.jsp

Keep in mind that `calculate.jsp` has the `jsp:useBean` action coded such that bean instances are known only within the page. Listing 6-4 contains the code for `calculate2.jsp`.

Listing 6-4: Accessing CalcBean in a second JSP page

```
<%-- Tell JSP that this page renders HTML --%>
<%@ page contentType="text/html" %>
<%-- Tell JSP to map CalcBean properties to like-named variables --%>
<jsp:setProperty name="CalcBean" property="*" />
<html>
<head>
<title>Calculate 2 Page</title>
</head>
<body bgcolor="#dddddd">
Here's what was entered from <i>calcpage.html</i>: <p>
<P>The next 4 lines show using the jsp:getProperty action to fetch bean properties
<%-- jsp:getProperty writes the value of the bean property where coded --%>
<p>Operand1 <b><jsp:getProperty name="CalcBean" property="operand1" /></b>
<p>Operand2 <b><jsp:getProperty name="CalcBean" property="operand2" /></b>
<p>Operation <b><jsp:getProperty name="CalcBean" property="operation" /></b>
<p>Result <b><jsp:getProperty name="CalcBean" property="result" /></b>
</body>
</html>
```

The `calculate2.jsp` file refers to the bean instance `CalcBean`. However, `calculate2.jsp` does not contain a `jsp:useBean` action. Because the instance of bean `CalcBean` created in `calculate.jsp` has `page` scope, you would expect problems to occur when `calculate2.jsp` attempts to access properties of the bean. Figure 6-4 illustrates what Tomcat indicates when you click the link and tell the JSP translator to process `calculate2.jsp`.
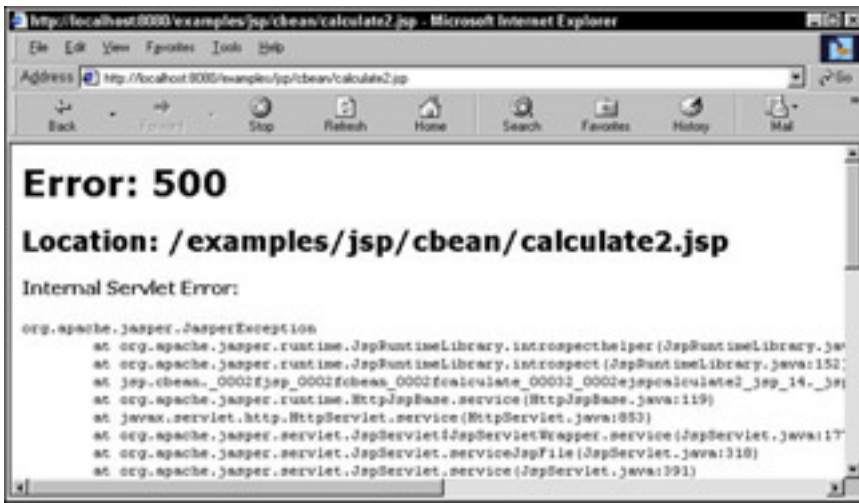
Figure 6-4: Tomcat tells you off.

You do not need the entire stack trace; the essential message is (as usual) on the first line of the trace.

Trying with application Scope

We saw in the previous section, "Trying with page Scope", that if we set the value of scope to `page` in `calculate.jsp`, CalcBean is no longer accessible when we attempt to access it in `calculate2.jsp`. Let's change the value of `scope` from `page` to `application` as follows:

```
<jsp:useBean id="CalcBean"  class="cbean.CalcBean" scope="application"/>
```

Using the `application` scope places the instantiated beans in the *Servlet Context,* which makes the bean accessible to any servlet running on the server. Figure 6-5 shows what you see after shutting down Tomcat, restarting Tomcat, displaying `calcpage.html`, entering some numbers, and clicking "Calculate."
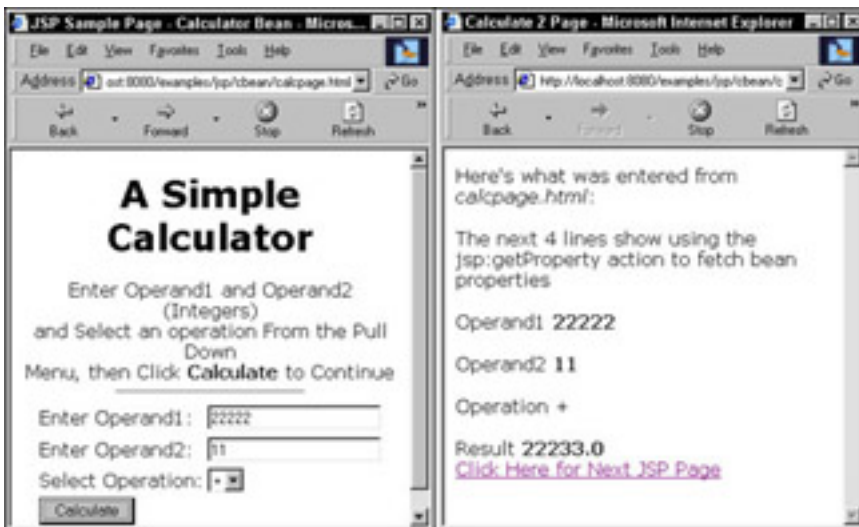
Figure 6-5: Running the calculator again with application scope

No news here. You may recall that the JSP page `calculate.jsp` has its `jsp:useBean` action coded to enable application access to bean instances in general and the bean instance named in the `useBean` action in particular. When you click the link for the next (`calculate2.jsp`) JSP page, Figure 6-6 is what you see.
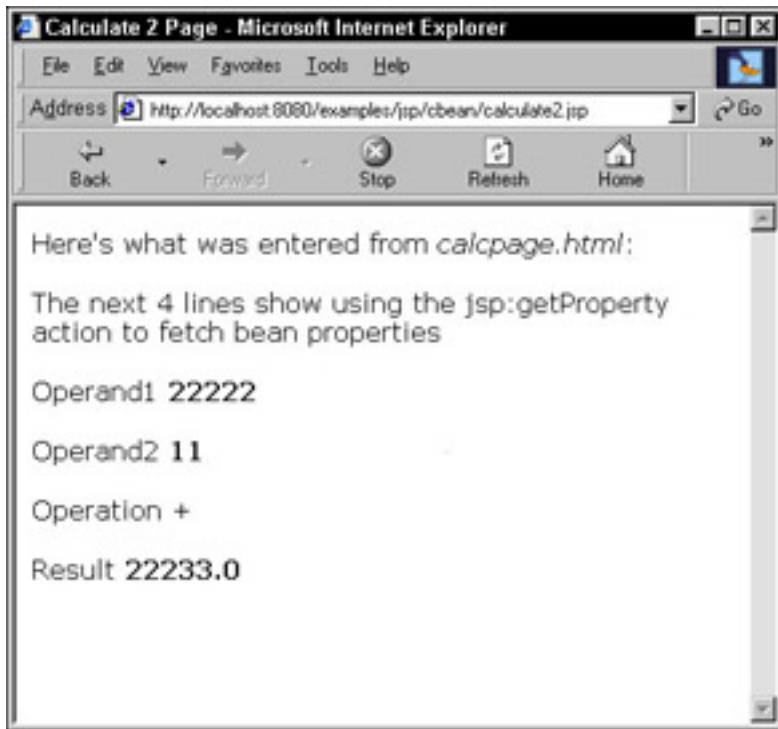
Figure 6-6: Accessing a bean instance from a page that didn't create the instance

Refresh your memory by glancing at to note that `calculate2.jsp` does not contain a `jsp:useBean` action.

Other scope Attribute Values

The scope attribute of the `jsp:useBean` action may also have a value of *session* or *request*. When you code `scope="session"`, you are telling the JSP engine to store your bean instance in the `session` object. You can turn off sessions with a JSP page directive containing a `session` attribute with the value of "false." A `useBean` scope value of `session` is incompatible with a page directive `session` value of false.

The remaining scope value is `request`. You've seen input data access with the `request` implicit object. Coding a bean with scope of `request` stores the bean instance within this implicit object.

Coding a bean scope of `request` accomplishes little. You get the same access to the `request` object with the default scope of `page`.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Coding JSP Standard Actions

Actions cause something to happen. In JSP, you have two categories of actions at your disposal: *custom* actions and *standard* actions. You implement custom actions with tag libraries. You'll have to wait until you get to Chapter 7 for the nitty-gritty on tag libraries. You don't implement standard actions; these actions are instead provided to you, the JSP developer, free of charge. Please note that a server provider may provide vendor-specific actions above and beyond the standard set.

Table 4-2 shows the JSP standard actions, an example of the syntax, and a brief description of each action.

Table 4-2: Short Descriptions of JSP Standard Actions

Action Name	Syntax Example	Short Description

`jsp:param`
and
`jsp:params`
	`<jsp:param name=@@dpparamName@@dp`
	`value=@@dpparamValue@@dp />`
		This action works with other standard action tags (`include`, `forward`, and `plugin` tags) to provide a value to a named parameter.

`jsp:forward`	`<jsp:forward page=@@dpsomeURL@@dp>`
	`….`
	`</jsp:forward>`
		This action provides a convenient mechanism to forward a request to another JSP or servlet.

jsp:getProperty	`<jsp:getProperty`	The JSP author uses the `getProperty` action to access the properties of a bean coded in a `useBean` action. The `getProperty` action is the compliment to the `setProperty` action.
	`                name=@@dpbeanInstanceName@@dp`	
	`                property=@@dppropertyName@@dp />`	
jsp:include	`<jsp:include page=@@dppageName@@dp flush=@@dptrue@@dp />`	The JSP author codes the `include` action to direct the engine to include a resource at request time. Do not confuse this *action* with the JSP `include` *directive*.
jsp:plugin	`<jsp:plugin type=@@dpapplet@@dp`	The JSP author codes the `plugin` action when he or she needs to generate client-specific HTML `OBJECT` or `EMBED` tags that ensure that a particular object is available and to invoke that bean or object. Most of the attributes are identical in function and coding to the HTML attributes for the `OBJECT` tag.
	`                code=@@dpjavaCode@@dp`	
	`                codebase=@@dpjavaClasses@@dp`	
	`                align=@@dpalignment@@dp`	
	`                archive=@@dpjarFiles@@dp`	
	`                height=@@dppixelsHigh@@dp`	
	`                width=@@dppixelsWide@@dp`	
	`                jreversion=@@dp1.2@@dp`	
	`                name=@@dpcomponentName@@dp`	
	`                title=@@dpcomponentTitle@@dp`	
	`                vspace=@@dppaddingAround@@dp`	
	`                nspluginurl=@@dpwhereNSPluginsAre@@dp`	
	`                iepluginurl=@@dpwhereIEPluginsAre@@dp`	
	`Show This When Applet or Bean Fails to Load`	
	`<jsp:fallback>`	
	`</jsp:fallback>`	
	`</jsp:plugin>`	

`jsp:setProperty`	`<jsp:setProperty name=@@dpbeanName@@dp`
	
	`property=@@dppropertyName@@dp`
	
	`param=@@dpparamName@@dp />`
	`<jsp:setProperty name=@@dpbeanName@@dp`
	
	`property=@@dppropertyName@@dp`
	
	`value=@@dpscriptletOrStringValue@@dp`
	`/>`
		The JSP author uses the `setProperty` action with the `useBean` action to set the values of properties in the beans named in the `name` attribute. The bean properties are coded in the `property` attribute; the value can be a string or scriptlet coded in the `value` attribute.

`jsp:useBean`	`<jsp:useBean id=@@dpbeanInstanceName@@dp`
	
	`scope=@@dppage@@dp`
	
	`class=@@dpclassName@@dp`
	
	`type=@@dpclassType@@dp />`
		This action allows the JSP author to use objects instantiated from a JavaBean. The `scope` attribute may be `page`, `request`, `session`, or `application`. The `useBean` action works with other actions described earlier.

The standard action commands are coded as *tags* following XML syntax rules. (See Appendix D for information on XML syntax.) In the sections that follow, you learn more about the set of standard actions available to the JSP developer on any JSP Web server.

The param and params Action

The `param` action provides other tags with parameter data. Use `param` to get data to the `forward`, `plugin`, and `include` actions. The syntax for the `param` action is as follows:

```
<jsp:param   name="parameterName" value="parameterValue" />
```

If you have a need to create more than one parameter name-value pair for use in another action, you need to enclose the multiple `param` actions inside a `params` action, as follows:

```
<jsp:params>
<jsp:param   name="parameterName1" value="parameterValue1" />
<jsp:param   name="parameterName2" value="parameterValue2" />
<jsp:param   name="parameterName3" value="parameterValue3" />
</jsp:params>
```

The forward Action

The `forward` action causes processing to immediately redirect to the indicated page. For example, when processing hits the following statement:

```
<jsp:forward    page="thenextpage.html" />
```

`thenextpage.html` is immediately displayed.

Before displaying the forwarded page, the output stream buffer (if one exists) will be cleared. If you want to make a name-value parameter known to the forwarded page, you use the `param` action as follows:

```
<jsp:forward    page="thenextpage.html" >
    <jsp:param   name="paramName"    value="paramValue" />
</jsp:forward>
```

Using the `forward` action enables you to direct categories of activities to specific pages.

The getProperty and setProperty Actions

The `getProperty` action accesses one or more properties of a JavaBean used by the JSP page. The `getProperty` action accesses the value of `property` from a JavaBean, converts the value to a string, and writes the string representation to output.

The `getProperty` action has the following syntax:

```
<jsp:getProperty name="beanInstanceName"
                 property="propertyName />
```

As you might imagine, `setProperty` is how the bean gets the property value set in the first place. The syntax for the `setProperty` action has several forms, as shown here:

```
<jsp:setProperty name="beanInstanceName" property="*" />

<jsp:setProperty name=" beanInstanceName"
                 property="propertyName" />

<jsp:setProperty name=" beanInstanceName"
                 property="propertyName"
                 param="parameterName" />

<jsp:setProperty name=" beanInstanceName"
                 property="propertyName"
                 value="propertyValue" />
```

The attribute `propertyName` is the name of the bean property you want to set.

The attribute `propertyValue` is a string or JSP expression that, of course, represents the value of the property you wish to set.

The attribute `paramName` is the value of a parameter that replaces the existing value of the property coded in the `setProperty` action.

You cannot code both `param` and `value` in the same `setProperty` action.

The include Action

The `include` action enables you to include content in your JSP page. Before you think that the `include` action is the same thing as the `<%@ include %>` directive, recall that the `include` directive brings in the external content *at translation time* whereas the `include` action is processed at *runtime* (or page request time).

The syntax of the `include` action is straightforward, as shown here:

```
<jsp:include page="relativeURL" flush="true" />
```

When considering the `include` action, it's important to note that the value of the `page` attribute can be a JSP expression or some other dynamically generated expression.

The `flush` attribute must be coded as true.

You can code `param` tags with the `include` action, as shown here:

```
<jsp:include   page="included.html" flush="true" >
   <jsp:param  name="paramName"    value="paramValue" />
</jsp:include>
```

The plugin and fallback Actions

You use the `plugin` action to generate HTML tags for embedding Java applets in the generated output page to ensure that the browser contains an appropriate Java runtime, and that it executes the applet properly.

All but 4 of the 13 attributes of the `plugin` action have the same meaning as the HTML counterparts. The parameters that have different meaning are:

- `type`: Identifies the type of the component; a bean or an applet

- `jreversion`: Java runtime required to execute the component

- `nspluginurl`: Location of the Netscape JRE download, as a URL

- `iepluginurl`: Location of the Internet Explorer JRE download, as a URL

The `plugin` action takes an optional `param` action as well.

You may code a `fallback` action to provide information when the `plugin` fails to load. Basically, the `fallback` action provides alternate text that performs the same function as the `ALT` attribute.

The useBean Action

The `useBean` action is used to make a JavaBean known to your JSP. You read more about bean use with your JSPs in , "JSP, JavaBeans, and JDBC." In this section, you get exposure to the syntax for the `useBean` action.

There are several forms for coding the `useBean` action, as shown here:

```
<jsp:useBean    id="beanInstanceName"  scope="contextScope"
                class="className" />

<jsp:useBean    id="beanInstanceName"  scope="contextScope"
                class="className"     type="typeName" />

<jsp:useBean    id="beanInstanceName"  scope="contextScope"
                beanName="beanName"   type="typeName" />

<jsp:useBean    id="beanInstanceName"  scope="contextScope"
                type="typeName" />
```

The attributes coded for `useBean` have the following meaning:

- `id`: The name of the `bean` object instance.

- `scope`: A context in which the `bean` reference is known. The different `scope` contexts are represented by implicit objects, covered more fully in the . Think of the `bean` object as having a different life cycle for different `scope` values. lists the values and meaning of the `scope` attribute of the `useBean` action.

- `class`: The fully qualified class name of the bean being associated with the JSP.

- `beanName`: The same name you would use to instantiate the bean, or the name you would supply to the `instantiate` method of `java.beans.Beans`.

- `type`: Defaults to the value of the `class` attribute but can be a valid superclass or interface implemented by the bean class.

Table 4-3: Values of the Scope Attribute Used in the JSP useBean Action

Scope	Description
`page`	The bean object dies after the servlet completes its `service()` invocation.
`request`	The bean lives for as long as the HTTP request lives, even if the HTTP request object is passed among different JSP pages.
`session`	The bean object lives as long as the session exists.
`application`	The bean object lives for the duration of the application's execution.

As you might imagine, the `useBean` action enables you to load JavaBeans for use in your JSP pages, thereby opening your JSP pages to the full power of the Java programming language. Also, you can take advantage of using software components, something that you cannot easily do with other products.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Appendix D: XML Overview

Throughout this book, you've read about JSP elements and EJB files coded in XML syntax. Although the examples of such elements and files presented in the book convey the essential flavor of XML syntax, a more thorough presentation is called for. Thus, the purpose of this appendix is to present the essentials of XML syntax.

This appendix provides an overview of XML, or Extensible Markup Language, a universal document format for structuring data for presentation on the Web. The appendix starts with an overview of XML features that overcome existing problems with HTML. Next, an extremely simple XML document is provided along with a discussion of XML document components. The important XML terms, well-formed documents and valid documents, are covered, as are XML Document Type Definitions (DTDs). Finally, a brief description of related technologies wraps up this appendix.

XML Features

XML does not have a fixed set of markup tags, overcoming HTML's greatest deficiency, according to some experts. XML is not a markup language per se; XML is a meta-markup language that enables document authors to define their own tags. As a result, authors can create markup languages peculiar to their particular industries, and XML document authors can use this markup language to encode data in industry-specific terminology.

XML requires document authors to follow certain rules in creating what is known as well-formed XML documents. If these rules are not followed, the XML document is useless. This XML specification prohibits XML tools from trying to fix problems with the document. The intent is to stop the browser madness prevalent in HTML, in which different browsers attempt to "fix" broken HTML and, of course, parse and display this HTML differently. For example, an HTML document author can write HTML with missing end tags, which the major browsers parse and display. Such foolishness cannot fly with XML; if an XML document is broken, the document cannot be rendered. Therefore, an XML author can confidently create XML documents, knowing that these documents are parsed identically with different pieces of compliant software.

XML stresses the separation of data content from data presentation. Over time, HTML has blurred the distinction between organizing document content and displaying the content. A typical HTML document has tags that describe relationships among document content (such as `<LI>` tags) and tags that govern the display of this content (`<U>`, `<B>`, and so on). XML describes document content structure and semantic relationships, not the content formatting. The XML author uses a related style sheet technology, such as CSS (Cascading Style Sheets) or XSL (Extensible Style Language), to govern the display of the document. One upshot of this clean separation of structure and display is that the same XML document can be displayed in various ways by using different style sheets, or the same style sheet can govern the display of similarly structured XML documents.

The nonproprietary nature of XML, combined with its ease of writing, makes XML an ideal format for data exchange among applications.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

There are many JSP engines on the market today, and you may or may not use Tomcat in your JSP projects. Since Tomcat is the reference JSP implementation endorsed by Sun Microsystems, it will support new JSP specifications more quickly than most other JSP engines. For this reason you will want to become familiar with Tomcat whether you use it regularly or not because it will be a valuable tool for both regular use and for examining new JSP releases now and in the future.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Appendix C: Configuring the Tomcat Web Server

This appendix discusses Tomcat and provides advice on how to get the Tomcat release 3.2 Web server. Tomcat is the Reference Implementation for the Java Servlet 2.2 and JavaServer Pages 1.1 Technologies. In other words, Tomcat is a Web server that implements the most current release of JSP and Java Servlets.

The Tomcat Web Server

Tomcat is a Web server that contains a JSP container. Tomcat is quick and easy to install and to use, and offers the following advantages:

- Tomcat is free.

- You can download the *source* code for Tomcat as well as the binaries. If you really want to learn about server internals, the Tomcat source is a great resource.

- Mailing lists about Tomcat are available to one and all. These lists are devoted to disseminating information, including posted questions and answers.

Tomcat is designed to work as both a standalone Web server or in conjunction with application servers, such as Apache or JBoss. For your purposes here, it is more interesting to run Tomcat as a standalone Web server because it offers a straightforward way to learn about JSPs. However, in the real world, if you opt to use Tomcat, you may want to integrate Tomcat with another Web application server.

Tomcat was, and currently is, developed by a community of dedicated individuals under the umbrella of the *Jakarta* Project. You are encouraged to learn about the Jakarta Project by taking a look at http://jakarta.apache.org/index.html. From this site, you can get to the page where you can download the Tomcat Web server.

As of this writing, the latest release of Tomcat is release 3.2.

Note   By the time you read this book, it is very likely that the Jakarta folk will have a more recent version of Tomcat available, probably release 4.0. Be advised that the description of Tomcat given here applies to release 3.2. Several details and particulars may change between successive releases.

Downloading Tomcat Release 3.2

You can get to the download page for Tomcat from the Jakarta URL or go to http://jakarta.apache.org/site/binindex.html. (If you prefer, you can get to the *source code* download from here as well.)

The Tomcat download site classifies downloads into *release* builds*, milestone* builds, and *nightly* builds*.* The release builds are the stable versions of the Tomcat product. Once you get to the download site, scroll to the *release builds* section and select "Tomcat 3.2.1" (or whatever is the most current release being offered). Figure C-1 shows the screen containing the download files.
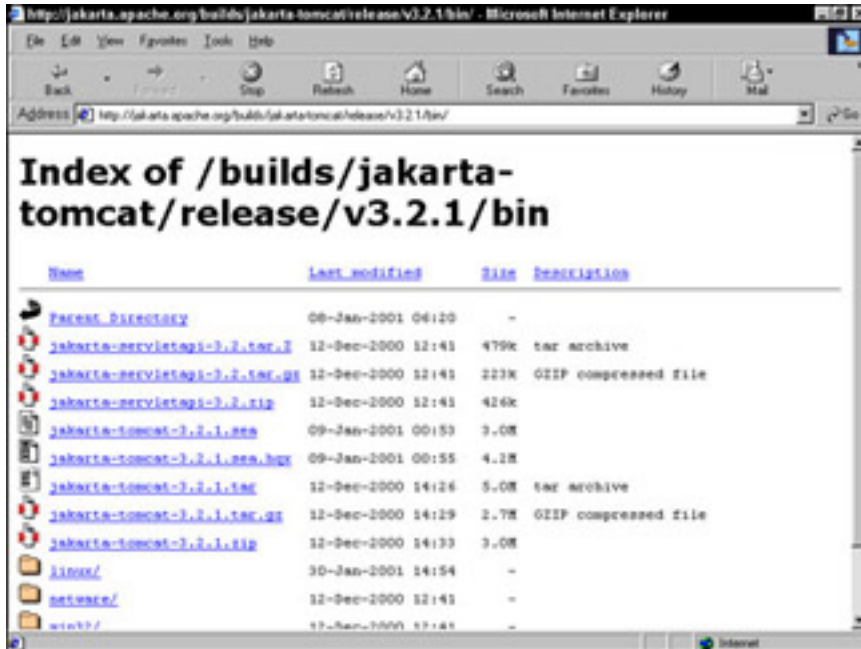
Figure C-1: Tomcat release 3.2 download page

From this page you can select a Tomcat version for Windows 9*x*, Windows NT, or Windows 2000, or for various UNIX flavors.

Assuming you are using some version of Windows, select `jakarta-tomcat-3.2.1.zip` to commence the download. After a successful download, you should have a zip file called `jakarta-tomcat-3.2.1.zip` on your hard drive.

Installing Tomcat

Installing Tomcat is a straightforward process. Just open the zip file and extract all contained files in the archive.

Note   You need a copy of PKZIP to extract the contained files. You can get a copy at
        `http://www.pkware.com/`. This site has compression/extraction tools for several operating systems.

You may want to create a directory at your disk root; for example, use `d:\tomcat32` to hold all the Tomcat files. After extracting the files, your directory structure should resemble that shown in Figure C-2.
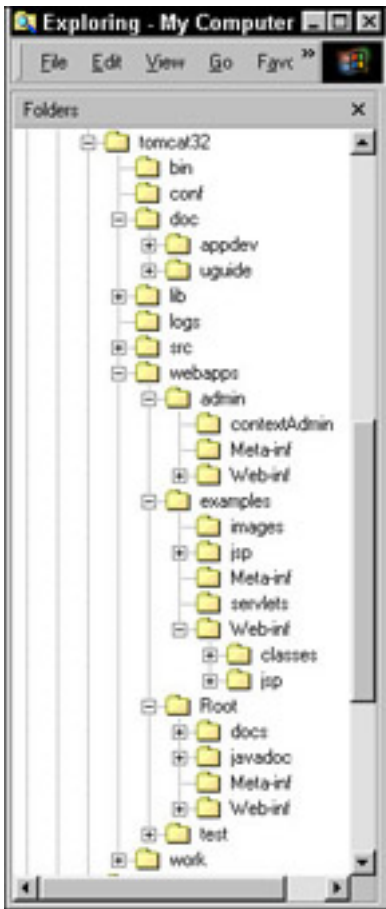
Figure C-2: Tomcat directory structure

As you can see, several directories shown in the figure have subdirectories. Not all the subdirectories within Tomcat can be shown here because of space limitations. Throughout this appendix we will examine different directories in the Tomcat installation that are relevant to the Tomcat configuration and JSP processing.

Assuming you see something similar to Figure C-2 on your screen, you are almost ready to check whether your Tomcat installation was successful. However, first you need to set up a few environment variables, as described in the following section.

Setting Your Tomcat Environment Variables

Tomcat uses a script located in the `bin` directory called, appropriately enough, `startup.bat`, which requires several environment variables to be set on your system for proper execution. Table C-1 lists these variables and their purpose.

Table C-1: Tomcat Environment Variables

Variable Name	Purpose
JAVA_HOME	Points to the root directory of your Java installation.
TOMCAT_HOME	Points to the root directory of your Tomcat installation.

If you are running Windows 9*x*, you can assign these environment variables in your `autoexec.bat` file as follows:

```
set JAVA_HOME=d:\jdk1.3
set TOMCAT_HOME=d:\tomcat32
```

After you code the assignments, you need to restart your machine or execute your `autoexec.bat` file to make the variable assignments.

If you are running Windows 2000 or Windows NT, you can use the System Properties control panel to set these environment variables. From your Start button, select Settings @@> Control Panel @@> System @@> Environment. Figure C-3 shows the Environment control panel.

Figure C-3: The Windows NT Environment control panel

If you find existing entries for environment variables in the control panel, you may click the entry to view and edit its value as necessary. Click OK or Apply to set the variables.

Caution   Be careful when setting environment variables in command windows. Windows starts a separate process for each command window opened. Therefore, if you issue a SET command to assign environment variables values in one window, the variables have these values when executing programs from within this command window only. Placing your SET commands in your `autoexec.bat` file or setting variable values in the control panel makes the values known to all processes.

Okay, you're almost home. With the Tomcat environment variables set, you are ready to test your Tomcat installation.

Testing Your Tomcat Installation

The directory `TOMCAT_HOME\bin` contains several startup and shutdown files. The files you are interested in are called `startup.bat` and `shutdown.bat`. First, run the startup file by double-clicking its icon or running the file from a command window. Once you execute the startup file, you should see two windows that resemble those shown in Figure C-4.
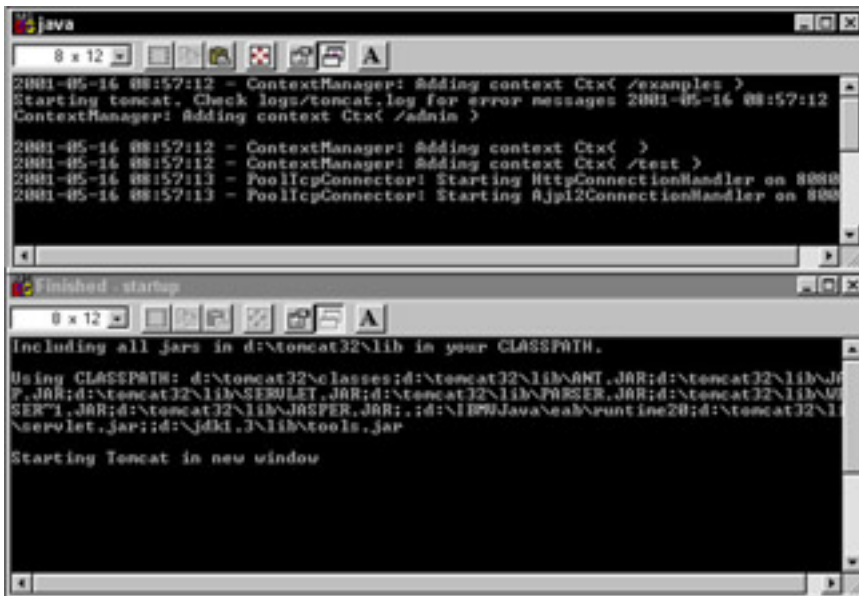
Figure C-4: What you should see when executing startup.bat

The windows in Figure C-4 are stacked for display purposes, so your display will differ. If you see two command windows such as the ones shown, you can be pretty sure that your Tomcat installation was successful.

You want to display Web pages in your Web server, right? Although the command windows shown in Figure C-4 show Tomcat executing, you need to call up a Web page and check out some servlets and JSPs. This process is covered in the following sections.

Displaying Web Pages in Tomcat

Open a browser and enter the following URL:

```
http://localhost:8080/
```

Figure C-5 shows what you should see on your screen.
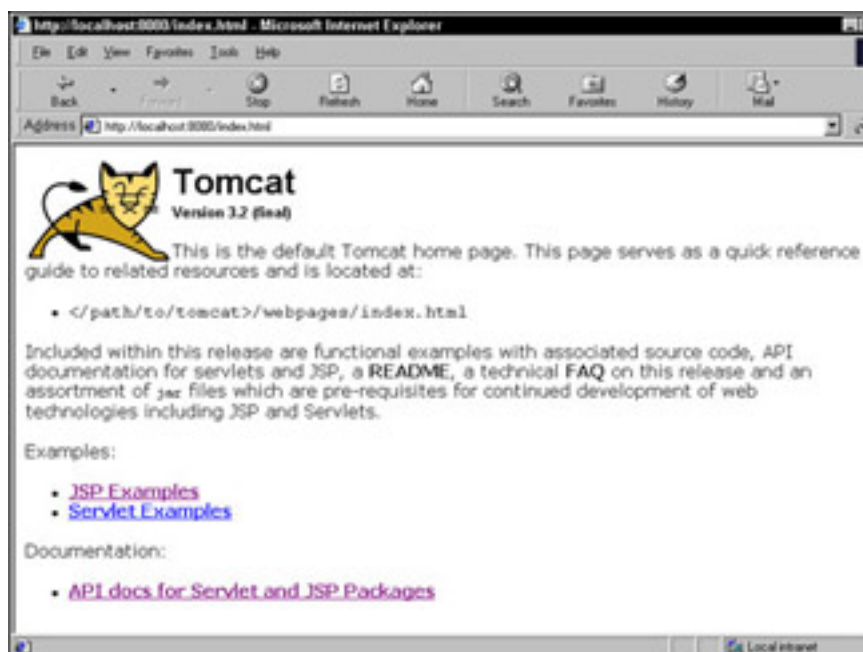
Figure C-5: The Tomcat greeting screen

If you see this screen, congratulations! You have a working version of the Tomcat Web server installed on your system.

Try out the JSP and servlet examples next. Figure C-6 shows the page of JSP examples.
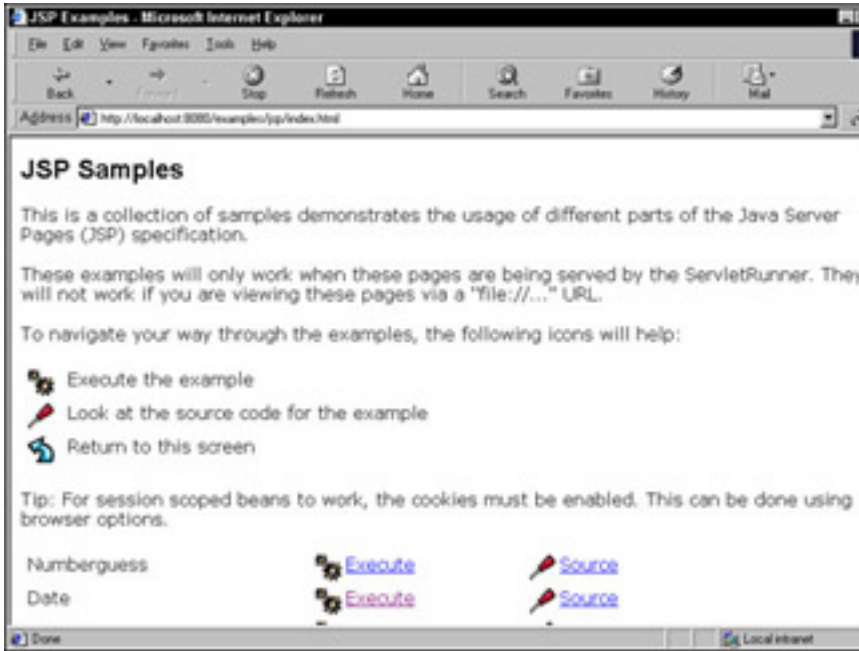
Figure C-6: The Tomcat JSP example page

The page in Figure C-6 contains 15 JSP examples, of which two are shown.

You have not seen a JSP execute yet. Click on the link labeled "Execute" next to the Date example. If you see a screen such as Figure C-7, you have a working, JSP-enabled Web server!
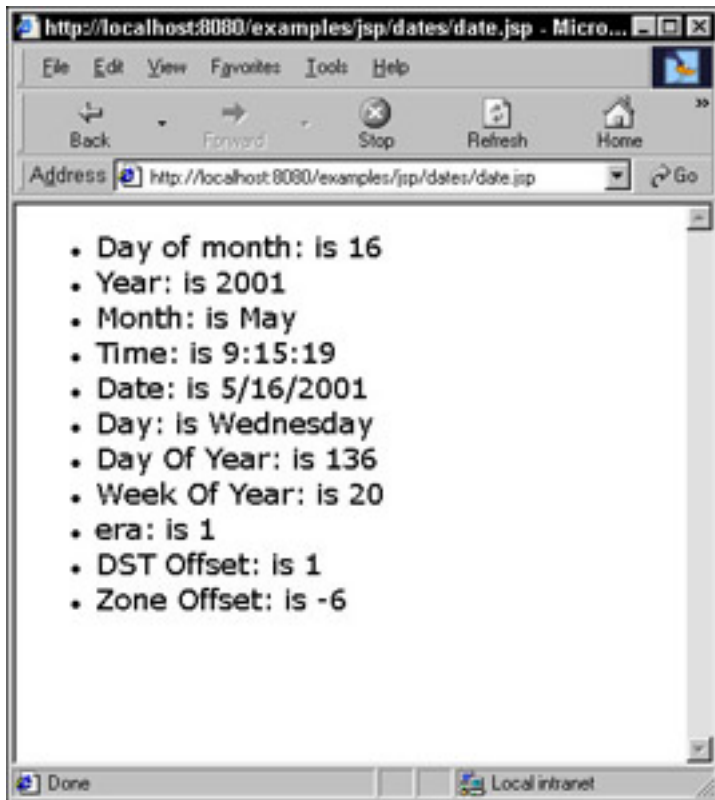
Figure C-7: The Date JSP executing in Tomcat

Tomcat Directories and JSP Processing

The documentation contained in `TOMCAT_HOME\doc` covers nearly everything you need to know about using Tomcat. In this section, you can see a quick rundown on placement of JSP files in Tomcat.

You have two options for running JSPs in Tomcat: You can create your own *Web application* or use the *example directory* to run your JSP pages. This section describes how to use the example directory. Consult the documentation for configuring Tomcat for your own Web application.

You have to place your JSP and HTML files in one directory and your class files from JavaBeans in another directory. First, let's look at the directories and read about where you'll place JSP and HTML files.

To run a set of related JSPs, you need to create a directory in the `TOMCAT_HOME\Webapps` directory. For example, let's create the directory `TOMCAT_HOME\Webapps\examples\jsp`.

Caution   Do not be misled by looking at the URLs in the browser. Notice that the URL shown in Figure C-7, `http://localhost:8080/examples/jsp/datesdate.jsp`, does not include the `Webapps` directory. When placing your JSPs, remember that the directory shown in the URL is really found in `TOMCAT_HOME\Webapps`.

Put your JSP files and static HTML files in your directory. For example, the series of JSPs for the Hotel Reservation System are stored in the directory `TOMCAT_HOME\Webapps\examples\jsp\hotelres`.

Caution   Do not invoke your JSPs or static HTML files by clicking their icons. You *must* invoke JSPs or HTML pages from the browser from the `http://localhost:8080` address. If you click HTML page icons, you see the pages in the browser (of course), but you cannot invoke any JSPs from the pages.

As for your class files representing your JavaBeans, Tomcat understands that the directory `TOMCAT_HOME\Webapps\examples\classes` holds class files. Because you are a sharp Java programmer, you've

already coded your bean classes within a package, which corresponds to a directory within the classes directory cited above. For example, beans created for the Hotel Reservation System are stored in `TOMCAT_HOME\Webapps\examples\classes\hotelres`.

Once again, store your JSPs and HTML pages together in a directory located in the JSP subdirectory and your bean classes in a directory (package) located in the `classes` subdirectory.

More Information on Tomcat

As previously mentioned, mailing lists are dedicated to disseminating information about Tomcat. If you have a question, you can post it to the Web site for this book, of course, or join one of the Tomcat mailing lists.

Tip   To join the Tomcat mailing list, visit http://jakarta.apache.org/tomcat and follow the directions found there.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Appendix B: The EJB API

This appendix lists the classes and interfaces that comprise Sun Microsystems' Enterprise JavaBeans API for quick
reference.

The EJB API

Here, you can read about the EJB 1.1 API and Sun's proposed changes for the upcoming EJB 2.0 release.

The entire EJB API is contained within the following two packages:

- javax.ejb

- `javax.ejb.spi` (2.0 only)

Before examining the contents of these two packages, let's take a quick look at the class and interface hierarchies for
the `javax.ejb` package.

Class and Inheritance Hierarchies for Package javax.ejb

The proposed classes and interfaces introduced with release 2.0 are noted in the following list. Notice that EJB works
with interfaces; the classes in the javax.ejb package are representations of exceptions.

```
class java.lang.Object
      class java.lang.Throwable (implements java.io.Serializable)
            class java.lang.Exception
            class javax.ejb.CreateException
                  class javax.ejb.DuplicateKeyException
      class javax.ejb.FinderException
            class javax.ejb.ObjectNotFoundException
      class javax.ejb.RemoveException
      class java.lang.RuntimeException
            class javax.ejb.EJBException
                  class javax.ejb.AccessLocalException
                  class javax.ejb.NoSuchEntityException
                  class javax.ejb.NoSuchObjectLocalException (2.0)
                  class javax.ejb.TransactionRequiredLocalException (2.0)
                  class javax.ejb.TransactionRolledbackLocalException (2.0)
```

Interface Hierarchy

Here is the interface hierarchy:

```
interface javax.ejb.EJBContext
      interface javax.ejb.EntityContext
      interface javax.ejb.MessageDrivenContext (2.0)
      interface javax.ejb.SessionContext
      interface javax.ejb.EJBLocalHome (2.0)
interface javax.ejb.EJBLocalObject (2.0)
interface javax.ejb.EJBMetaData
interface java.rmi.Remote
      interface javax.ejb.EJBHome
      interface javax.ejb.EJBObject
interface java.io.Serializable
      interface javax.ejb.EnterpriseBean
            interface javax.ejb.EntityBean
            interface javax.ejb.MessageDrivenBean (2.0)
            interface javax.ejb.SessionBean
      interface javax.ejb.Handle
      interface javax.ejb.HomeHandle
interface javax.ejb.SessionSynchronization
```

Because the javax.ejb.spi package (new with release 2.0, remember?) contains a single interface, let's go ahead and discuss it.

The javax.ejb.spi Package and Its Interface

The `javax.ejb.spi` package contains a single interface called `HandleDelegate`, which is implemented by the EJB container. This interface is used by portable implementations of `javax.ejb.Handle` and `javax.ejb.HomeHandle`. `HandleDelegate` is not used by EJB components or by client components. It provides methods to serialize and deserialize `EJBObject` and `EJBHome` references to streams. Table B-1 lists the methods available in the `HandleDelegate` interface.

Table B-1: Methods of the HandleDelegate Interface

Method Signature	Description
`EJBHome readEJBHome( ObjectInputStream istream)`	Invoked by the EJB container to deserialize the `EJBHome` reference corresponding to a `HomeHandle`.
`EJBObject readEJBObject( ObjectInputStream istream)`	Invoked by the EJB container to deserialize the `EJBObject` reference corresponding to a `Handle`.
`void writeEJBHome( EJBHome home, ObjectOutputStream ostream )`	Serializes the `EJBHome` reference corresponding to a `HomeHandle`.
`void writeEJBObject( EJBObject object, ObjectOutputStream ostream )`	Serializes the `EJBObject` corresponding to a `Handle`.

The vast bulk of the EJB API is contained in the `javax.ejb` package. Let's take a look at `javax.ejb` now.

The javax.ejb Package

Enterprise JavaBeans work with remote interfaces. Hence, the majority of the methods defined in the EJB spec are

contained within interfaces. EJB classes are limited to exception classes. Later in this appendix, you can examine a list of all available methods from the interfaces contained in `javax.ejb`. For now, let's look at the exception classes, which are listed in . These exception classes inherit the "usual" methods, such as `printStackTrace`, and use the "usual" constructors.

Table B-2: Exception Classes of the javax.ejb Package

Class Name	Description
AccessLocalException	An `AccessLocalException` is thrown to indicate that the caller does not have permission to call the method.
CreateException	The `CreateException` exception must be included in the throws clauses of all create methods defined in an enterprise bean's `home` interface.
DuplicateKeyException	The `DuplicateKeyException` exception is thrown if an entity EJB object cannot be created because an object with the same key already exists.
EJBException	The `EJBException` exception is thrown by an enterprise bean instance to its container to report that the invoked business method or callback method could not be completed because of an unexpected error (for example, the instance failed to open a database connection).
FinderException	The `FinderException` exception must be included in the throws clause of every `findMETHOD()` method of an entity bean's `home` interface.
NoSuchEntityException	The `NoSuchEntityException` exception is thrown by an entity bean instance to its container to report that the invoked business method or callback method could not be completed because the underlying entity was removed from the database.
NoSuchObjectLocalException	A `NoSuchObjectLocalException` is thrown if an attempt is made to invoke a method on an object that no longer exists (new with release 2.0).
ObjectNotFoundException	The `ObjectNotFoundException` exception is thrown by a `finder` method to indicate that the specified EJB object does not exist.
RemoveException	The `RemoveException` exception is thrown at an attempt to remove an EJB object when the enterprise bean or the container does not enable the EJB object to be removed.
TransactionRequiredLocalException	This exception indicates that a request carried a null transaction context, but the target object requires an activate transaction (new with release 2.0).
TransactionRollbackLocalException	This exception indicates that the transaction associated with processing of the request has been rolled back, or marked to roll back (new with release 2.0).

As you see, the new exception classes in release 2.0 deal with exceptions thrown by local objects.

The `javax.ejb` package defines no other classes than the exception classes listed in [Table B-2](#); the remainder of the package consists of interfaces. [Table B-3](#) lists the interfaces available in the `javax.ejb` package. The interfaces newly available with the release 2.0 are noted.

Table B-3: Interfaces of the javax.ejb Package

Interface Name	Description
EJBContext	The `EJBContext` interface provides an instance with access to the container-provided runtime context of an enterprise bean instance.
EJBHome	The `EJBHome` interface must be extended by all enterprise bean's remote home interfaces.
EJBLocalHome	The `EJBLocalHome` interface must be extended by all enterprise bean's local home interfaces (new with release 2.0).
EJBLocalObject	The `EJBLocalObject` interface must be extended by all enterprise bean's local interfaces.
EJBMetaData	The `EJBMetaData` interface enables a client to obtain the enterprise bean's metadata information.
EJBObject	The `EJBObject` interface is extended by all enterprise bean's remote interfaces.
EnterpriseBean	The `EnterpriseBean` interface must be implemented by every enterprise bean class.
EntityBean	The `EntityBean` interface is implemented by every entity enterprise bean class.
EntityContext	The `EntityContext` interface provides an instance with access to the container-provided runtime context of an entity enterprise bean instance.
Handle	The `Handle` interface is implemented by all EJB object handles.
HomeHandle	The `HomeHandle` interface is implemented by all home object handles.
MessageDrivenBean	The `MessageDrivenBean` interface is implemented by every message-driven enterprise bean class (new with release 2.0).
MessageDrivenContext	The `MessageDrivenContext` interface provides access to the runtime message-driven context that the container provides for a message-driven enterprise bean instance (new with release 2.0).
SessionBean	The `SessionBean` interface is implemented by every session enterprise bean class.
SessionContext	The `SessionContext` interface provides access to the runtime session context that the container provides for a session enterprise bean instance.
SessionSynchronization	The `SessionSynchronization` interface enables a session bean instance to be notified by its container of transaction boundaries.

Note that all of the new interfaces available with release 2.0 deal with the `MessageDrivenBean` and local objects. The rest of the interfaces in the `javax.ejb` package existed in the previous EJB specification release. All of these interfaces are discussed in this appendix.

The EJBContext Interface

The `EJBContext` interface provides a bean with the context of the EJB container. As such, the methods available through `EJBContext` enable a bean to glean information about the container and the beans contained within. Table B-4 lists the methods from the `EJBContext` interface.

Table B-4: Methods of the EJBContext Interface

Method Signature	Description
`Principal getCallerPrincipal()`	Returns the security Principal that identifies the method caller.
`EJBHome getEJBHome()`	Returns the bean's (remote) home interface.
`EJBLocalHome getEJBLocalHome()`	Returns the bean's local home interface (new with release 2.0).
`boolean getRollbackOnly()`	Determines if the transaction is marked for rollback.
`UserTransaction getUserTransaction()`	Returns a reference to the current transaction demarcation interface.
`boolean isCallerInRole()`	Determines if the method caller has a given security role.
`void setRollbackOnly()`	Sets the current transaction for rollback.

The `EJBContext` interface is extended by the `SessionContext` and `EntityContext` interfaces, which contain methods peculiar to the two bean types.

The EJBHome Interface

The `EJBHome` interface must be extended by all remote home interfaces. With release 2.0, EJB draws a distinction between a *remote* home interface and a *local* home interface. Later in this appendix you'll read about the methods in the `EJBLocalHome` interface. All methods contained in the `EJBHome` interface throw (minimally) a `RemoteException`. The proposed 2.0 release does not add new methods, or deprecate existing ones. Table B-5 lists the methods in the `EJBHome` interface.

Table B-5: Methods of the EJBHome Interface

Method Signature	Description
`EJBMetaData getEJBMetaData()`	Returns the metadata for the bean. The bean's metadata is rarely used by application developers.
`HomeHandle getHomeHandle()`	Returns a handle for the (remote) home object.
`void remove( Handle handle )`	Removes an EJB object referenced by its handle.
`void remove( Object primaryKey)`	Removes an EJB object referenced by its primary key.

`getEJBHome()`	Returns the bean's (remote) home interface.

The EJBLocalHome Interface

The `EJBLocalHome` interface is conceptually similar to the `EJBHome` interface except that the `EJBLocalHome` interface should be extended for all *local* clients of enterprise beans. This interface, new with release 2.0, contains one method, `remove`, which has the following signature:

```
void remove( Object primaryKey )
      throws RemoveException, EJBException ;
```

The `remove` method can be called *only* by local clients of an entity bean. Recall that session beans do not have methods that rely on the existence of a primary key (such as finder methods). The implementation of this interface is the responsibility of the EJB container.

The EJBLocalObject Interface

The `EJBLocalObject` interface, new with release 2.0, serves the same function as the `EJBObject` interface, but for local clients. An enterprise bean's local interface provides the local client view of an EJB object. An enterprise bean's local interface defines the business methods callable by local clients. The implementation of this interface is the responsibility of the EJB container. Table B-6 lists the methods available from a class that implements the `EJBLocalObject` interface.

Table B-6: Methods of the EJBLocalHome Interface

Method Signature	Description
`EJBLocalHome getEJBLocalHome()`	Returns the bean's local home interface.
`Object getPrimaryKey()`	Returns the primary key for the EJB local object.
`boolean isIdentical( EJBLocalObject lobj )`	Determines if the given EJB local object is identical to the invoking EJB local object.
`void remove()`	Removes the EJB local object.

The EJBMetaData Interface

The `EJBMetaData` interface enables a client to obtain the enterprise bean's metadata information. The metadata is intended for development tools used for building applications that use deployed enterprise beans, and for clients using a scripting language to access the enterprise bean. Table B-7 shows the methods available by a class that implements the `EJBMetaData` interface.

Table B-7: Methods of the EJBMetaData Interface

Method Signature	Description
`EJBHome getEJBHome()`	Returns the bean's remote home interface.
`Class getHomeInterfaceClass()`	Returns the class for the enterprise bean's remote home interface.
`Class getRemoteInterfaceClass()`	Returns the class for the enterprise bean's remote interface.

`boolean isSession()`	Determines if the bean's type is session as opposed to entity, or (with release 2.0 only) message driven.
`boolean isStatelessSession()`	Determines if the bean's type is "stateless session."
`object getPrimaryKey()`	Returns the primary key for the EJB local object.

The EJBObject Interface

The `EJBObject` interface is extended by all enterprise bean's remote interfaces. An enterprise bean's remote interface provides the remote client view of an EJB object. An enterprise bean's remote interface defines the business methods callable by a remote client. Table B-8 shows the methods available by a class that implements the `EJBObject` interface.

Table B-8: Methods of the EJBObject Interface

Method Signature	Description
`EJBHome getEJBHome ()`	Returns the bean's remote home interface.
`Handle getHandle()`	Returns a handle for the invoking EJB object.
`Object getPrimaryKey()`	Returns the primary key of the EJB object.
`boolean isIdentical( EJBObject eobj )`	Determines if a given EJB object is identical to the invoking EJB object.
`void remove()`	Removes the EJB object.

The EnterpriseBean Interface

The `EnterpriseBean` interface contains no method signatures or constants. Interface `EnterpriseBean` serves as the superinterface for the `EntityBean`, `SessionBean`, and `MessageDrivenBean` interfaces.

The EntityBean Interface

The `EntityBean` interface must be extended by any class that implements an entity EJB. The container uses the methods defined in the entity bean class to notify the bean instances of various events in the bean's life cycle. Table B-9 lists the methods available in an entity bean class that implements the `EntityBean` interface.

Table B-9: Methods of the EntityBean Interface

Method Signature	Description
`void ejbActivate()`	The container invokes `ejbActivate` when the bean instance is loaded from the bean pool to become associated with a particular EJB object.
`void ejbLoad()`	The container invokes `ejbLoad` to synchronize the entity bean's state by refreshing the bean with data from the underlying database.

`void ejbPassivate()`	The container invokes `ejbPassivate` before the bean instance becomes disassociated with a specific EJB object, possibly by placing the bean instance into the instance pool.
`void ejbRemove()`	The container invokes `ejbRemove` before it removes the EJB object associated with the bean instance.
`void ejbStore()`	The container invokes `ejbStore` to synchronize the bean's state by storing the bean data to the underlying database.
`void setEntityContext()`	Sets the entity context for the newly created entity bean.
`void unsetEntityContext()`	Unsets the entity context immediately before removing the bean instance.

The EntityContext Interface

The `EntityContext` interface, a subinterface of `EJBContext`, provides an instance with access to the container-provided runtime context of an entity enterprise bean instance. The container passes the `EntityContext` interface to an entity enterprise bean instance after the instance has been created. The `EntityContext` of an entity bean stays associated with the bean for the bean's entire life. Table B-10 lists the methods available from the `EntityBean` interface.

Table B-10: Methods of the EntityContext Interface

Method Signature	Description
`EJBLocalObject getEJBLocalObject()`	Returns a reference to the EJB local object currently associated with the bean's instance (new with release 2.0).
`EJBObject getEJBObject()`	Returns a reference to the EJB object currently associated with the bean instance.
`Object getPrimaryKey()`	Returns the primary key of the EJB object currently associated with this bean instance.
`void ejbLoad()`	The container invokes `ejbLoad` to synchronize the entity bean's state by refreshing the bean with data from the underlying database.

The Handle Interface

The `Handle` interface is implemented by all EJB object handles. A handle is an abstraction of a network reference to an EJB object. A handle is used as a "robust" persistent reference to an EJB object.

The `Handle` interface defines one method, `getEJBObject`, with the following signature:

```
EJBObject getEJBObject( ) throws RemoteException ;
```

The HomeHandle Interface

The `HomeHandle` interface is implemented by all home object handles. A handle is an abstraction of a network

reference to a home object. A handle is used as a "robust" persistent reference to a `homeobject`.

The `HomeHandle` interface defines one method, `getEJBHome`, with the following signature:

```
EJBHomeObject getEJBHome ( ) throws RemoteException ;
```

The MessageDrivenBean Interface

The `MessageDrivenBean` interface is implemented by every message-driven enterprise bean class. The container uses the `MessageDrivenBean` methods to notify the enterprise bean instances of the instance's life cycle events.

The `MessageDrivenBean` interface, as well as any interface dealing with message-driven beans, is new with release 2.0. Table B-11 lists the methods available from the `MessageDrivenBean` interface.

Table B-11: Methods of the MessageDrivenBean Interface

Method Signature	Description
`void ejbRemove()`	The container invokes `remove` immediately prior to ending the life of the message bean (new with release 2.0).
`void setMessageDrivenContext( MessageDrivenContext mctx )`	Sets the message-driven context immediately after creating the instance of the message bean.

The MessageDrivenContext Interface

The `MessageDrivenContext` interface, new with release 2.0, is a subinterface of `EJBContext` that provides an instance with access to the container-provided runtime context of a message bean instance. The container passes the `MessageDrivenContext` interface to a message enterprise bean instance after the instance has been created. The `MessageDrivenContext` of a message bean stays associated with the bean for the bean's entire life.

The `MessageDrivenContext` interface requires no methods other than those from the superinterface, `EJBContext`, be implemented.

The SessionBean Interface

The `SessionBean` interface is implemented by every session enterprise bean class. The container uses the `SessionBean` methods to notify the enterprise bean instances of the instance's life cycle events. Table B-12 lists the methods available from the `SessionBean` interface.

Table B-12: Methods of the SessionBean Interface

Method Signature	Description
`void ejbActivate()`	The container invokes `ejbActivate` when the session bean instance becomes associated with an EJB object by fetching from a bean pool or deserializing from storage (enters the active state).
`void ejbPassivate()`	The container invokes `ejbPassivate` when the session bean is about to go into a bean pool or be persisted (enters the passive state).

`void ejbRemove()`	The container invokes `remove` immediately prior to ending the life of the session bean.
`void setMessageDrivenContext(` `MessageDrivenContext mctx )`	Sets the associated session context after the container creates an instance of the session bean.

The SessionContext Interface

The `SessionContext` interface, a subinterface of `EJBContext`, provides an instance with access to the container-provided runtime context of a session enterprise bean instance. The container passes the `SessionContext` interface to a session enterprise bean instance after the instance has been created. The `SessionContext` of a session bean stays associated with the bean for the bean's entire life. Table B-13 lists the methods available from the `SessionBean` interface.

Table B-13: Methods of the SessionContext Interface

Method Signature	Description
`EJBLocalObject getEJBLocalObject()`	Returns a reference to the EJB local object currently associated with the bean's instance (new with release 2.0).
`EJBObject getEJBObject()`	Returns a reference to the EJB object currently associated with the bean instance.

The SessionSynchronization Interface

The `SessionSynchronization` interface enables a session bean instance to be notified by its container of transaction boundaries. A session bean class is not required to implement this interface unless it wishes to synchronize its state with the transactions. Table B-14 lists the methods available from the `SessionSynchronization` interface.

Table B-14: Methods of the SessionSynchronization Interface

Method Signature	Description
`void afterBegin()`	The container invokes `afterBegin` to notify a session bean that a new transaction has started.
`void afterCompletion ( boolean` `committed )`	The container invokes `afterCompletion` to notify a session bean that a commit or a rollback occurred.
`void beforeCompletion()`	The container invokes `beforeCompletion` to notify a session bean that a transaction is about to be committed.

Interfaces and Classes Added to JSP Release 1.2

The up-and-coming JSP release 1.2 includes some additional interfaces and classes to the `javax.servlet.jsp.tagext` package. Table A-25 provides a quick look at what's new with JSP 1.2

Table A-25: New Components of the javax.servlet.jsp.tagext Package (JSP 1.2)

Component	Type	Description
IterationTag	Interface	Extends the `Tag` interface by providing one additional method that controls the reevaluation of its body.
TryCatchFinally	Interface	Another interface of a tag-handler class that wants more hooks for managing resources.
PageData	Abstract Class	Objects of class `PageData` are generated by the JSP translator and passed to a Tag library validator.
TagLibraryValidator	Abstract Class	Translation-time validator class for a JSP page.
TagVariableInfo	Class	Contains information about tag variables coded in the tld.

The IterationTag Interface

The `IterationTag` interface extends `Tag` by defining one additional method that controls the reevaluation of its body.

A tag handler that implements `IterationTag` is treated as one that implements `Tag` regarding the `doStartTag` and `doEndTag` methods. `IterationTag` provides a new method: `doAfterBody` with the following signature:

```
int doAfterBody()
```

The `IterationTag` interface defines an additional constant that requests the evaluation of the tag body, as follows:

```
public static final int EVAL_BODY_AGAIN
```

Note that `EVAL_BODY_AGAIN` replaces the deprecated tag `BodyTag.EVAL_BODY_TAG`.

The TryCatchFinally Interface

The `TryCatchFinally` interface is the auxiliary interface of a `Tag`, `IterationTag`, or `BodyTag` tag handler that wants additional hooks for managing resources.

This interface provides two new methods: `doCatch(Throwable)` and `doFinally`, with the following signatures:

```
void doCatch( Throwable t )
void doFinally()
```

The `doCatch` method is invoked whenever an exception is thrown within the body of a tag. The `doFinally` method is invoked in all cases after the `doEndTag` method for classes implementing the `Tag`, `BodyTag`, or the `IterationTag` interfaces.

The PageData Abstract Class

The `PageData` class contains one method (Table A-26), `getInputStream`, that returns an input stream of an XML document representing the translated JSP page. You, the JSP author, do not code the `getInputStream` method.

Table A-26: Methods in the PageData Class

Method Signature	Description
`abstract InputStream getInputStream()`	Returns an input stream of an XML document representing the translated JSP page.

The TagLibraryValidator Abstract Class

This class is the translation-time validator class for a JSP page. A validator operates on the XML document associated with the JSP page. The tld file associates a `TagLibraryValidator` class and some `init` arguments with a tag library. Table A-27 lists the methods available from the `TagLibraryValidator` class.

Table A-27: Methods in the TagLibraryValidator Class

Method Signature	Description
`Map getInitParameters()`	Returns the `init` parameters data from the tld as an immutable map. Parameter names are keys, and parameter values are the values.
`void release()`	Releases any data kept by this instance of the tag-handler class.
`void setInitParameters( Map parmMap )`	Sets the `init` parameters as key/value pairs for this instance of the validator.
`String validate( String tagPrefix, String tagURI, PageData page )`	Validates a JSP page. Returns a null string if validation is successful or a diagnostic if not.

The TagVariableInfo Abstract Class

This class contains variable information for a tag in a tag library. It is instantiated from the Tag Library Descriptor file (tld) and is available only at translation time. The methods contained in the `TagVariableInfo` class are similar to those contained in the `VariableInfo` class. Table A-28 lists the methods available from the `TagVariableInfo` class.

Table A-28: Methods in the TagVariableInfo Class

Method Signature	Description
`String getClassName ()`	Returns the value (body) of the `variable-class` element coded in the tld.
`boolean getDeclare()`	Returns the value (body) of the `declare` element coded in the tld.
`String getNameFromAttribute()`	Returns the value (body) of the `name-from-attribute` element coded in the tld.
`int getScope()`	Returns the value (body) of the `scope` element coded in the tld.

end

.

Appendix A: The JSP API

This appendix lists the classes and interfaces that comprise Sun Microsystems' JavaServer Pages API for quick reference.

The JSP API

Here, you can read about the JSP 1.1 API and Sun Microsystems' proposed changes for the upcoming JSP 1.2 release. Only a few differences exist between the two.

Because JSPs eventually get translated into Java servlets, you can read references to Sun's Java Servlet API here. Also, the JSP classes and interfaces inherit from the servlet classes, making a discussion of the JSP API tightly interwoven with mentioning the Servlet API.

One more point worth mentioning is that you, the JSP programmer, often will not invoke the JSP API methods directly. When you create a JSP page, the JSP translator generates references to the JSP API within the generated servlet. However, when you create custom tags, you must code references to the JSP API for your classes that implement your tag's functionality.

All of the JSP API is contained within the following two packages:

- javax.servlet.jsp

- javax.servlet.jsp.tagext

Before examining the contents of these two packages, take a quick look at the class and interface hierarchies for the JSP API for JSP, releases 1.1 and 1.2.

The Class and Interface Hierarchies for the javax.servlet.jsp Package

The following are the proposed classes and interfaces for the JSP 1.2 release.

```
class java.lang.Object

        class javax.servlet.jsp.JspEngineInfo
      class javax.servlet.jsp.JspFactory
      class javax.servlet.jsp.PageContext
      class java.lang.Throwable (implements java.io.Serializable)
            class java.lang.Exception
                  class javax.servlet.jsp.JspException
      class javax.servlet.jsp.JspTagException (JSP 1.2)
            class java.io.Writer
class javax.servlet.jsp.JspWriter
      interface javax.servlet.Servlet
      interface javax.servlet.jsp.JspPage
      interface javax.servlet.jsp.HttpJspPage
```

The Class and Inheritance Hierarchies for the javax.servlet.jsp.tagext Package

The following are the proposed classes and interfaces for the JSP 1.2 release.

```
class java.lang.Object
        class javax.servlet.jsp.tagext.PageData (1.2)
        class javax.servlet.jsp.tagext.TagAttributeInfo
        class javax.servlet.jsp.tagext.TagData (implements java.lang.Cloneable)
        class javax.servlet.jsp.tagext.TagExtraInfo
        class javax.servlet.jsp.tagext.TagInfo
        class javax.servlet.jsp.tagext.TagLibraryInfo
        class javax.servlet.jsp.tagext.TagLibraryValidator (1.2)
        class javax.servlet.jsp.tagext.TagSupport (implements javax.servlet.jsp.tagext.IterationTag,
java.io.Serializable)
        class javax.servlet.jsp.tagext.BodyTagSupport (implements javax.servlet.jsp.tagext.BodyTag)
        class javax.servlet.jsp.tagext.TagVariableInfo (1.2)
        class javax.servlet.jsp.tagext.VariableInfo
           class java.io.Writer
                  class javax.servlet.jsp.JspWriter
        class javax.servlet.jsp.tagext.BodyContent interface javax.servlet.jsp.tagext.Tag
        interface javax.servlet.jsp.tagext.IterationTag (1.2)
           interface javax.servlet.jsp.tagext.BodyTag
interface javax.servlet.jsp.tagext.TryCatchFinally (1.2)
```

The javax.servlet.jsp Package

The `javax.servlet.jsp` package contains the classes and interfaces that describe and define the contracts between a JSP page implementation class and the runtime environment provided for an instance of such a class by a JSP container. Table A-1 lists the interfaces, classes, and exceptions of the `javax.servlet.jsp` package.

Table A-1: Components of the javax.servlet.jsp Package

Component	Type	Description
JspPage	Interface	Describes requirements for a JSP-generated servlet class.
HttpJspPage	Interface	Describes requirements for a JSP — uses the HTTP protocol.
JspEngineInfo	Abstract Class	Provides information about the JSP engine in use.
JspFactory	Abstract Class	Defines factory methods that the JSP page may use to create needed runtime objects.
JspWriter	Abstract Class	Enables the creation of a buffered version of `java.io.PrintWriter` that throws IO exceptions.
PageContext	Abstract Class	`PageContext` objects enable the JSP page to access page-specific attributes, including a `JspWriter` and `errorpage` exception processing.
JspError	Exception	When thrown, output generation stops and processing is directed to error pages.
JspException	Exception	Generic exception known to the JSP engine.

Note that the JSP author does not invoke the code that invokes most of the methods in the interfaces and classes listed in .

The JspPage Interface

`JspPage` defines methods that create and destroy a generated instance of the JSP page. The JSP container automatically invokes the methods when appropriate. However, the JSP specification enables the JSP author to invoke the methods as well. shows the methods defined in `JspPage`.

Table A-2: Methods of the JspPage Interface

Method Signature	Description
`void jsp_init()`	Invoked by the JSP container when the JSP page is initialized.
`void jsp_destroy()`	Invoked by the JSP container just before the container destroys the JSP page.

The JSP spec enables the JSP author to invoke these methods but the runtime invokes them when needed.

The HttpJspPage Interface

The `HttpJspPage` interface extends `JspPage` and provides an additional method, shown in .

Table A-3: The Method of the HttpJspPage Interface

Method Signature	Description
`void _jspservice()`	`jspService` corresponds to the body of the JSP page. This method is defined automatically by the JSP processor and should never be invoked by the JSP author.

The Abstract Class JspEngineInfo

`JspEngineInfo` contains a single method that returns the version number of the JSP engine in use, shown in .

Table A-4: The Method of the JspEngineInfo Class

Method Signature	Description
`abstract String getSpecificationVersion()`	Returns a version number for the JSP engine in use.

The Abstract Class JspFactory

The `JspFactory` is an abstract class that defines a number of factory methods available to a JSP page at runtime for the purposes of creating instances of various interfaces and classes used to support the JSP implementation. The JSP author does not invoke the methods in this class, which are shown in .

Table A-5: Methods of the JspFactory Class

Method Signature	Description

`static JspFactory getDefaultFactory()`	Returns the default factory used in this JSP container.
`abstract JspEngineInfo getEngineInfo()`	Gets implementation-specific data on the current JSP engine.
`abstract PageContext getPageContext( Servlet serv, ServletRequest request, ServletResponse response, String errorPageURL, boolean needsSession, int buffer, boolean autoFlush)`	Gets an instance of the `PageContext` abstract class for the servlet generated by the JSP engine.
`abstract void releasePageContext( PageContext thisPage )`	Releases a previously allocated `PageContext` object.
`static void setDefaultFactory( JspFactory defFactory )`	Sets the default JSP factory used by this engine.

The Abstract Class JspWriter

Instances of `JspWriter` represent the output stream to the client. Many of the methods available in `JspWriter` mimic those in `java.io.PrintWriter`. The methods of `JspWriter` are listed in Table A-6.

Table A-6: Methods of the JspWriter Class

Method Signature	Description
`abstract void clear()`	Clears the output stream's buffer. Throws an `IOException` if buffer is already clear.
`abstract void clearBuffer()`	Clears the output stream's buffer. Does not throw an `IOException` if buffer is already clear.
`abstract void close()`	Closes and flushes the output stream.
`abstract void flush()`	Flushes the output stream.
`int getBufferSize()`	Returns the size of the output buffer.
`abstract int getRemaining()`	Returns the number of unused bytes in the output buffer.
`boolean isAutoFlush()`	Returns true if buffer automatically flushes; false if IO exceptions are thrown on buffer overflows.
`abstract void newLine()`	Writes a line separator (`line.separator` in system properties) which does not need to be a `newline` character.
`abstract void print( argType printIt )`	Prints a variety of primitive and reference types. Parameters to `print` are the same as those to `java.io.print`.
`abstract void println( argType printIt )`	Prints a variety of primitive and reference types followed by a `newline` character. Parameters to `println` are the same as those to `java.io.print`.

The fields available from class `JspWriter` are listed in Table A-7.

Table A-7: Fields Declared in Class JspWriter

Declaration	Description
`protected boolean autoFlush`	Whether or not the output buffer flushes automatically when full (true) or throws an `IOException` when full (false).

`protected int bufferSize`	Buffer size in use.
`static int DEFAULT_BUFFER`	Variable indicating that the output stream is buffered and using the default buffer size.
`static int NO_BUFFER`	Variable indicating that the output stream is not buffered.
`static int UNBOUNDED_BUFFER`	Variable indicating output stream is unbounded.

The Abstract Class PageContext

The `PageContext` class provides methods to the JSP author that enable the following functions:

- Managing the various scoped namespaces (`page scope`, `session scope`, and so on)

- Accessing the various public objects (`out`, `request`, `response`, and so on)

- Fetching the `JspWriter` for output

- Managing session usage by the page

- Exposing page directive attributes to the scripting environment

- Forwarding or including the current request to other active components in the application

- Handling `errorpage` exception processing

Table A-8 lists the methods available in class `PageContext`.

Table A-8: Methods in Class PageContext

Method Signature	Description
`abstract Object findAttribute( String attrName )`	Searches for the attribute named `attrName` in the order of `page`, `request`, `session`, and `application` scopes, and returns if found or returns null if no attribute with `attrName` exists.
`abstract void forward( String resourceURL )`	Redirects the current response or request to another component (usually a servlet or another JSP page).
`abstract Object getAttribute( String attrName )`	Returns the attribute named `attrName` in `page` scope only or returns null if no such attribute exists.
`abstract Object getAttribute( String attrName, int scope )`	Returns the attribute named `attrName` in the specified scope or returns null if no such attribute exists.
`abstract Enumeration getAttributeNamesInScope( int scope)`	Returns the names of the attributes in the specified scope.
`abstract int getAttributeScope( String attrName )`	Returns the scope by which the attribute `attrName` is known, or 0 if no attribute exists.
`abstract Exception getException()`	Returns the last exception thrown.
`abstract JspWriter getOut()`	Returns the current instance of `JspWriter` used to hold servlet-generated output to the client.
`abstract Object getPage()`	Returns the servlet instance associated with the current `PageContext`.
`abstract ServletRequest getRequest()`	Returns the request object associated with the current `PageContext`.

`abstract ServletResponse getResponse()`	Returns the response object associated with the current `PageContext`.
`abstract ServletConfig getServletConfig()`	Returns the `ServletConfig` object associated with the current `PageContext`.
`abstract ServletContext getServletContext()`	Returns the `ServletContext` object associated with the current `PageContext`.
`abstract HttpSession getSession ()`	Returns the session object associated with the current `PageContext`.
`abstract void handlePageException( Exception pExc )`	Processes an unhandled page level exception, perhaps by redirecting to a JSP error page or taking application-specific action.
`abstract void include( String resourceURL )`	Causes `resourceURL` to be processed as part of the current request or response.
`abstract void initialize( Servlet aServlet, ServletRequest request, ServletResponse response, String errorPageURL, boolean requiresSession, int bufSize, boolean autoFlush )`	Initializes a `PageContext`, usually in response to a `JspFactory.getPageContext` method.
`BodyContent pushBody()`	Returns an instance of `BodyContent` and saves the current instance of `JspWriter` (the implicit "out" object).
`JspWriter popBody()`	Returns the saved version of `JspWriter` by a previous call to `pushBody`.
`abstract void removeAttribute( String attrName )`	Removes the attribute named `attrName` within the `page` scope.
`abstract void removeAttribute( String attrName, int scope )`	Removes the attribute named `attrName` within the specified scope.
`abstract void setAttribute( String attrName, Object attrValue )`	Sets a `page` scope attribute named `attrName` with the value `attrValue`.
`abstract void setAttribute( String attrName, Object attrValue, int scope )`	Sets an attribute named `attrName` with the value `attrValue` within the specified scope.

Table A-9 lists the class variables (all declared *static*) available in class `PageContext`.

Table A-9: Class Variables Declared in Class PageContext

Declaration	Description
`String APPLICATION`	The name of the `ServletContext` in the `PageContext` name table.
`int APPLICATION_SCOPE`	Value representing `application` scope.
`String CONFIG`	Name used to store `ServletConfig` in `PageContext` name table.
`String EXCEPTION`	Name used to store uncaught exception in `PageContext` name table.
`String OUT`	Name used to store current `JspWriter` in `PageContext` name table.
`String PAGE`	Name used to store the generated servlet in the `PageContext` name table.

int PAGE_SCOPE	Value representing page scope. PAGE_SCOPE is the default scope.
String PAGECONTEXT	Name used to store the current PageContext in its own name table.
String REQUEST	Name used to store ServletRequest in the PageContext name table.
int REQUEST_SCOPE	Value representing request scope.
String RESPONSE	Name used to store ServletRequest in the PageContext table.
String SESSION	Name used to store HttpSession in PageContext name table.
int SESSION_SCOPE	Session scope (assuming JSP participates in a session).

The JspError and JspException Classes

The JspError and JspException classes behave as expected, with constructors that accept a string argument as a default message.

The javax.servlet.jsp.tagext Package

The javax.servlet.jsp.tagext package contains the classes and interfaces needed to support the use of custom JSP tags. Table A-10 lists the interfaces and classes that constitute the javax.servlet.jsp.tagext package.

Table A-10: Components of the javax.servlet.jsp.tagext Package

Component	Type	Description
BodyTag	Interface	Extends the Tag interface by defining additional methods for the tag-handler class to access and to manipulate the tag body.
Tag	Interface	Describes the basic protocol between a tag-handler class and the class that implements the JSP page.
BodyContent	Abstract Class	A subclass of JspWriter used to hold the results of evaluating a tag body to a tag-handler class that implements the BodyTag interface.
BodyTagSupport	Class	Class that contains methods to assist in writing tag-handler classes that implement the BodyTag interface.
TagAttributeInfo	Class	Instances of TagAttributeInfo contain information on Tag attributes derived from the Tag Library Descriptor file.
TagData	Class	Tag instance attribute and value pairs.
TagExtraInfo	Abstract Class	Extra tag information. This class is coded in the Tag Library Descriptor file.

TagInfo	Class	Tag information for a custom tag. Instances of this class have values derived from the Tag Library Descriptor file.
TagLibraryInfo	Abstract Class	Information on the tag library. Instances of this class have values derived from the Tag Library Descriptor file.
TagSupport	Class	A base class used to define new tag handlers.
VariableInfo	Class	Information on scripting variables used by a tag-handler class at runtime.

Let's take a closer look at these interfaces and classes.

The BodyTag Interface

The BodyTag interface extends Tag by defining additional methods to enable a Tag handler to access its body.

The interface provides two new methods. The first method is invoked with the BodyContent for the evaluation of the body. The second method reevaluates after every body evaluation. Table A-11 lists the methods defined in the BodyTag interface.

Table A-11: Methods in the BodyTag Interface

Method Signature	Description
int doAfterBody()	Performs processing after the body of a custom tag has been evaluated.
int doInitBody()	Performs processing before processing of the tag body.
void setBodyContent()	Setter method for the BodyContent property.

The BodyTag interface also provides a class variable with the following declaration:

```
static int EVAL_BODY_TAG
```

This declaration requests the creation of new BodyContent to evaluate the body of this tag.

The Tag Interface

The Tag interface defines the basic protocol between a Tag handler and JSP page implementation class, describing the life cycle and the methods to be invoked at start and end tag. Table A-12 shows the methods of the Tag interface.

Table A-12: Methods in the Tag Interface

Method Signature	Description
int doStartTag()	Processes the start tag.
int doEndTag()	Processes the end tag.
Tag getParent()	Returns the parent tag or null if no parent exists.
void release()	Calls on a tag-handler class to release the state of the tag.

Declaration	Description
`void setPageContext( PageContext thisPC )`	Sets the current page context. This method is called before calls to `doStartTag`.
`void setParent( Tag pTag )`	Establishes `pTag` as the `parent` tag of the current tag.

The `Tag` interface also defines a few class variables, which are listed in .

Table A-13: Class Variables Declared in the Tag Interface

Declaration	Description
`static int EVAL_BODY_INCLUDE`	Returned by the `doStartTag` method to include the evaluation of the tag body into the output stream (current instance of `JspWriter`).
`static int EVAL_PAGE`	Returned by the `doEndTag` method to direct the JSP to continue to evaluate the JSP page.
`static int SKIP_BODY`	Returned by the `doStartTag` and `doAfterBody` methods to omit evaluation of the tag body.
`static int SKIP_PAGE`	Returned by the `doEndTag` method to omit evaluation of the remainder of the JSP page.

The BodyContent Abstract Class

The `BodyContent` class is a subclass of `JspWriter` that can be used to process body evaluations so they can be re-extracted at a later time. lists the methods available in class `BodyContent`.

Table A-14: Methods in the BodyContent Class

Method Signature	Description
`void clearBody()`	Clears the contents of a `BodyContent` object.
`int flush()`	Redefines `flush` to make a call to `flush` illegal for objects of `BodyContent`.
`JspWriter getEnclosingWriter()`	Returns a reference to the `JspWriter` object from which the current `BodyContent` is derived from.
`abstract Reader getReader()`	Returns the instance of `BodyContent` as a `Reader`.
`abstract String getString()`	Returns the instance of `BodyContent` as a `String`.
`abstract void writeOut()`	Writes the instance of `BodyContent` to a `Writer`.

The BodyTagSupport Class

The `BodyTagSupport` class is a base class for defining tag handlers that implement the `BodyTag` interface. The `BodyTagSupport` class implements the `BodyTag` interface and adds additional convenience methods including `getter` methods for the `BodyContent` property and methods to get at the previous `JspWriter` "out" object. lists the methods available with the `BodyTagSupport` class.

Table A-15: Methods in the BodyTagSupport Class

Method Signature	Description
`int doStartTag()`	This method is invoked first when a tag is encountered.
`int doEndTag()`	Invoke this method when processing the `end` tag.
`int doInitBody()`	Invoke this method before evaluating the tag body.
`BodyContent getBodyContent()`	Returns the current `BodyContent` object.
`JspWriter getPreviousOut()`	Returns the enclosing `JspWriter`.
`void release()`	Resets the state of the tag.
`void setBodyContent()`	Prepares for tag body evaluation.

The TagAttributeInfo Class

This class contains information on Tag attributes found in the Tag Library Descriptor file (tld). Only the information needed to generate code is included in this reference. Additional information such as SCHEMA can be found in the complete JSP Specification (java.sun.com/products/jsp).

Table A-16 lists the methods for the `TagAttributeInfo` class.

Table A-16: Methods in the TagAttributeInfo Class

Method Signature	Description
`boolean canBeRequestTime()`	Indicates whether this attribute can hold a request-time value
`static TagAttributeInfo getIDAttribute( tagAttributeInfo[] tai )`	Returns the `ID` attribute (if one exists) in the attribute list argument.
`String getName()`	Returns the name of the attribute.
`String getTypeName()`	Returns the type of the attribute as a string.
`boolean isRequired()`	Indicates whether this attribute is required or not.
`String toString()`	Overrides Object.toString.

The `TagAttributeInfo` class also contains a class variable declared as follows:

`static final String ID = "ID"`

TheTagData Class

The `TagData` class contains translation-time information for the attributes and values of a tag instance. `TagData` is only used as an argument to the `isValid` and `getVariableInfo` methods of `TagExtraInfo`, which are invoked at translation time. Table A-17 lists the methods for the `TagData` class.

Table A-17: Methods in the TagData Class

Method Signature	Description
`Object getAttribute( String attName )`	Returns the value of the attribute named `attName` or null if no attribute exists.
`String getAttributeString( String attName )`	Returns the value of the attribute named `attName` as a string or null if no attribute exists.

`String getID()`	Returns the value of the `ID` type attribute or null if no `ID` attribute exists.
`void setAttribute( String attName, Object attValue )`	Sets the value of the attribute named `attName` to the value `attValue`.

The `TagData` class contains a variable coded as follows:

`static Object REQUEST_TIME_VALUE`

This variable tells the JSP container that the value of an attribute is available as a run-time expression, but will not be available at translation time.

TheTagExtraInfo Class

This class provides extra information about a custom tag. To associate a `TagExtraInfo` class with a tag handler class, this class must be mentioned in the Tag Library Descriptor file (tld). This class must be used if the tag defines any scripting variables or if the tag wants to provide translation-time validation of the tag attributes. Table A-18 lists the methods for the `TagExtraInfo` class.

Table A-18: Methods in the TagExtraInfo Class

Method Signature	Description
`TagInfo getTagInfo()`	Returns the instance of `TagInfo` for the tag class.
`VariableInfo[] getVariableInfo( TagData td )`	Returns information on scripting variables defined by this tag.
`boolean isValid()`	Translation-time validation of tag attributes.
`void setTagInfo( TagInfo ti )`	Sets the `TagInfo` object for this class.

The class contains an instance variable coded as follows:

`protected TagInfo tagInfo ;`

This instance variable represents the instance of `TagInfo` associated with instances of `TagExtraInfo`.

TheTagInfo Class

Tag information for a tag in a Tag Library; this class is instantiated from the Tag Library Descriptor file (tld). Table A-19 lists the methods for the `TagInfo` class.

Table A-19: Methods in the TagInfo Class

Method Signature	Description
`TagAttributeInfo getAttributes()`	Returns a `TagAttributeInfo` object describing the attributes of the tag or null if tag has no attributes.
`String getBodyContent()`	Returns the `BodyContent` object as a string.
`String getInfoString()`	Returns an information string coded in the tld.
`String getTagClassName()`	Returns the name of the class that provides the run-time handler for the tag.
`TagExtraInfo getTagExtraInfo()`	Returns the instance of the `TagExtraInfo` class, if any exist.

`TagLibraryInfo getTagLibrary()`	Returns the tag library owning this tag.
`String getTagName()`	Returns the tag name.
`VariableInfo[] getVariableInfo( TagData td )`	Returns information on the object created by this tag at runtime. Null means no such object created. Default is null if the tag has no "id" attribute, otherwise the array, `{"id", Object}`, is returned.
`String toString()`	Overridden version of `Object.toString`.

The `TagData` class also includes several class variables, used as arguments to the `getBodyContent` method, which are listed in .

Table A-20: Class Variables Declared in the TagInfo Class

Declaration	Description
`static String BODY_CONTENT _EMPTY`	The current instance of `BodyContent` is empty.
`static String BODY_CONTENT_JSP`	The current instance of `BodyContent` contains JSP code.
`static String BODY_CONTENT_TAG_DEPENDENT`	The current instance of `BodyContent` depends on the evaluation of another tag.

TheTagLibraryInfo Class

This class contains information found in the Tag Library Descriptor file about the tag library. lists the methods for the `TagInfo` class.

Table A-21: Methods in the TagLibraryInfo Class

Method Signature	Description
`String getInfoString()`	Returns the info string coded in the tld.
`String getPrefixString()`	Returns the prefix string used to reference tags within the JSP page.
`String getReliableURN()`	Returns the URN to the tld.
`String getRequiredVersion()`	Returns the version of the JSP container.
`String getShortName()`	Returns the preferred short name of the library.
`TagInfo getTag(String shortName )`	Returns the `TagInfo` object (`tagInfo`) for the library with the short name argument.
`TagInfo[] getTags()`	Returns an array of `TagInfo` objects — one for each tag described in the tld.
`String getURI()`	Returns the URI from the `taglib` directive for this library.

Instances of class `TagLibraryInfo` have instance variables that hold values returned from the `get` methods listed in .

The TagSupport Class

The `TagSupport` class is a base class for defining new tag handlers implementing the `Tag` interface. The `TagSupport` class is a utility class to be used as the base class for new tag handlers. The `TagSupport` class implements the `Tag` and `IterationTag`

interfaces and adds additional convenience methods including `getter` methods for the properties in `Tag`.

`TagSupport` has one static method that is included to facilitate coordination among cooperating tags. Many tag handlers extend `TagSupport` and only redefine a few methods. Table A-22 lists the methods available in the `TagSupport` class.

Table A-22: Methods in the TagSupport Class

Method Signature	Description
`int doEndTag()`	Invokes this method when processing the `end` tag.
`int doStartTag()`	Invokes this method when processing the `start` tag.
`static tag findAncestorWithClass( Tag from, Class tagClass )`	Finds the instance of the class named `tagClass` that is the closest ancestor to the tag named `from`.
`Tag getParent()`	Returns the tag instance enclosing this tag instance.
`String getTagId()`	Returns the value of the `ID` attribute for this tag (if it exists), or null.
`Object getValue( String key )`	Returns a value associated with the argument key.
`Enumeration getValues()`	Returns an enumeration representing all the values associated with this tag.
`void release()`	Invoked after a call to `doEndTag` to reset the state of the tag.
`void removeValue( String key )`	Removes a key/value pair associated with this tag.
`void setPageContext( PageContext pc )`	Sets the `PageContext` for this tag.
`void setParent( Tag ptag )`	Sets the parent (`ptag`) for this tag.
`void setTagID( String idAttr )`	Sets the `ID` attribute of the tag.
`void setValue( String key, Object value )`	Sets a value for a key/value pair in this tag.

The `TagSupport` class also has two instance variables that are coded as shown in the following:

```
protected String id ;
protected PageContext pageContext ;
```

The VariableInfo Class

This class contains information on the scripting variables that are created/modified by a tag at run-time. This information is provided by `TagExtraInfo` classes and it is used in the translation phase of JSP. Table A-23 lists the methods available in the `VariableInfo` class.

Table A-23: Methods in the VariableInfo Class

Method Signature	Description
`String getClassName()`	Returns the class name of the scripting variables coded in the tld as the `<variable-class>` element.
`boolean getDeclare()`	Returns the value of the `<declare>` element coded in the tld.
`int getScope()`	Returns the value of the `<scope>` element coded in the tld.
`String getVarName()`	Returns the name of the scripting variable coded in the tld as the `<variable>` element coded in the tld.

The `VariableInfo` class contains three class variables, listed in [Table A-24](#).

Table A-24: Class Variables Declared in the VariableInfo Class

Declaration	Description
`static int AT_BEGIN`	Variable is visible after the `start` tag.
`static int AT_END`	Variable is visible after the `end` tag.
`static int NESTED`	Variable is visible within the `start` and `end` tags.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

In this chapter, you've seen how you can leverage the features of JavaServer Pages to create clients for enterprise
beans. By using JSP pages, you can hide functions within JavaBeans or custom tags, thereby separating your
presentation from your logic.

You've seen how a JSP page can use a custom tag to locate an instance of a bean's home object. The single, empty
tag containing needed data as values of attributes allows a JSP page to work with the EJB architecture. Once the
page has a reference to a home object, the page can request the execution of bean methods like any EJB client.

Tag libraries are a powerful feature of JSP. By using tag libraries, your JSP pages contain less Java scriptlet code and
more tags. Since scriptlet code implies business logic, by keeping scriptlet code to a minimum, you'll have less
mingling of presentation and logic.

You've seen the enterprise bean code that performs the same functions as the code shown in Chapter 10. Aside from
including dummy method implementations as required by EJB, the code that accesses the database is mostly the
same as that shown in Chapter 10. However, by adhering to the EJB specification, your code now creates objects that
are distributed objects with location transparency and have access to transaction and security resources by way of the
EJB container.

EJB & JSP: Java On The Edge, Unlimited Edition

 by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Chapter 10: The "Make Money" Brokerage Application

In previous chapters I've covered the capabilities of JavaServer Pages and shown you some sample JSP code. In this chapter, you can see how JSP can be used in the construction of a brokerage application.

First, you can read about the functions available in our "Make Money" brokerage application. Next, you will read a description of the underlying data store and client scenario. Last, you will see the JSP and Java code that implements a typical client request.

The "Make Money" Application

The "Make Money" application is a small JSP program modeling an online stock trading system. This application will be used to model several aspects of JSP development that we have covered thus far in this book. We will see the use of JavaBeans in JSP pages. We will also see demonstrated the use of JSP error-handling tools discussed in the previous chapter. If this is one of your first JSP applications, you should finish this chapter with a better understanding of how you can use JSP successfully in developing Web applications.

The "Make Money" application provides clients with several functions. Once clients are successfully logged on, they can:

- View their account history

- View their portfolio

- Place a buy or sell order

- Change personal information

The application is hardly full-featured. My intent is to present the code required to implement some of the operations of the application. The next section describes the implemented features in more detail.

Implemented Application Features

Later in the chapter we will see the JSP and Java code that enables the following functions:

- Handling user logons

- Displaying a screen of selections

- Displaying the user's account history

The JSP and Java code for other options is similar to what is presented here. In other words, it is not necessary to

understand new concepts or capabilities of JavaServer Pages to implement the remaining features.

Before you delve into the code, let's take a look at the structure of the database that contains the customer, portfolio, and security information used by the application.

The Application Data Defined

The database for our application consists of four tables. Even though you don't need all four tables to implement the three features mentioned in the previous section, you can describe the tables to provide a more complete view of the application.

The four tables containing the data are described below.

- **CustomerInfo:** Contains the usual customer information (name, address, credit card info, and so on), an account ID (primary key), and a password. The customer must supply the correct password to gain access to the application.

- **CustomerPortfolio:** Contains an account ID (primary key), a stock ticker symbol, and a number representing how much of this security the customer holds.

- **SecurityInfo:** Contains a stock ticker symbol, a trade date, and the selling price of the security on the trade date. This table uses a compound key consisting of the stock ticker symbol and the trade date.

- **TransactionHistory:** Contains an account ID, a transaction date and type, a stock ticker symbol, and the number of shares traded on the transaction date. This table uses a compound key of the account ID, transaction date, stock ticker symbol, and the transaction type (buy or sell).

The preceding four tables have primary and foreign key relationships to ensure referential and data integrity. For sake of simplicity, the JSP and Java code in this sample application does not have code to capture errors arising from referential integrity constraint violations.

The Client Scenario

With the application data described, we can now look at a common client scenario. The application will allow a client to proceed through the following steps:

1. The client requests access to the application by entering an account ID and a password.

2. Once the application receives a matching account ID and password combination, the application displays a list of choices.

3. The client requests a listing of his or her transaction history; the application displays the list.

Without further delay, let's look at the implementation of this scenario in the "Make Money" application.

Logging on to the "Make Money" Application

The customer enters a URL that identifies the JSP that handles the interactions necessary to grant access to the client. Figure 10-1 shows the client logon screen.

Figure 10-1: Logging on to the application

Listing 10-1 shows the JSP code, `logon.jsp`, that displays the screen shown in Figure 10-1.

Listing 10-1: JSP for logon form (logon.jsp)

```
<html>
<head>
<title>Lou's Brokerage....Password Entry Screen</title>

<script language="Javascript">

 function giveFocus() {
   document.passwordform.acctNumber.focus() ;
 }
 function submit() {
   document.passwordform.submit() ;
 }
 function reset() {
   document.passwordform.reset() ;
   document.passwordform.acctNumber.focus() ;
 }

</script>

</head>

<jsp:include page="imagetable.html" flush="true" />

</center>

<center>
<form name="passwordform" action="checkLogin.jsp" method="POST" >


<p>Enter Your Account Number and Password in the Fields Below
<br>Then Click <b>Logon</b> to Continue
<br>
<hr width="50%">
<table>
<tr>
    <td><P>Enter Your Account Number:</td>
    <td><input type="text" name="acctNumber" value="" width="25"></td>
```

```
    </tr>
    <tr>
        <td><P>Enter Your Password:</td>
        <td><input type="password" name="enteredPassword" value="" width="25"></td>
    </tr>
    <tr>
        <td><input type="button" name="Logon" value="Logon" onClick="submit()"></td>
        <td><input type="button" name="Reset" value="Reset" onClick="reset()"></td>
    </tr>
</table>
<hr width="50%">
</center>
</form>
<%-- Here is the diagnostic when the user enters an invalid account number --%>
<%  String message = (String)session.getAttribute( "message" ) ;
    if ( message != null ) {
%>
<center><font color="red">
<%= message %>
</font></center>
<% } %>

</body>
</html>
```

Many of the code features presented in Listing 10-1 are discussed in the following sections.

Using jsp:include

The following line includes an HTML table that shows the pictures of the stock exchange floor and the piggy banks:

```
<jsp:include page="imagetable.html" flush="true" />
```

Because you're probably curious, Listing 10-2 shows the listing for the included HTML table, `imagetable.html`.

Listing 10-2: HTML to create the page banner (Imagetable.html)

```
<body bgcolor="#dddddd" topmargin=0>
<center>
<br>
<h1>
<img src="images\piggie.gif">
Make Money Brokerage
<img src="images\piggie.gif">
</h1>

<table>
<tr>
  <td><img src="images\nysefloor.jpg"></td>
  <td>   </td>
  <td><img src="images\nyseauction.jpg"></td>
</tr>
</table>
</center>
```

No surprises here, right? The use of the `jsp:include` tag enables you to use this HTML page header for any or all pages in the application.

Using a Form to Reference a JSP Page

The following line of code from directs the server to invoke the JSP page `checkLogin.jsp` when the client clicks the submit button located at the bottom of the HTML form:

```
<form name="passwordform" action="checkLogin.jsp" method="POST" >
```

Using a Form to Associate Client Inputs with Program Variables

The following lines of code from create the form elements `acctNumber` and `enteredPassword` that will be associated with program variables used in subsequent JSPs and JavaBean code.

```
<input type="text" name="acctNumber" value="" width="25"></td>
<input type="password" name="enteredPassword" value="" width="25">
```

Using Scriptlet Code to Display a Diagnostic

The following scriptlet code from causes `login.jsp` to display a message stating that the account number is not on file or the password entered for the account ID is invalid.

```
<%  String message = (String)session.getAttribute( "message" ) ;
    if ( message != null ) {
%>
<center><font color="red">
<%= message %>
</font></center>
<% } %>
```

The attribute `message` is set in the JSP page that checks the user login (`checkLogin.jsp`). If the account ID is not on file or the password does not match the entered ID, `checkLogin.jsp` sets a `session` variable named `message` to the diagnostic. The preceding scriptlet code checks the value of the message variable and, if not null, displays the message in red.

Checking the Account ID/Password Combination

Once the client enters an account ID/password combination and clicks "Logon," the application invokes the JSP page `checkLogin.jsp`. This JSP page does not contain any static code for display. The purpose of `checkLogin.jsp` is to access the `CustomerInfo` table to determine the following:

- Is the account ID entered on file?

- Is the password entered for an existing account ID the same as the password stored in the `CustomerInfo` table?

shows the code for `checkLogin.jsp`.

Listing 10-3: JSP to verify login (checkLogin.jsp)

```
<%@ page contentType="text/html"
        errorPage="errorpageex1.jsp"
        import="chapter10.*" %>

<%-- Determine if password matches account ID  --%>
<%
  String    enteredPassword       =
            request.getParameter("enteredPassword") ;
  String    acctNumber            =
            request.getParameter("acctNumber")  ;
```

```
  CustomerBean  customer           = new CustomerBean( acctNumber ) ;
  String        password           = customer.getPassword() ;

  boolean   redirectToLogin        = false ;

  if ( password.length() == 0 ) {
      session.setAttribute("message",
            "Account Number " + acctNumber +
      " Not on File. Enter Another Account Number") ;
      redirectToLogin = true ;

  }
  else
  if ( !password.equals(enteredPassword) ) {
     session.setAttribute("message",
     "Password Entered Does Not Match Password For Account " +
     acctNumber ) ;
     redirectToLogin = true ;
  }

  if ( redirectToLogin ) {
%>
<jsp:forward page="login.jsp" />
<% } else {
     session.setAttribute("customer", customer ) ;
%>
<jsp:forward page="showcustoptions.jsp" />
<% } %>
```

The first statement worthy of note in `checkLogin.jsp` is the *page directive*, shown here:

```
<%@ page contentType="text/html"
        errorPage="errorpageex1.jsp"
        import="chapter10.*" %>
```

The page directive serves several uses in `checkLogin.jsp`, as explained in the following sections.

Using the JSP Page Directive

Note the use of a JSP error page, `errorpageex1.jsp`, in the preceding code example. Any errors in JSP page processing cause the JSP engine to invoke `errorpageex1.jsp`. For more information on error page `errorpageex1.jsp`, see the sidebar "Explaining `errorpageex1.jsp`."

Explaining errorpageex1.jsp

The code in Listing 10-3 indicates `errorpageex1.jsp` as the page which should be used if any errors occur while processing the login page. Let's look at how this error page works.

```
<%-- Tell JSP that this is an error page --%>
<%@ page isErrorPage="true" %>

<html>
<head>
<title>An Error Has Occurred!!!</title>
</head>

<body bgcolor="#dddddd">
```

```
<P><font size=+3>
The Following Error Occurred on <%= new java.util.Date() %>
</font>
<br>
<hr>
<font color="red"><%=exception %>   <br>

<%

exception.printStackTrace();

%>
</font>
<hr>
<br>
<p>Call <b>4-4444</b> and report the above line in <font color="red">red</font>

</body>

</html>
```

First, we see the use of the `isErrorPage` directive, indicating that this page is an error page. Since this page is an error page, we have access to the implicit `exception` object. The use of the statement `<%=exception %>` outputs the error to the browser, giving the client a message as to what occurred. Here we also print the program stack trace of the exception to the browser. Normally the stack trace and exception messages would be saved for logging or debugging output (see Chapter 9), and what would be output here is a user-friendly message indicating what the user should do because of the error.

The next attribute set in the page directive is the `import` attribute. The `import` attribute of the page directive serves the same purpose as the import statement in a Java program. In this `import` attribute, you want the code in a package called `chapter10` to be known without qualification to our JSP.

Using the Implicit Request Object to Reference Entered Data

How does the `checkLogin.jsp` page know what values were entered by the client in the previously displayed JSP page, `login.jsp`? The following code in `checkLogin.jsp` references the `request` implicit object.

```
String    enteredPassword       =
          request.getParameter("enteredPassword") ;
String    acctNumber            =
          request.getParameter("acctNumber")  ;
```

The strings passed to the `getParameter` method must match the names coded for the form text elements in `login.jsp`.

Now that the `String` objects `enteredPassword` and `acctNumber` contain what the client entered, you need to see how `checkLogin.jsp` verifies that the account ID is on file and the password entered matches the password on file.

Matching Entered Data to Stored Data

The JSP uses Java code stored in a class called `CustomerBean` to verify the account ID and to check the entered password. Before looking at the code in `CustomerBean.java`, which does the actual checking, here's the JSP code that accesses the code in `CustomerBean.java`:

```
CustomerBean   customer        = new CustomerBean( acctNumber ) ;
String         password        = customer.getPassword() ;
```

The `password` string is the password stored on the `CustomerInfo` table for the entered account ID.

The `checkLogin.jsp` page checks if the `getPassword` method returns a password. If the returned password is blank, the account ID entered is not on file and `checkLogin.jsp` sets a message stating that fact. The following code reflects the above logic:

```
if ( password.length() == 0 ) {
      session.setAttribute("message",
            "Account Number " + acctNumber +
      " Not on File. Enter Another Account Number") ;
      redirectToLogin = true ;

  }
```

The Boolean `redirectToLogin` causes `checkLogin.jsp` to display the login page.

If the `getPassword` method returns a password, `checkLogin.jsp` compares the returned password with the entered password. If the two passwords do not match, `checkLogin.jsp` sets a message to that effect. The following code reflects this logic:

```
else
  if ( !password.equals(enteredPassword) ) {
     session.setAttribute("message",
     "Password Entered Does Not Match Password For Account " +
     acctNumber ) ;
     redirectToLogin = true ;
  }
```

Notice that `checkLogin.jsp` checks for a valid account ID first, followed by checking for a valid password. Both code blocks use the `setAttribute` method of class `Session`. By using the `session` object, the attributes set in one JSP are known to other JSPs sharing the session.

Finally, if either mismatch condition arises, the Boolean `redirectToLogin` is examined and, if true, `checkLogin.jsp` forwards processing back to the `login.jsp` page. If no mismatches are found, `checkLogin.jsp` saves the customer information in the session object and forwards the client to the "Show Options" screen. The following code reflects this logic:

```
  if ( redirectToLogin ) {
%>
<jsp:forward page="login.jsp" />
<% } else {
     session.setAttribute("customer", customer ) ;
%>
<jsp:forward page="showcustoptions.jsp" />
<% } %>
```

and show a mismatch of password and account ID, respectively.

Figure 10-2: Entered password does not match password on file

Figure 10-3: Account ID does not exist

Before we look at the `showcustomeroptions.jsp` page, let's explore the code in `CustomerBean.java`.

Creating the "Make Money" JavaBeans

So far we have looked at several of the JSPs in the "Make Money" application. Let's now get our first look at one of the JavaBeans in this application (Listing 10-4). In the "Make Money" application, the customer is the central figure. The CustomerBean models the necessary attributes that a customer would have.

Listing 10-4: JavaBean implementing the Customer (CustomerBean.java)

```
package chapter10 ;
import java.sql.* ;
public class CustomerBean {
```

```java
private String acctNumber     = "" ;
private String password       = "" ;
private String customerName   = "" ;
private String mailingAddress = "" ;
private String billingAddress = "" ;
private String creditCardType = "" ;
private String creditCardNum  = "" ;
private String expirationDate = "" ;

//Use the account number to access the customer table....
public CustomerBean( String accountID ) throws Exception {
     String persInfoQuery  = "select customername," +
             " mailingaddress, billingaddress, " +
            " creditcardtype, creditcardnumber," +
            "expirationdate, password " +
            "from customerinfo where accountid = " ;

     StatementBean sqlStmt = new StatementBean() ;
     String         query = persInfoQuery +
             "'" + accountID + "'";

     Connection  aConn = sqlStmt.connectToDB() ;
     Statement      stmt  = aConn.createStatement();
     ResultSet    myResultSet = stmt.executeQuery( query );

     if ( myResultSet.next() ) {
          acctNumber = accountID ;
          customerName =
              myResultSet.getString( "customername" ) ;
          mailingAddress =
              myResultSet.getString( "mailingaddress" ) ;
          billingAddress =
              myResultSet.getString( "billingAddress" ) ;
          creditCardType =
              myResultSet.getString( "creditcardtype" ) ;
          creditCardNum =
              myResultSet.getString( "creditcardnumber" ) ;
          expirationDate =
              myResultSet.getString( "expirationdate" ) ;
          password = myResultSet.getString( "password" ) ;
       }
    aConn.close() ;
}
//Get/Set methods follow
public String getAcctNumber() {
     return acctNumber ;
}
public void setAcctNumber( String acctnum ) {
     acctNumber = acctnum ;
}
public String getPassword() {
     return password ;
}
public void setPassword( String pswd ) {
     password = pswd ;
}
public String getCustomerName() {
     return customerName;
}
public void setCustomerName( String cName ) {
```

```
              customerName = cName ;
        }
        public String getMailingAddress() {
             return mailingAddress;
        }
        public void setMailingAddress( String mAddr ) {
             mailingAddress = mAddr ;
        }
        public String getBillingAddress() {
             return billingAddress;
        }
        public void setBillingAddress( String mAddr ) {
             billingAddress = mAddr ;
        }
        public String getCreditCardType() {
             return creditCardType ;
        }
        public void setCreditCardType( String cType ) {
             creditCardType = cType ;
        }
        public String getCreditCardNum() {
             return creditCardNum;
        }
        public void setCreditCardNum( String mAddr ) {
             creditCardNum = mAddr ;
        }
        public String getExpirationDate() {
             return expirationDate ;
        }
        public void setExpirationDate( String eDate) {
             expirationDate = eDate ;
        }

}
```

The code in `CustomerBean.java` is straightforward: `get` and `set` methods enable you to set instance properties from any JSP. The constructor is responsible for extracting information for a customer based on the value of the table's primary key, `acctNumber`.

The constructor for `CustomerBean.java` uses another class called `StatementBean.java`, which, in turn, uses code from another class called `SQLBean.java`. shows the code for `StatementBean.java`, while shows the code for `SQLBean.java`.

Listing 10-5: JavaBean implementation of a SQL statement (StatementBean.java)

```
package chapter10;
import java.sql.*;
import java.io.*;

public class StatementBean extends SQLBean
  {
  String passwordQuery = "select password from customerinfo " +
                          " where accountid = ";
  String accountInfoQuery = "select transactiondate, " +
                  " transactiontype, security, numbershares " +
                  "from transactionhistory where accountid = " ;

  String myDeleteQuery = "delete " ;
  String myInsertQuery = "insert into " ;
```

```java
    ResultSet myResultSet = null;
    public StatementBean() {super();}

    public String getPassword( String accountID ) throws Exception {

         String passwordOnDB = null ;
         String query    = passwordQuery  + accountID ;
        Statement stmt = myConn.createStatement();
        myResultSet = stmt.executeQuery( query );
        if ( myResultSet != null ) {
            myResultSet.next() ;
          passwordOnDB = myResultSet.getString( "password" ) ;
          myConn.takeDown() ;
        }

        return passwordOnDB ;
    }

    public boolean getAccountInfoQuery( String accountID ) throws Exception {
            String query = accountInfoQuery + accountID ;
        Statement stmt = myConn.createStatement();
        myResultSet = stmt.executeQuery( query );
        return (myResultSet != null);
    }

    public boolean get() throws Exception
      {
      return myResultSet.next();
      }

    public String getColumn( String inCol) throws Exception
      {
      return myResultSet.getString(inCol);
      }
    }
```

The `StatementBean` class contains code to perform the actual database connect through a superclass called `SQLBean`. The code for `SQLBean` is shown in Listing 10-6.

Listing 10-6: Code for SQLBean.java

```java
package chapter10 ;
import java.sql.*;
import java.io.*;

public class SQLBean
  {
  private String myDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
  private String myURL = "jdbc:odbc:stock";

  protected Connection myConn;

  public SQLBean() {}

  public void makeConnection() throws  Exception
    {
    Class.forName( myDriver);
    myConn = DriverManager.getConnection(myURL);
    }
```

```
public Connection connectToDB() throws  Exception
   {
   Class.forName( myDriver);
   return DriverManager.getConnection(myURL)   ;
   }

public void takeDown() throws Exception
   {
   myConn.close();
   }
}
```

SQLBean contains code that handles the actual connection and disconnection from the database containing tables for the application.

There's nothing specific or peculiar to using these Java classes with JSPs. The JSPs that require instances from these classes reference the instances through scriptlet code or bean references.

The StatementBean class also extracts the account history information. You can see references to the StatementBean class later in this chapter.

Next, let's examine the JSP that presents the user with a list of options — the JSP page named showcustoptions.jsp.

Examining the showcustoptions.jsp Page

Once the user has successfully logged into the "Make Money" application, the customer is able to access any of the account options. The showcustoptions.jsp page presents the customer with a list of all of the implemented options, as shown in <u>Figure 10-4</u>.

Figure 10-4: The list of user choices

Notice the inclusion of the client's name in the HTML page. shows the JSP code for `showcustoptions.jsp`.

Listing 10-7: JSP to display customer options (showcustoptions.jsp)

```jsp
<%@ page contentType="text/html"
         import="chapter10.*"
         errorPage="errorpageex1.jsp" %>

<%-- If we've gotten this far, we have a valid account number and password. --%>
<%-- Why not show the options list?  --%>

<%
  String          custoptions     = request.getParameter("custoptions") ;
  CustomerBean    customer        = (CustomerBean)session.getAttribute("customer") ;
  String          customerName    = customer.getCustomerName() ;

  if ( custoptions != null ) {
%>
<jsp:forward page="<%= custoptions %>" />
<% } %>


<html>
<head>
<title>Options for <%= customerName %> </title>
</head>

<jsp:include page="imagetable.html" flush="true" />

<center>

<form name="optionsform" action="showcustoptions.jsp" method="POST">

<p><%= customerName %>, select a Choice Below and Click <b>Proceed</b> to Continue<br>
<br>
<hr width="50%">
<table>
<tr>
    <td><input type="radio" name="custoptions" value="viewhistory.jsp" checked>
        View Your Account History
    </td>
</tr>
<tr>
    <td><input type="radio" name="custoptions" value="viewportfolio.jsp">
        View Your Portfolio
    </td>
</tr>
<tr>
   <td><input type="radio" name="custoptions" value="dotransaction.jsp">
       Place an Order
   </td>
</tr>
<tr>
    <td><input type="radio" name="custoptions" value="changepersonal.jsp">
        Change Personal Information
    </td>
</tr>
<tr>
    <td><p><input type="submit" name="proceed" value="Proceed"></td>
    <td></td>
</tr>
```

```
</table>
<hr width="50%">
</center>
</form>

</body>

</html>
```

The workings of the preceding page are based on associating a JSP corresponding to a user option with each value of the radio button on the form. In our case, when the client clicks the "View Your Account History" option, the form sets the value to the name of a JSP page corresponding to that option (`viewhistory.jsp`).

In particular, the attribute `custoptions` used in the form holds the value of the JSP page corresponding to the client selection. The following code in `showcustoptions.jsp` forwards JSP processing to the selected page.

```
<%
  String          custoptions      = request.getParameter("custoptions") ;

  if ( custoptions != null ) {
%>
<jsp:forward page="<%= custoptions %>" />
<% } %>
```

Other pieces of code in `showcustoptions.jsp` fetch the customer name, as shown here:

```
CustomerBean    customer        = (CustomerBean)session.getAttribute("customer") ;
 String       customerName    = customer.getCustomerName() ;
```

Assume the client clicks "View Account History." The `showcustoptions.jsp` forwards JSP processing to `viewhistory.jsp`. Let's take a look at the workings of `viewhistory.jsp` next.

Examining viewhistory.jsp

One of the implemented customer options is the ability to view a history of all transactions. This feature, displayed to the customer as "View Account History," is implemented in `viewhistory.jsp` with the help of one new class, `AccountHistory`. shows an HTML page generated by `viewhistory.jsp`.

Figure 10-5: The client's account history

Listing 10-8 shows the code for `viewhistory.jsp`.

Listing 10-8: Code for viewhistory.jsp

```jsp
<%@ page contentType="text/html"
         import="chapter10.*"
         errorPage="errorpageex1.jsp" %>

<%-- Display the account activity for this customer --%>

<%
  CustomerBean    customer         = (CustomerBean)session.getAttribute("customer") ;
  String          customerName     = customer.getCustomerName() ;
  String        acctNum        = customer.getAcctNumber() ;
%>
<html>
<title>Transaction History for <%=customerName %> <%= acctNum %> </title>
<body bgcolor="#dddddd" >
<center>

<%-- Put in the pictures for the page top --%>

<jsp:include page="imagetable.html" flush="true" />

<%
  AcctHistory anAcctHistory = new AcctHistory() ;

  if ( anAcctHistory.getHistoryThisAccount( acctNum ) ) {

%>
<p><%= customerName %>, here is a list of your transactions
<br>
<hr width="50%">
<form name="historyform" action="showcustoptions.jsp" method="POST">

<table>

<%
      while ( anAcctHistory.getNextHistRecord() ) {
%>
```

```
<tr><td>
 On <%= anAcctHistory.getColumn("transactiondate") %>, you traded
    <%= anAcctHistory.getColumn("numbershares") %> shares of
    <%= anAcctHistory.getColumn("security") %> on a
    <%= anAcctHistory.getColumn("transactiontype") %> order.
    </td>
 </tr>
 <tr bgcolor="red"><td> </td> </tr>

<%
     }

  }
%>
<tr>
    <td><p><input type="submit" name="Return" value="Return to Customer Options"></td>

</tr>
</table>
</form>
<hr width="50%">
</center>


</body>
</html>
```

The `viewhistory.jsp` page uses much of the same code as we've seen in the other JSP pages in this application, so we won't cover them again. What's new in this JSP page is the use of an instance of an account history object from class `AccountHistory`. Before exploring the `viewhistory.jsp` page further, let's first take a look at the `AccountHistory` class.

Examining the AccountHistory Class

The `AccountHistory` class contains code to access the `TransactionHistory` table given a valid account ID. shows the code for the `AccountHistory` class.

Listing 10-9: AccountHistory.java

```
package chapter10 ;
import java.sql.* ;
public class AcctHistory extends SQLBean {

     private String acctNumber = "" ;
     private String transactionDate = "" ;
     private String transactionType = "" ;
     private String security = "" ;
     private String numberShares = "" ;

     public ResultSet myResultSet ;

     public AcctHistory() { super(); } ;

     //Use the account number to access the history table....
     public  boolean getHistoryThisAccount( String accountID )
                                                throws Exception {
         String histQuery  = "select accountid, transactiondate," +
               " transactiontype, security, numbershares " +
```

```java
            "from transactionhistory where accountid = " ;

        AcctHistory sqlStmt  = new AcctHistory() ;
        String query = histQuery + "'" + accountID + "'";

        myConn = sqlStmt.connectToDB() ;
        Statement stmt = myConn.createStatement();
        myResultSet = stmt.executeQuery( query );

        return myResultSet != null ;
    // aConn.close() ;
}
public AcctHistory getHistoryRecord( ) throws Exception {

        AcctHistory aHistRec = new AcctHistory() ;
        aHistRec.acctNumber =
            myResultSet.getString( "accountid" ) ;
        aHistRec.transactionDate =
            myResultSet.getString( "transactiondate" ) ;
        aHistRec.transactionType =
            myResultSet.getString( "transactiontype" ) ;
        aHistRec.security =
            myResultSet.getString( "security" ) ;
        aHistRec.numberShares =
            myResultSet.getString( "numbershares" ) ;

        return aHistRec ;

}

public boolean getNextHistRecord() throws Exception
{
        return myResultSet.next();
}

public String getColumn( String inCol) throws Exception
{
        return myResultSet.getString(inCol);
}

public void takeDown() throws Exception
{
        myConn.close();
}

//Get/Set methods follow
public String getAcctNumber() {
        return acctNumber ;
}
public void setAcctNumber( String acctnum ) {
        acctNumber = acctnum ;
}
public String getTransactionDate() {
        return transactionDate ;
}
public void setTransactionDate( String tdat) {
        transactionDate = tdat;
}
public String getTransactionType() {
        return transactionType;
}
public void setTransactionType( String ttyp) {
```

```
                transactionType =  ttyp;
        }
     public String getSecurity() {
             return security;
        }
     public void setSecurity( String sec) {
             security = sec;
        }
     public String getNumberShares() {
             return numberShares;
        }
     public void setNumberShares( String nShrs) {
             numberShares = nShrs ;
        }
}
```

Revisiting viewhistory.jsp

Now that we've looked at the implementation of the `AccountHistory` class, let's return to `viewhistory.jsp`. The JSP generates an HTML table inside a form. The form includes a button that enables the user to return to the customer options screen.

The HTML table writes a line of account history between two red lines. The following code accesses the instances of `AccountHistory` that contain history information:

```
<%
  AcctHistory anAcctHistory = new AcctHistory() ;

  if ( anAcctHistory.getHistoryThisAccount( acctNum ) ) {

%>

<%
      while ( anAcctHistory.getNextHistRecord() ) {
%>
<tr><td>
 On <%= anAcctHistory.getColumn("transactiondate") %>, you traded
    <%= anAcctHistory.getColumn("numbershares") %> shares of
    <%= anAcctHistory.getColumn("security") %> on a
    <%= anAcctHistory.getColumn("transactiontype") %> order.
    </td>
 </tr>
 <tr bgcolor="red"><td> </td> </tr>

<%
      }

  }
%>
```

Note   Some HTML table code has been omitted from this code sample to enable you to view the JSP code that accesses and lists the account history.

The JSP creates an instance of the `AccountHistory` class and then uses methods in the `AccountHistory` class (`getHistoryThisAccount` and `getNextHistRecord`) and a method in the superclass SQLBean (`getColumn`) to access history information.

The remaining code in `viewhistory.jsp` formats the fetched information into an HTML table and provides a client with a way of returning to the options screen.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Chapter 9: JSP Errors and Debugging

Overview

Few processes execute as smoothly as originally planned. Nowhere is this more true than in the world of
programming. Even unseasoned programming novices know that programming errors and the debugging needed to
locate and repair such errors are part of the job.

This chapter covers how to deal with programming errors related to JSP development. You'll read about the JSP
features that can route errors to specific JSP pages, and then spend some time examining JSP translation and
runtime errors. You'll also explore JSP-specific exception classes.

After encountering and handling JSP errors, you must track down the root cause of the error. This chapter discusses
some effective techniques for debugging your JSP pages. You'll read about the straightforward methods, such as
writing to a log, and the not-so-straightforward, such as creating a custom debugging class.

Before you get into the details of JSP error handling, you may be wondering why the topic of JSP errors and
debugging deserves special treatment. The next section provides some answers.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

You've now seen many of the similarities and differences between JSP pages and servlets. It is necessary to understand these two technologies and how they work so that you can decide whether to use a JSP page or a servlet in a specific situation. Remember, above all, that neither JSP nor servlets are the final solution. As we return our focus to JSP in the following chapters, remember that the power of JSPs and servlets may best be found in using them together.

.

Using JSPs with Servlets

You've learned about JSP pages containing Java code that gets passed to servlets. You've witnessed JSPs using JavaBeans. You've seen examples of JSP custom-tag libraries. By now, you've discovered quite a lot about JavaServer Pages. But is there anything in the realm of generating dynamic Web content that JSPs cannot do? More specifically, does a situation exist in which using *only* JSPs to generate dynamic Web content is not the *best* solution?

JSP development assumes that your pages have a common presentation style and theme. JSPs may be inadequate at providing dramatically different looks based on different user inputs or different data. A JSP page that effectively displays data as an HTML table may do a poor job displaying data as an animated chart.

Then what is one to do? Although opinions may differ, you can *leverage servlets* to help your JSP page development in some situations. A servlet may start the dynamic content preparation process and then forward the request to one or more JSPs to complete the presentation. In the *Model 2* approach, the Model-View-Controller pattern is applied to this situation by having a servlet act as the controller, the beans as the model, and the JSPs as the view. A more recent point of view embraces the idea of a server acting as a *dispatcher* of requests to JSPs and other Java container objects, such as Enterprise JavaBeans. Appropriately, the term used to describe the above-cited point of view is called the *Dispatcher* approach, which is illustrated in Figure 8-2.
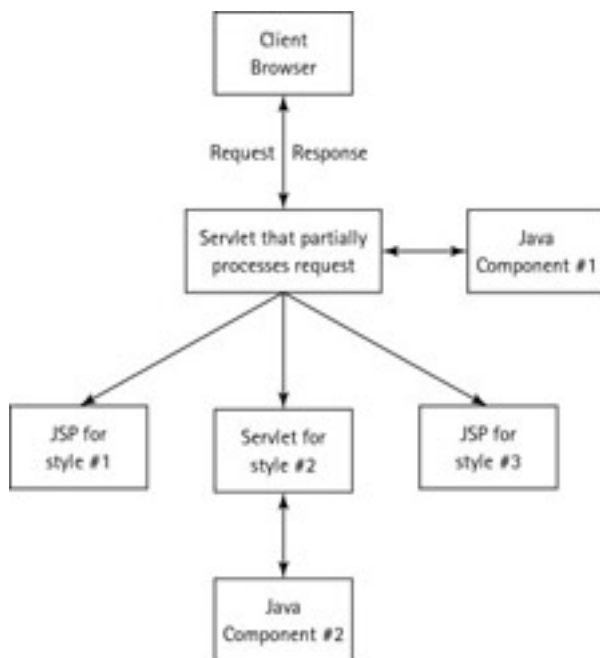
Figure 8-2: A servlet acting as a dispatcher

As Figure 8-2 shows, the servlet captures the request and manages the application flow. The dispatching servlet may not be responsible for generating any dynamic presentation content. Figure 8-2 shows the dispatching servlet

accessing a Java component in addition to dispatching requests and fetched data (from other components) to some JSP pages or other servlets.

Forwarding Requests from Servlets

In the recent past, the Java servlet programmer did not have a convenient way of implementing the preceding approach. But with the release of the Servlet API 2.1, the Java programmer can implement the `RequestDispatcher` interface. Implementations of `RequestDispatcher` are available at `ServiceContext` and can be used to send a request to a static resource (an HTML page, for example) or a dynamic resource (a JSP or servlet, for example).

The servlet programmer has two methods to implement: `forward`, to transfer control to another resource; or `include`, to handle the overall management of the request from, and the response to, the client.

The code, shown in the following, forwards a request from a servlet to a JSP:

```
//Assume request and response have their 'usual' meaning
RequestDispatcher rDisp =
    getServletContext().getRequestDispatcher("myJSPPage.jsp");
rDisp.forward(request, response ) ;
```

The URL argument in `getRequestDispatcher` is a *relative path* URL.
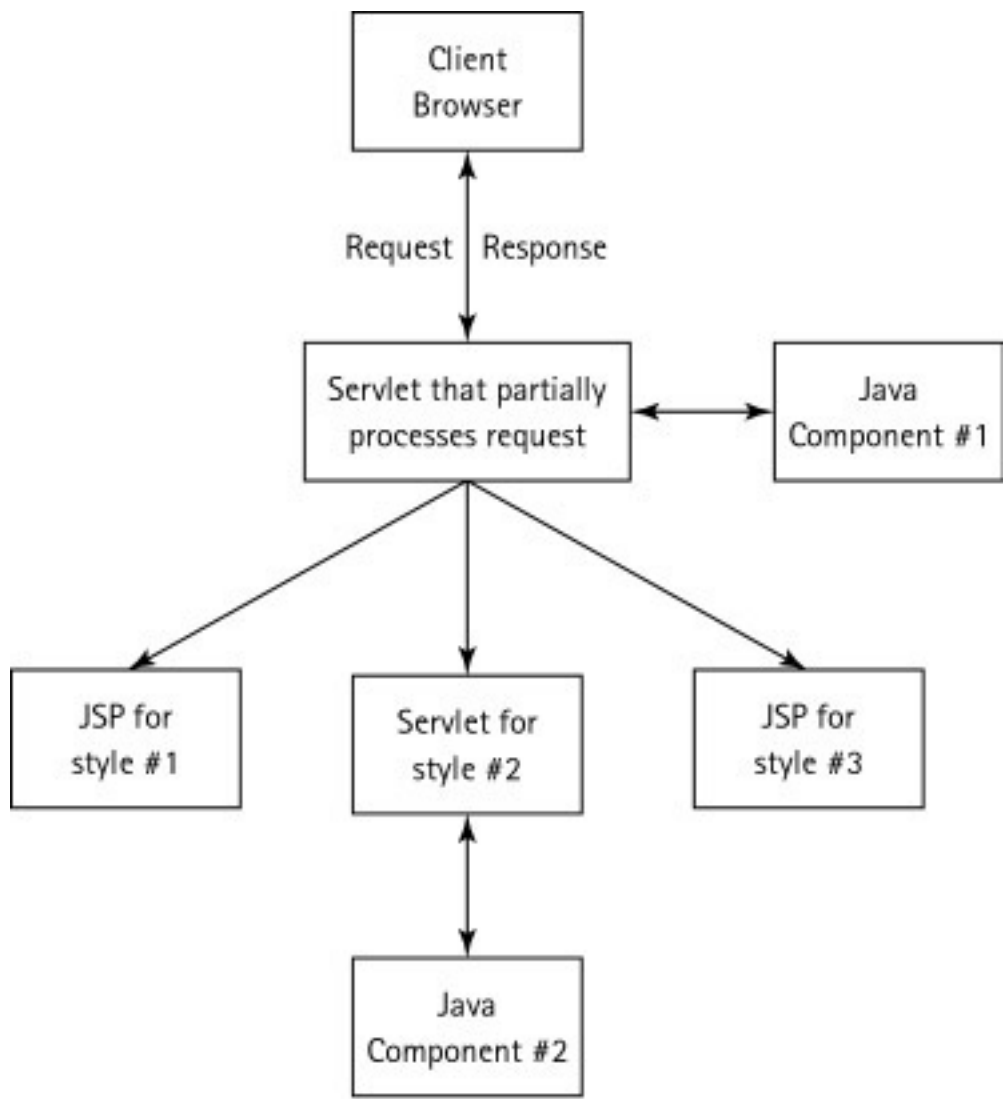
```
            ┌──────────────┐
            │    Client    │
            │   Browser    │
            └──────────────┘
                   ↕
          Request │ Response
                   ↕
    ┌──────────────────────┐        ┌──────────────────┐
    │ Servlet that partially│ ←────→ │      Java        │
    │  processes request    │        │  Component #1    │
    └──────────────────────┘        └──────────────────┘
       ╱          │          ╲
      ↙           ↓           ↘
┌──────────┐ ┌──────────┐ ┌──────────┐
│ JSP for  │ │Servlet for│ │ JSP for  │
│ style #1 │ │ style #2 │ │ style #3 │
└──────────┘ └──────────┘ └──────────┘
                   ↕
             ┌──────────────┐
             │     Java     │
             │ Component #2 │
             └──────────────┘
```

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Using the JSP/Servlet Environment

Now that we have reviewed the methods related to the JSP and servlet life cycles, let's take a close look at the environment in which they exist. Understanding the JSP/servlet environment will help us to better leverage the features of this environment that are useful for writing robust Web applications. Figure 8-1 depicts the relevant environmental components and the request/response flow between clients, JSPs, and servlets.
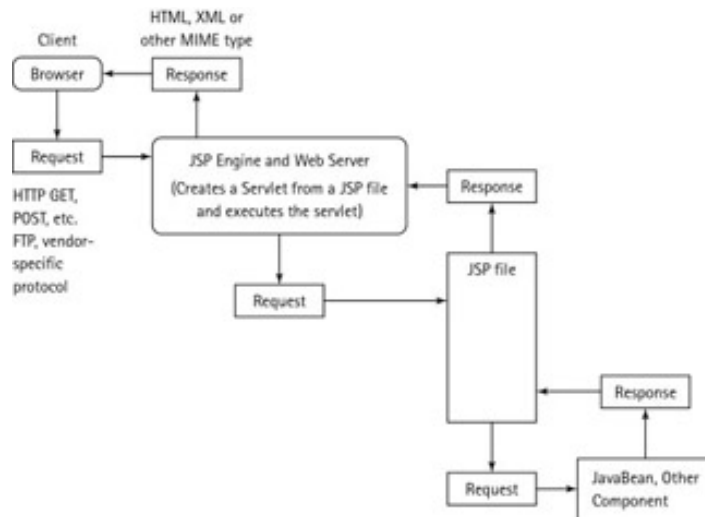
Figure 8-1: A high-level view of JSP and servlet processing

HTTP, FTP, or any other supported protocol request type originates from a browser (client) and is sent to the Web server. The JSP-enabled server recognizes the `.jsp` extension and realizes that the request is packaged with a JavaServer Page. The server translates the JSP page into a servlet. Along the way, the original request gets passed to the `_jspService` method in the generated servlet. After the servlet executes, perhaps by communicating with other Java components such as JavaBeans, the servlet returns a response in the form of an HTML, XML, plain text, or other MIME-type resource.

Useful Servlet Environment Features

The servlet environment provides important features to JSP pages. This section describes some of these features.

Session Management

One advantage of using servlets and, as a result, JSPs, is that servlets provide session management services. Recall that a session is a connection between a client and a server that enables the two to share data. The server identifies sessions by using a session key, which the server stores in a dictionary-type object.

Servlets and JSP pages use *cookies* by default to manage sessions. A cookie is a set of values, a name-value pair, which is sent to a client. Cookie implementations usually involve files stored on the client machine, and the location of such files is browser dependent.

It is important to remember that cookies can't be trusted to maintain sessions. For example, the client machine may have cookies disabled. In this situation, the servlet can use a technique called *URL rewriting*, which involves encoding the session key in the request URL. The servlet can decode the URL to extract the session key, thereby identifying the appropriate client belonging to that particular session.

Encoding and Parsing Form Data

Data sent with a `get` or `post` request may be encoded in a scheme known as *URL encoding*. The encoding replaces special characters, such as spaces and unprintable characters, with symbols and hex values. Names and values are encoded separately. You've seen this encoding on search engines before. For example, an advanced search in the *Google* search engine (http://www.google.com/) encodes search parameters as follows:

```
http://www.google.com/search?as_q=quantum+computing&num=10&btnG=Google+Search&as_oq=&as_epq=&as_eq=&as_occt=title&lr=&as_dt=i&sitesearch=&safe=off
```

Notice the name-value pairs **(q=quantum+computing)** with the + symbol replacing the blank, the **&** symbol connecting multiple search criteria, and the setting of hidden parameters (`safe=off`).

Servers are capable of *automatically* decoding this data. Whether the data is sent by a `get` request or `post` request, your JSP page can decode and retrieve the data by using the `getParameter` method of the `request` object. For example, the JSP expression shown below retrieves the value of the `q` parameter:

```
<%= request.getParameter("q") %>
```

Using the above method eliminates the necessity of writing code to parse the data or having to use the `java.net.URLEncoder` and `java.net.URLDecoder` classes.

Accessing Shared Data

JSPs and servlets may exchange data by using a set of methods available to objects of class `ServletContext`. In JSP lingo, the application scope represents objects derived from class `ServletContext`. Some of these methods are `getAttribute`, `getAttributeNames`, `setAttribute`, and `removeAttribute`.

Servlets and JSPs also can share initialization parameters and configuration settings by using methods available to objects of class `ServletConfig`, such as `getInitParameter` and `getInitParameterNames`.

Your Web server may have additional methods to expose various properties and attributes of your server environment to your JSP pages.

Servlets provide the JSP programmer with powerful features that are accessed with standard JSP expressions. You may think that you'll never have to code a servlet because all your dynamic Web page content needs are addressed by JSP. Although JSP brings unparalleled abilities to the Web application developer, you shouldn't throw out that servlet API documentation just yet. The following section touches on some cases in which you may want to use Java servlets with your JSPs.
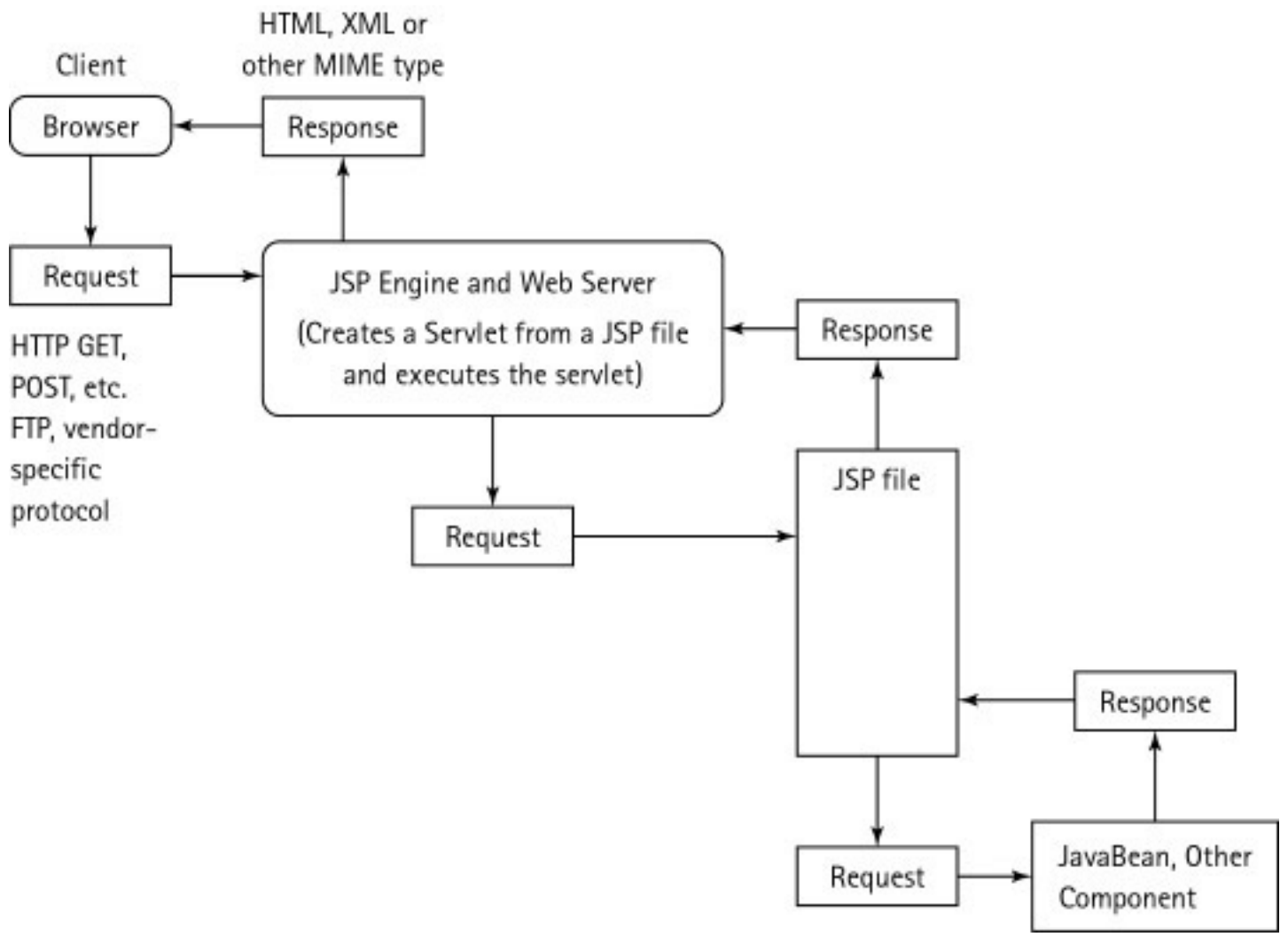
Client
HTML, XML or
other MIME type

Browser ← Response

Request

HTTP GET,
POST, etc.
FTP, vendor-
specific
protocol

JSP Engine and Web Server
(Creates a Servlet from a JSP file
and executes the servlet)

← Response

Request →

JSP file

Request → JavaBean, Other Component

← Response

.

Chapter 8: JSPs and Servlets Revisited

In previous chapters, you've read much about the relationship between JavaServer Pages and Java servlets. In this chapter, you can take a closer look at this relationship. You can delve into the servlet life cycle and discover the differences between a "raw" servlet and a JSP-generated servlet. You can also read about important servlet and JSP methods, along with learning about the servlet environment. In addition, you can gain insight into why you need to use servlets, which is discussed in this chapter's section on using Java servlets and JSPs together.

Examining the Servlet Life Cycle

Because JSPs get translated into Java servlets, the JSP life cycle closely parallels that of servlets. In brief, when you request a JSP page, the translator generates a servlet and the Java compiler on the server compiles the generated servlet. Then the server invokes the class loader to load the servlet and start execution.

If the servlet contains an `init` method, the container calls it. The `init` method runs only once. You can see that, for JSP-generated servlets, the analogue for the servlet `init` method is called `jspInit`. Both `init` and `jspInit` are optional methods.

After execution of `init` or `jspInit`, the servlet executes its `service` method. The JSP equivalent for the `service` method is `_jspService`. The server can run multiple threads accessing the service or the `_jspService` method simultaneously, or you can force the server to single thread the method's access by using a single-threaded model using the `isThreadSafe` attribute of the page directive.

Next, in servlets the `service` method invokes either the `doGet` or `doPost` method. The `doGet` and `doPost` methods of the servlet implement the `get` and `post` requests made from the client browser to the servlet, respectively. JSP-generated servlets do not have implementations for `doPost` and `doGet`. JSP-generated servlets perform both `post` and `get` requests in the `_jspService` method. Actually, a servlet can implement various `do` methods depending on the particulars of the HTTP request, such as `doPut` and `doDelete` methods.

When the server unloads a servlet, the server invokes the `destroy` method. The JSP-generated equivalent to the `destroy` method is `jspDestroy`. As with `init` and `jspInit`, `destroy` and `jspDestroy` are optional.

Writing the Minimal Servlet

Given that most of the servlet methods discussed above are optional, you may wonder what is the "smallest" or minimal servlet? Listing 8-1 provides an example of a minimal servlet.

Listing 8-1: A contender for the minimal servlet

```
import java.io.* ;
import java.text.* ;
import java.util.* ;
import javax.servlet.* ;
import javax.servlet.http.* ;

public class minimalServlet extends HttpServlet {

    public void doGet( HttpServletRequest request,
                       HttpServletResponse response)
             throws IOException, ServletException
    {
        response.setContentType("text/html") ;
        PrintWriter out = response.getWriter() ;
        out.println("<HTML> <BODY> Hello World </BODY>
 </HTML>" ) ;
        out.close() ;


    }
    public void doPost( HttpServletRequest request,
                        HttpServletResponse response)
                  throws IOException, ServletException
```

```
    {
        doGet( request, response ) ;
    }}
```

The above servlet is as bare bones as it gets. It is important to note that you do not have to override the `doGet` or `doPost` methods if you instead override the `service` method and handle all requests there. This would not be considered good form in servlet design, but it is an option. In the next section we will see that a minimal JSP-generated servlet is a bit different since the `doGet` and `doPost` methods do not exist.

Writing a Minimal JSP-Generated Servlet

The nuts and bolts of a JSP-generated servlet are dependent on the JSP-to-servlet translator used with a particular Web server. Listing 8-2 shows a rather simple JSP page in which Tomcat 3.2 generated the servlet. However, this page is not the simplest because it has actual JSP scripting elements — a simple page would have nothing but static text.

Listing 8-2: A simple JSP page with a couple of scripting elements

```
<%@ page contentType="text/html" %>
<html>
<head>
<title>Minimal JSP Page</title>
</head>

<body>
<%! String hello = "Hello World"; %>
<%= hello %>
</body>

</html>
```

Listing 8-3 shows the servlet that Tomcat generates from the JSP page in Listing 8-2.

Listing 8-3: JSP-generated servlet for minimal JSP page in Listing 8-2

```
package jsp.loutest;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;


public class _0002fjsp_0002floutest_0002floutest_0002ejsploutest_jsp_7
 extends HttpJspBase {

    /* begin [file="D:\\tomcat32\\Webapps\\examples\\jsp\\loutest\\loutest.jsp";from
=(7,3);to=(7,34)] */
        String hello = "Hello World";
    // end

    static {
    }
    public _0002fjsp_0002floutest_0002floutest_0002ejsploutest_jsp_7( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws JasperException {
    }

    public void _jspService(HttpServletRequest request,
    HttpServletResponse  response)
        throws IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
```

```
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                _jspx_init();
                _jspx_inited = true;
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html");
            pageContext =
_jspxFactory.getPageContext(this, request, response,
                                        "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

            /* HTML begin [file="D:\\tomcat32\\Webapps\\examples\\jsp\\loutest\\loutest.jsp";from=(0,35);to=(7,0)] */
            out.write("\r\n<html>\r\n<head>\r\n<title>Minimal JSP
Page</title>\r\n</head>\r\n\r\n<body>\r\n");
            // end
            /* HTML  begin
[file="D:\\tomcat32\\Webapps\\examples\\jsp\\loutest\\loutest.jsp";from=(7,36);to=(8,0)] */
                out.write("\r\n");
            // end
            /* HTML begin  [file="D:\\tomcat32\\Webapps\\examples\\jsp\\loutest\\loutest.jsp";from=(8,3);to=(8,10)] */
                out.print( hello );
            // end
            /* HTML  begin [file="D:\\tomcat32\\Webapps\\examples\\jsp\\loutest\\loutest.jsp";from=(8,12);to=(12,0)] */
                out.write("\r\n</body>\r\n\r\n</html>\r\n");
            // end

        } catch (Exception ex) {
            if (out.getBufferSize() != 0)
                out.clearBuffer();
            pageContext.handlePageException(ex);
        } finally {
            out.flush();
            _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

Whew! What is the reason for showing the generated code? The first reason is to demonstrate the labor and toil expended in generating a servlet from a very small JSP page. Second, examining generated code is best left to computers, not humans — don't you agree?

Note the absence of `jspInit`, `jspDestroy`, `doGet`, and `doPost` methods in the generated servlet (or take my word for it!). Also, take note that the bolded lines are generated in response to the JSP code in the page.

Because an implementation of the `init` method is not required for servlet execution and an implementation for the `jspInit` method is not required for JSP execution, why would you implement these two methods? You can discover why in the .

Examining the init and jspInit Methods

As previously mentioned, the `init` method is called when the servlet first loads. The `init` method is *not* called for each user request. Hence, `init` is used to perform one-time initializations. Actually, Java applets have an `init` method, which is not required, that serves the same function as `init` for servlets.

You may code the `init` method sans arguments as follows:

```
public void init() throws ServletException {
```

Also, you may pass an object of `ServletConfig` to `init` as follows:

```
public void init( ServletConfig sConfig )
 throws ServletException {
super.init( sConfig ) ;
```

You would use the second signature for `init` when your servlet requires server settings. Creating server settings is dependent on the server being used. Some

servers use a configuration file, whereas others use a GUI to set values for server settings.

Note the bolded invocation of the superclass constructor in the second example. Do yourself a favor and code the call to `super.init` as the *first line* in your `init` implementation when you require an object of `ServerConfig`.

As with `init`, `jspInit` is not required for JSP execution. However, you can code a `jspInit` method in your JSP page, which is passed to the generated servlet. The servlet engine executes the `jspInit` method only once upon loading the generated servlet. Listing 8-4 shows how to code a `jspInit` method in your JSP pages.

Listing 8-4: Example of jspInit method in a JSP

```
<%@ page contentType="text/html" %>
<html>
<head>
<title>jspInit</title>
</head>

<body>
<%! String hello = "Dummy Value";
    public void jspInit() {
     hello = "Initial value 'Hello World' changed in
 jspInit()" ;
}%>
Java variable hello is now <b><%= hello %></b>

</body>

</html>
```

When this page runs, the initial value of `Dummy Value` is changed by the assignment inside the `jspInit` method. In a real-world application, you would not see one simple String value overriding another, as is seen in this listing. You might override the `Dummy Value` with information retrieved from a bean or a database, depending on your application needs.

Rather than show you the vast amount of code generated by the JSP translator here, please take my word that this generated servlet contains a `jspInit` method.

Examining the destroy and jspDestroy Methods

It is unnecessary for you to code an implementation of the `destroy` method for your servlets. However, if you do, the server invokes your `destroy` method before unloading your servlet. The `destroy` method is a good place to perform various cleanup activities, such as closing database connections and writing any remaining persistent data to disk.

The JSP equivalent to `destroy` is `jspDestroy`. As with `destroy`, `jspDestroy` is not required for JSP execution. The `jspDestroy` method serves the same purpose as the `destroy` method for "raw" servlets. As with `jspInit`, you may code an implementation of `jspDestroy` in your page, or use a page directive to include an implementation.

A good rule of thumb is that if you code a `jspInit` method that grabs resources, such as pooled database connections, you should code a `jspDestroy` method to release the grabbed resources.

Note Other than `jspInit` and `jspDestroy`, you cannot code methods that start with `jsp`, `jspx`, `_jsp`, or `_jspx`. These method prefixes are reserved for future use by Sun.

Examining the service and _jspService Methods

You do not need to code an implementation of the `service` method. In fact, it is best if you do not override `service`. Instead, it is more effective to override the `doGet` and `doPost` methods. Your main advantages of overriding `doGet` and `doPost` as opposed to overriding `service` are as follows:

- You can add other `do` methods more easily when you override `doGet` and `doPost`. Overriding `service` removes your ability to add these other methods easily, especially if your servlet is then subclassed.

- You have automatic support for various requests, such as `TRACE` and `OPTIONS` requests, even if your servlet is subclassed.

The accepted way of overriding the same action for a POST or GET is to override `doGet` and `doPost` in a servlet and have `doPost` invoke `doGet`, or vice versa. (Refer to Listing 8-1 for an example of `doPost` calling `doGet`.)

The `_jspService` method is *required* for JSP execution. However, you must *never* code a `_jspService` method. The `_jspService` method is automatically generated by the JSP translator. `_jspService` is the "meat and potatoes" of the JSP; most of your JSP code finds its way into the `_jspService` method.

Table 8-1 summarizes the servlet and JSP methods discussed in the preceding sections.

Table 8-1: Summary of Important Servlet and JSP Methods

Method	Servlet or JSP	Required	Description/Notes
`init`	Servlet	No	Performs one-time initializations.
`jspInit`	JSP	No	Same as `init`. JSP author may provide code in JSP.
`service`	Servlet	No	Handles the request. Not a good idea to code an implementation
`_jspService`	JSP	Yes	Handles the HTTP request from the JSP page. The JSP author does not have to implement it. It is implemented by the JSP translator.
`doGet`	Servlet	No	Handles an HTTP GET request. Required only if the servlet must handle GET requests.
`doPost`	Servlet	No	Handles an HTTP POST request. Required only if the servlet must handle POST requests.
`destroy`	Servlet	No	Performs cleanup immediately prior to servlet purge by server.
`jspDestroy`	JSP	No	Performs cleanup immediately prior to JSP purge by server.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Summary

The topic of JSP tag extensions is relatively recent, being introduced in JSP release 1.1. The draft specification for
JSP release 1.2 discusses additional features of the JSP tag extensions. Searching for "JSP tag extensions" at
http://www.google.com/ returns over 1,300 sites. Those who are "in the know" realize that JSP tag libraries are
an essential component of JSP technology.

Top

Prev   Next

Writing an Example Nested Tag

Although the `formatLine` and `repeatLine` tags in the [previous section](#) were shown to execute when nested, the tags can execute independently of each other. In other words, the presence or absence of one tag has no effect on the execution of the other. At times, one tag requires a specific parent to execute, or a parent requires specific children. In JSP terms, parent/child tags that depend on one another for proper execution are called *nested tags*.

Writing nested tags is very similar to writing the sort of tags you've already learned. The difference is that the objects (beans) in the nested tags need to communicate. The tag library interfaces have methods that enable a child tag to determine its parent. Once done, you can invoke parent tag methods from the child tag.

The following sections describe how to construct a *case construct* as a series of nested tags. The general format is as follows:

```
<aTagLib:case value="some_value">
   <aTagLib:when value="case_value1">
    JSP to evaluate when some_value = case_value1
   </aTagLib:when >
    <aTagLib:when value="case_value2">
    JSP to evaluate when some_value = case_value2
   </aTagLib:when >
   <%-- Other casevalue tags may follow --%>
   <aTagLib:otherwise>
    JSP to evaluate when some_value not equals case_values
   </aTagLib:otherwise>
</aTagLib:case>
```

The relationships you need to enforce are as follows:

- A `when` tag must be enclosed within a `case` tag. The code has to know about the existence of a `case` tag (parent) when processing the `when` tag (child).

- An `otherwise` tag must be enclosed within a `case` tag. The code has to know about the existence of a `case` tag (parent) when processing the `otherwise` tag (child).

- The existence of an `otherwise` tag requires the existence of at least one `when` tag. The code has to know about the existence of a past-processed `when` tag when processing the `otherwise` tag.

Of course, you may want to implement the common understanding of a case construct.

The case Statement Tag

The `case` statement tag is the first tag that we'll examine in looking at the implementation of these nested tags. This tag is

the parent tag of the `when` and `otherwise` tags.

The code uses indicators (flags, if you will) as properties in the parent tag handler class, which you can access from the child classes. You must have two indicators: `whenstatementfound` indicates the presence of a `when` statement within the `case` tag and `whenstatementvaluefound` indicates the presence of a `when` statement with a value that matches that of the `case` statement. Listing 7-10 shows the tag handler for the case statement.

Listing 7-10: Tag handler class for the case tag

```
package chapter7;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

import java.io.*;

public class casestatement extends TagSupport {
    //This is the value to match on some when tag
    private String value ;
    //This boolean states whether or not at lease one
    //whenstatement is found
    private boolean whenstatementfound = false ;
    //This boolean states whether or not one of the when
    //statements has a value that matches that of the case
    //statement.
    private boolean whenstatementvaluefound = false ;

    public void setValue( String val) {
            value = val ;
    }
    public String getValue() {
            return value ;
    }
    public void setWhenstatementfound( boolean found) {
            whenstatementfound = found ;
    }
    public boolean getWhenstatementfound() {
            return whenstatementfound ;
    }
    public void setWhenstatementvaluefound( boolean found) {
            whenstatementvaluefound = found ;
    }
    public boolean getWhenstatementvaluefound() {
            return whenstatementvaluefound ;
    }
    public int doStartTag() {
            return EVAL_BODY_INCLUDE ;
    }
}
```

The code for the `case` tag (the outer tag) merely establishes the indicator properties and the `value` attribute that may or may not match some `when` tags.

The `doStartTag` method coded here instructs the JSP container to continue to evaluate the tag body. Notice that, because you are not using the tag body by a call to `doAfterBody`, the `casestatement` class extends the convenience class `TagSupport`, not `BodyTagSupport`.

At to this point, you haven't seen anything new in this tag handler. Next, you can take a look for an implementation of the

`when` tag, which shows how to access parent class properties.

The when Statement Tag

The `when` tag is the child tag of the `case` tag and needs to access information in the parent tag's implementing class. For this reason, you need a method that identifies the class *already instantiated* with a bean from the parent tag's implementing class. The method `findAncestorWithClass` does exactly that; `findAncestorWithClass` returns the instance of the parent tag class, thereby enabling child classes access to parent tag properties. Listing 7-11 shows the code for the `when` statement tag.

Listing 7-11: Tag handler class for the when tag

```
public class whenstatement extends BodyTagSupport {
    private String value ;
    public void setValue( String val) {
          value = val ;
    }
    public String getValue() {
          return value ;
     }
    public int doStartTag() throws JspException {
          //See if this is enclosed within a case tag
          casestatement caseTag =
                      (casestatement) findAncestorWithClass( this, casestatement.class );
          if (caseTag == null)
            throw new
             JspException("when tag not enclosed in case tag") ;
        else //set when statement found indicator
           caseTag.setWhenstatementfound( true ) ;
        //See if this is the when statement that has a value
        //matching that of the case tag
        if ( caseTag.getValue().compareTo( getValue() ) == 0 ) {
             //Set indicator in case tag to indicate a match
             caseTag.setWhenstatementvaluefound( true ) ;
             return EVAL_BODY_TAG ;
         }
         else
             return SKIP_BODY ;

    }

     public int doAfterBody() throws JspException {
          BodyContent tagBody = getBodyContent() ;
          String tagBodyAsString = tagBody.getString() ;
          try {
             JspWriter out = tagBody.getEnclosingWriter() ;
             out.print( tagBodyAsString ) ;
             } catch (IOException ex) {
                   throw new JspTagException(ex.toString());
             }              return SKIP_BODY ;
     }
}
```

The `doStartTag` method enables you to determine if a parent (case) class exists or if the value coded in the `when` tag matches that coded in the `case` tag. When a match of values is found, code in `doStartTag`, which sets the property `whenstatementvaluefound` in the `case` tag class to true; the `otherwise` tag accesses this indicator as you can see in

the . The `doAfterBody` method lists the tag body.

The otherwise Statement Tag

The code for the `otherwise` tag handler is very similar to the code for the `when` tag handler. Decisions to process the tag body are made in the `doStartTag` method; the `doAfterBody` tag lists the tag body to the screen. Listing 7-12 shows the code.

Listing 7-12: Tag handler class for the otherwise tag

```
package chapter7;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;


public class otherwisestatement extends BodyTagSupport {
    public int doStartTag() throws JspException {
        //See if this is enclosed within a case tag
        casestatement caseTag =
                    (casestatement) findAncestorWithClass( this, casestatement.class);
        if (caseTag == null)
           throw new JspException("otherwise tag not enclosed"+
                                  "within case tag") ;
        //See if a when statement was found
        if ( !caseTag.getWhenstatementfound() )
           throw new JspException("otherwise tag found without"+
                                  "when tag(s)") ;
        /*See if a previous when statement was executed.
        Actually, see if a previous when statement has the
        same value as that found on the case statement */
        if ( !caseTag.getWhenstatementvaluefound() )
            return EVAL_BODY_TAG  ;
        else
            return SKIP_BODY;
    }
    public int doAfterBody() throws JspException {
        BodyContent tagBody = getBodyContent() ;
        String tagBodyAsString = tagBody.getString() ;
        try {
            JspWriter out = tagBody.getEnclosingWriter() ;
            out.print( tagBodyAsString ) ;
        } catch (IOException ex) {
            throw new JspTagException(ex.toString());
        }
          return SKIP_BODY ;
    }
}
```

The `doStartTag` method uses `findAncestorWithClass` to communicate with the parent class. The JSP API does not provide a mechanism to communicate directly with siblings. Siblings are tags that have the same parent tag. One procedure for sibling tags to communicate is to access properties of a shared parent class, as is done here.

Implementing a Custom Tag

Throughout the rest of the chapter, we will work in stages to create the following custom tag:

```
<mytaglib:formatLine fontSize="5"
                     fontColor="blue"
                     reverse="true">
This is Line 1
</mytaglib:formatLine>
```

The implementation is shown in different stages. First we'll see how to implement an empty tag without attributes that writes the text "Here's another line" to the page:

```
<mytaglib:formatLine />
```

Then, we'll change the tag components to implement an empty tag with attributes:

```
<mytaglib:formatLine htmlline="This is Line 1" />
```

After adding an attribute, we'll change the tag components to implement the tag with a body:

```
<mytaglib:formatLine>
This is Line 1
</mytaglib:formatLine>
```

Finally, we'll create the tag with the body and three attributes, `fontSize`, `fontColor`, and `reverse`, as shown at the beginning of this section.

Implementing the Empty Tag Without Attributes

As previously mentioned, you need to code three parts: a `taglib` directive that references the `tld` file, the class that implements the tag behavior, and the `tld` file, although not necessarily in that order.

The JSP Page

You might as well start with the JSP page called `example2.jsp` containing the taglib directive and the tag reference. Listing 7-1 shows the page:

Listing 7-1: JSP page example2.jsp containing a custom empty tag

```
<%-- Tell JSP that this page renders HTML --%>
<%@ page contentType="text/html" %>
<html>
<head>
<title>Using a Custom Tag to Generate HTML</title>
```

```
</head>
<body bgcolor="#dddddd">
<%-- Here is the taglib directive --%>
<%@ taglib uri="ch7taglib.tld" prefix="mytaglib" %>
<%-- Here is the tag reference --%>
<mytaglib:formatLine />

</body>
</html>
```

Figure 7-2 below shows the output of the page:
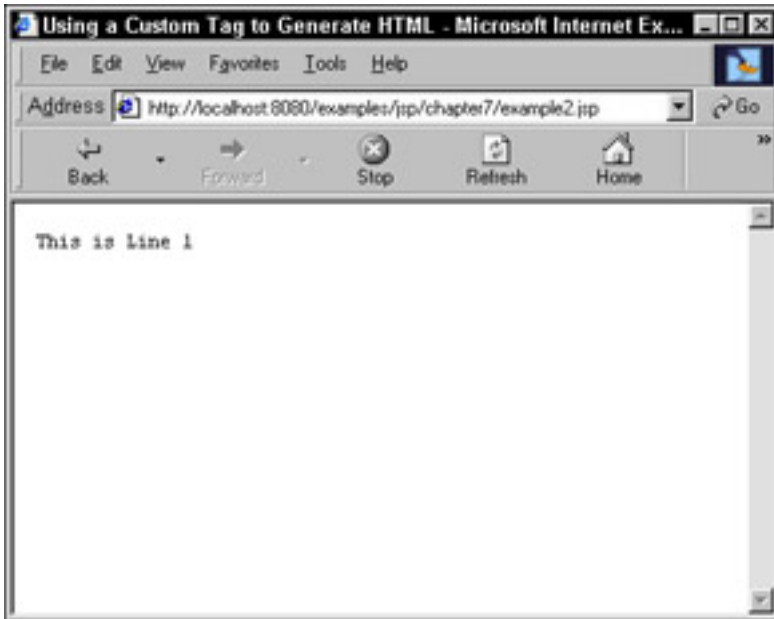
Figure 7-2: JSP output from page in Listing 7-1

The `taglib` directive names the `tld` as the file `ch7taglib.tld`, stored in the same directory as this JSP page. The reference to the `formatLine` tag in the `tld` (Listing 7-3) names the implementing class for this tag as `formatLine`. Even though the tag and the implementing class have the same name in this example, this does not have to be the case. Now let's take a look at the implementation of the `class formatLine`.

The Tag Handler Class: Empty Tag Without Attributes

Listing 7-2 shows an implementation of our custom tag.

Listing 7-2: Implementing the empty tag without attributes

```
package chapter7;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.Writer;
import java.io.IOException;
//or import java.io.* if you prefer
/**
 * Example 1: Write a line of text to the page
 */
public class formatLine extends TagSupport {
    public int doStartTag() throws JspException {
        try {
```

```
        JspWriter out= pageContext.getOut() ;
        out.print("This is Line 1") ;
    } catch (IOException ex) {
        throw new JspTagException(ex.toString());
    }
    return SKIP_BODY ;
  }
}
```

Aside from being good programming practice, some servers (Tomcat included) require that your tag implementations be stored in packages. At a minimum, you can code the import statements that follow the package statement whenever you implement a custom tag.

As previously mentioned, you can implement an empty tag by extending the convenience class `TagSupport`. Here, all you need to do is direct the server to take action when the JSP container detects the start tag (`<mytaglib:formatLine />`) by overriding the `doStartTag` method. You could have directed the server to produce output when the JSP container detected the `end` tag by overriding the `doEndTag` method in `TagSupport`, too.

Note   Although you don't code an `end` tag per se when the tag is empty, the server invokes a `doEndTag` method.

The `doStartTag` method throws a `JspException` so you can normally enclose your code within a try/catch block, as shown in Listing 7-2. Here, you're performing a write operation, so you can catch `IOExceptions` and throw a `JspException`.

Output performed by code implementing custom tags is directed to the implicit object `out`, an instance of the specialized writer class named `JspWriter`. The output is merely the text you want to appear in the page.

The `doStartTag` method returns an integer. As mentioned earlier, the tag interface defines four constants that determine the disposition of the tag body. The constant `SKIP_BODY` instruct the server to ignore the body of the tag. Your `doStartTag` method should return `SKIP_BODY` when included in the implementation of an empty tag.

The last component that needs coding is the tag library descriptor file.

The tld that Describes the Empty Tag Without Attributes

Listing 7-3 shows the `tld` that describes the `formatLine` tag.

Listing 7-3: The TLD for the empty tag without attributes

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
PUBLIC "-//SunMicrosystems, Inc.//DTD JSP Tag Library 1.1//EN"
       "http://java.sun.com/dtd/Web-jsptaglibrary_1_1.dtd">

<!-- Tag library descriptor -->

<taglib>

  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>mytaglib</shortname>
  <uri></uri>
  <info>
     An example tag library description file for Chapter 7
  </info>
  <!-- Place Tag information between <tag> tags -->
```

```
  <tag>
    <name>formatLine</name>
    <tagclass>chapter7.formatLine</tagclass>
    <info> Write a hardcoded string to the JSP page </info>
    <bodycontent>empty</bodycontent>
  </tag>
  <!-- Other tag descriptions could follow -->
</taglib>
```

Your `tld` starts with an XML declaration and a `DOCTYPE` statement. As Sun Microsystems releases new versions of JSP, the `PUBLIC` and `SYSTEM` identifiers in the `DOCTYPE` statement will change to reflect the new releases.

Caution   Sun changed the element names in the DTD for tag library descriptors in JSP release 1.2. Table 7-2 shows the new element names corresponding to the elements used in JSP release 1.1. The element names not listed in Table 7-2 are the same for both releases.

Table 7-2: TLD Element Names in JSP Release 1.1 and 1.2

Element Name, R1.1	Element Name, R1.2	Description
tlibversion	tlib-version	Version of your tag library.
jspversion	jsp-version	JSP release.
shortname	short-name	Prefix used in referring to tags within the library. Notice that the short name in the `tld`, `mytaglib`, is used in the `taglib` directive in Listing 7-1.

The `uri` tag names a public URI that points to the `tld`. In the example, you are not using a public `uri`, hence, the `uri` tag has no content.

The `info` tag provides a short description about the tag library.

`Tld`'s that describe tags that contain attributes and bodies contain additional tags, which are covered later in this chapter.

The content of the `tld` is mostly contained within the `<tag>` elements. The elements shown in Listing 7-3 have the following meaning:

- `name` — The name of the tag used in the JSP page. Actually, the content of the name element is only part of the tag name; the actual name of the tag is the prefix coded in the `taglib` directive followed by the value of the name element in the `tld`.

- `tagclass` — The package and class name that contains the implementation of the tag's behavior, or the name of the tag handler class.

- `info` — A short description of the tag.

- `bodycontent` — One of three values: `empty` for empty tags (such as the example), `JSP` for tags that contain JSP statements in the tag body, or `tagdependent` for tags that do not rely on the JSP container for processing.

Every tag in the tag library has an accompanying `<tag>` entry in the `tld`.

To use this tag library, you can place the `tld` file in the same directory as the JSP page. Place the tag handler class in directory `Web-inf/classes/chapter7`. Depending on your operating system, the case of this directory may or may not matter, so check the directory where you JSP page is located to be sure. Now you can invoke your JSP page. If you're using Tomcat, enter this URL in your browser:

```
http://localhost:8080/examples/jsp/chapter7/example2.jsp
```

Refer to the screen shown in Figure 7-2. Now, let's change our tag by adding attributes.

Implementing the Empty Tag with Attributes

Here's the tag you want to implement:

```
<mytaglib:formatLine htmlline="This is Line 1" />
```

Of course, the value of the attribute `htmlline` can be any string.

The JSP page is the same as Listing 7-1 except for the preceding tag reference ; there's no need to show the entire page here. Also, the displayed page is the same. The changes you need to make are in the tag handler class and the `tld`.

The Tag Handler Class: Empty Tag with Attributes

Listing 7-4 shows the code for the tag handler class for the tag shown in the preceding section.

Listing 7-4: Implementing the empty tag with attributes

```java
package chapter7;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.Writer;
import java.io.IOException;
//or import java.io.* if you prefer
/**
 * Example 2: Process the Tag with attributes
 */
public class formatLine extends TagSupport {
    private String htmlline ;
    //Code get and set methods for the attribute
    public void setHtmlline( String aLine ) {
        htmlline = aLine ;
    }
    public String getHtmlline( ) {
        return htmlline ;
    }
    public int doStartTag() throws JspException {
        try {
          JspWriter out= pageContext.getOut() ;
          out.print( getHtmlline() ) ;
        } catch (IOException ex) {
            throw new JspTagException(ex.toString());
        }
        return SKIP_BODY ;
    }
}
```

To get the JSP container to process the attribute coded in the tag, you simply code a pair of `get/set` methods for the

attribute. Then you can use the attribute value while processing the tag methods.

You do not need to code both `get` and `set` methods for your tag attributes. You are required to code a `set` method. Of course, if you want any other class (bean) to access the values of your tag attributes, you *must* code a `get` method because attribute instance variables should have *private* visibility.

You see that the tag handler class implementing the behavior of the tag with attributes is truly a bean. Unlike beans available to JSPs by using the `jsp:useBean` action, tag handlers may use non-empty constructors. You may have to change the `tld` to reference the tag attributes. The next section shows you how.

The tld that Describes the Empty Tag with Attributes

For each attribute coded in the tag, include an `<attribute>` tag in your `tld`. Listing 7-5 shows the tag element in the `tld` for this empty tag with attributes; the rest of the `tld` is the same as that shown in Listing 7-2.

Listing 7-5: Tag library definition entry for an empty tag with attributes

```
<tag>
      <name> formatLine </name>
      <tagclass>chapter7.formatLine</tagclass>
      <info> Format an HTML Line </info>
      <bodycontent>empty</bodycontent>
      <attribute>
          <name>htmlline</name>
          <required>true</required>
          <!-- rtexpvalue is optional -->
          <rtexpvalue>true</rtexpvalue>
      </attribute>
      <!-- other attributes for this tag would follow -->
  </tag>
```

Here's what the additional tags in the `tld` mean:

- `attribute` — Required tag that signals the start of an attribute description.

- `name` — Required tag that is the name of the attribute.

- `required` — Required tag that is true if a value for this attribute is required, false, if not. If you code false for the required tag, you may omit the attribute when you code the tag in your JSP page. However, when you omit the tag, the tag's corresponding `set` method does not get invoked. You would be wise to code an initial value for an optional attribute in your tag handler class.

- `rtexpvalue` — Optional attribute that is true if the value of the attribute *may* be a Java runtime expression, false if not (value must be a fixed string). For example:
  ```
  <apref:atag attr="<%= new java.util.Date() %>" />
  ```

  If `atag` has the `rtexpvalue` element set to true, the expression is evaluated and used as the value returned by the tag handler's `getAtag` method; if false, the string is used as the value returned. False is the default value so you can safely omit coding this tag in your `tld` when the attribute value is a fixed string.

So far, you've seen how to code custom empty tags with and without attributes. Next, you can read how to code custom tags that contain a body.

Implementing the Tag with a Body

Here's the tag you want to implement:

```
<mytaglib:formatLine>
This is Line 1
</mytaglib:formatLine>
```

You've taken the line of text, formerly an attribute, and placed it in the body of the tag. There's no change to the JSP `taglib` directive, so let's get to the tag handler class for the preceding tag.

[Listing 7-6](#) shows the code for the tag handler class for the tag shown in the preceding section. The presence or absence of attributes has nothing to do with the coding constructs required to process the tag body. First, let's look at the required code:

Listing 7-6: Tag handler for a tag with a body

```
package chapter7;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.Writer;
import java.io.IOException;
//or import java.io.* if you prefer
/**
 * Example 3: Process the tag body
 */
public class formatLine extends TagSupport {
    public int doStartTag() {
        return EVAL_BODY_INCLUDE ;
    }
}
```

That's it! The code states that when the JSP container encounters the start of the tag, the container processes the tag body. If the tag body contains JSP expressions, the expressions are evaluated.

Notice that the inclusion of a tag body in your JSP page is no guarantee that the body is processed. The decision whether or not to process the tag body is made by the tag handler class, not the tag reference or the `tld`. If you code the `doStartTag` method as follows, your tag body won't be processed.

```
public int doStartTag() {
    return SKIP_BODY ;
}
```

And you cannot expect the following code to work because by the time the JSP container gets to the end tag, it's too late to process the tag body.

```
public class formatLine extends TagSupport {
    public int doEndTag() {
        return EVAL_BODY_INCLUDE ;
    }
}
```

Speaking of the `doEndTag` method, when would you ever code this method? The [next section](#) provides some answers.

The Tag Handler Class: Coding a doEndTag Method

When a tag is empty, you rarely take action at the end of this tag. When a tag has a body, you can find yourself coding `doEndTag` methods from time to time. Imagine your tag handler performing some output at the start of a tag, in a `doStartTag` method, such as writing HTML for an HTML table to the implicit `out` object. The tag body contains JSP and

static text that gets written to `out`. You could use a `doEndTag` method to complete the HTML for the table. Your `doEndTag` method could resemble the following:

```
public int doEndTag() {
    JspWriter out = pageContent.getOut() ;
    //Several out.print statements that finish the table
    return EVAL_PAGE;
}
```

The preceding example returned a constant instructing the JSP container to continue processing the remainder of the JSP page. At times, you may want to skip the remainder of the page. Imagine your JSP page is laid out such that if a certain condition arises after you process the tag body, you do not want to process the remainder of the JSP page. For instance, a condition may arise when a registered user logs onto a system and the remainder of the page shows user information; but if that user is not registered, no information is displayed and there's no need to process the remainder of the page. Your `doEndTag` method could resemble the following:

```
public int doEndTag() {
    if (conditionFromEvalTagMeansSkipRestofPage)
        return SKIP_PAGE;
    else
        return EVAL_PAGE ;
}
```

Now you know how to conditionally process the remainder of the page. Next, you can see how to code the `tld` to describe a custom tag with a body.

The tld That Dwescribes the Tag with a Body

To code the `tld` to describe a custom tag with a body, you need make only a *single* change to the `tld`. <u>Listing 7-7</u> shows the `tld` tag entry for a tag with a body.

Listing 7-7: Tag library descriptor entry for a tag with a body

```
<tag>
      <name> formatLine </name>
      <tagclass>chapter7.formatLine</tagclass>
      <info> Write Body of Text to Page </info>
      <bodycontent>JSP</bodycontent>
      <attribute>
         <!-- Describe an attribute here -- >
      </attribute>

   </tag>
```

The only change you need to make is to the value of the `bodycontent` tag, from `empty` to `JSP`.

Note The `bodycontent` tag can use a value of `tagdependent`, which causes your tag to interpret the tag body as non-JSP.

Optionally Processing the Tag Body

To this point, you've been shown implementations of tags that are empty, that omit the tag body, or always include the tag body. If you code a `doStartMethod` according to the following template, you can optionally include or skip the tag body.

```
public int doStartTag() {
```

```
   if (youWantToProcessTagBody)
       return EVAL_BODY_INCLUDE ;
   else
       return SKIP_BODY ;
}
```

In the section on coding the `doEndTag` method, you encountered a similar situation in which, based on a condition, the method returned a constant, generated during tag body processing that directed processing of the remainder of the page. Because this discussion surrounds whether or not to process the tag body, you cannot use a condition generated during body processing; you may not want to process the tag body. You can assume that the origin of the condition that determines tag body processing must occur before the JSP container encounters the tag. Possible origins of the condition are from a tag attribute that has its value dynamically generated, and processes earlier in the page or earlier in some other page. In general, somewhere before encountering the start of the tag, some property in some object or bean must be set that the `doStartTag` method can access.

Attribute values from the tag can be accessed by invoking the appropriate `get` methods, as can bean properties. If the property is derived from some request time parameter, the `doStartTag` method accesses the implicit object by invoking the `getRequest` method of the `pageContext` object.

What if you want to do something other than include the tag body or skip it? What if you want to selectively process parts of the tag body? The next section discusses how you would affect the tag body contents.

Selectively Processing the Tag Body

Here, you want to implement the following tag:

```
<mytaglib:formatLine fontSize="5"
                     fontColor="blue"
                     reverse="true">
This is Line 1
</mytaglib:formatLine>
```

When the `reverse` attribute has a true value, you want to list the string backwards. You also want to append font tags to get the specified color and font size. Listing 7-8 shows the tag handler that accomplishes the goal.

Listing 7-8: Tag handler for a tag that selectively processes its body

```
package chapter7;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
/**
 * Example 4: Color, Size text and optionally Print backwards
 */
public class formatLine extends BodyTagSupport

{
    private boolean reverse ;
    private String color, fontSize ;

    public void setReverse(boolean rev) {
        reverse = rev;
    }
    public boolean getReverse() {
        return reverse ;
    }
    public void setFontSize( String size ) {
```

```
            fontSize = size;
      }
      public String getFontSize() {
            return fontSize;
      }

      public void setColor( String col ) {
            color = col ;
      }
      public String getColor() {
            return color ;
      }
    public int doAfterBody() throws JspException {
            BodyContent tagBody = getBodyContent() ;
            String tagBodyAsString = tagBody.getString() ;
            try {
            JspWriter out = tagBody.getEnclosingWriter() ;

            if ( getReverse() )
               tagBodyAsString = ((new StringBuffer(tagBodyAsString)).reverse() ).toString() ;

            out.print( "<font color=" + getColor() +
                    "><font size=" + getFontSize() +
                    ">" + tagBodyAsString + "</font></font>" ) ;

            } catch (IOException ex) {
               throw new JspTagException(ex.toString());
         }
         return SKIP_BODY ;
      }
}
```

The `get` and `set` methods for the attributes are as before — nothing new there. The new code constructs are in italic. Let's take a look.

The first new construct is found on the `class` statement:

`public class formatLine extends BodyTagSupport`

Classes that change the tag body *must* implement the `BodyTag` interface or extend the convenience class `BodyTagSupport`. The `BodyTag` interface extends the tag interface so you can continue to code `doStartTag` and `doEndTag` methods if you need to. In this example, you do not need to take action at the start or the end of the tag, hence, you did not code the `doStartTag` and `doEndTag` methods.

For the most part, you would code `doStartTag` and `doEndTag` the same way in classes that extend `BodyTagSupport` as you would in classes that extend `TagSupport`. However, `doStartTag` should return the constant `EVAL_BODY_TAG`, a constant not found in the tag interface.

The name of the method that processes the tag's body is `doAfterBody`. More accurately, the `doAfterBody` method is invoked after the JSP container evaluates any statements or expressions. If you want to perform processing on the tag body *before* any JSP statements are evaluated, code a `doInitBody` method.

Perhaps a table showing the order of method invocation is in order. illustrates the order of method invocation.

Table 7-3: Order of Method Invocation

Method Name	When Executed
doStartTag	The JSP container hits the start tag.
doInitBody	After doStartTag execution, before JSP processing of tag body.
no method	The JSP container processes the tag body.
doAfterBody	After the JSP container processes the tag body.
doEndTag	When the JSP container hits the end tag, after processing the tag body.

The preceding order can be short-circuited by the return values of the methods. In the following section, you learn that if the doAfterBody method returns EVAL_BODY_TAG, the doAfterBody method gets reinvoked after JSP tag body evaluation.

For you to process the tag body in your bean (tag handler), you must have access to the tag body. The method getBodyContent provides such access, returning the tag body as an object of, appropriately, class BodyContent. Class BodyContent is an abstract extension of JspWriter. You operate on the tag body by invoking methods of class BodyContent.

In Listing 7-8, you retrieved the tag body as a String by invoking the getString method. The String returned by getString reflects the JSP processing done on the body.

To write output, you need a reference to the implicit out object. Although you may assume that you can access out by referencing the pageContext object, you can't. While your tag handler class would successfully compile, you would not get any output.

To get a reference to the out object, you must invoke the getEnclosingWriter method. Once you have the reference, you write to the out object as usual.

Class BodyContent includes the getReader method that returns the invoking BodyContent object as an input stream, thereby enabling input stream operations to be performed on tag bodies.

Repetitively Processing the Tag Body

The doAfterBody method should return SKIP_BODY, assuming the tag body is to be evaluated only once. If you want to repetitively process the tag body, your doAfterTag method should return a value of EVAL_BODY_TAG. When doAfterTag returns EVAL_BODY_TAG, the tag body is again processed by the JSP container and is followed by another call to doAfterTag.

For example, if you want the following tag:

```
<mytaglib:repeattag repeat="3">
This is line number
</mytaglib:repeattag>
```

To produce the following lines of output:

```
This is line number 1
This is line number 2
This is line number 3
```

The tag handler shown in Listing 7-9 does the trick.

Listing 7-9: Tag handler for repeating tag body processing

```
public class repeatline extends BodyTagSupport
{
    private int numTimes, numTimesLeft ;
    private String repeat ;

    public void setRepeat( String repString) {
      repeat = repString ;
      try {
            numTimesleft = Integer.parseInt( repString ) ;
      } catch (NumberFormatException nfe) {
            numTimesLeft = 1 ;
      }


  }
  public String getRepeat() {
      return repeat ;
  }
 public int doStartTag() {
  try {
      numTimes = Integer.parseInt( getRepeat() ) ;
      } catch (NumberFormatException nfe) {
            numTimes = 1 ;
      }
      return EVAL_BODY_TAG ;
}
public int doAfterBody() throws JspException {
  if (numtimesleft > 0 ) {
      int idx = numtimes - numtimesleft + 1 ;
      BodyContent tagBody = getBodyContent() ;
      String tagBodyAsString = tagBody.getString() ;
      try {
            JspWriter out = tagBody.getEnclosingWriter() ;
            out.print( tagBodyAsString + " " + idx + "<br>" ) ;
            tagBody.clearBody() ;
            numtimesleft-- ;
    } catch (IOException ex) {
            throw new JspTagException(ex.toString());
      }
      return EVAL_BODY_TAG ;
  }
  else
    return SKIP_BODY ;
 }
}
```

The overall idea is to process the tag body based on the value of an integer named `numTimesLeft`. Each time the tag body is processed, the code decrements `numTimesLeft`. When `numTimesLeft` is zero, the code decides not to process the loop body.

The key to this code is the returned value of `doAfterBody`. When `doAfterBody` returns `EVAL_BODY_TAG`, the JSP engine processes the tag body, and then the server calls `doAfterBody` again. Take note of the following line:

```
tagBody.clearBody() ;
```

The `clearBody` method erases the tag body content associated with the tag body object. If you omit this line of code, your output would be as follows:

```
Here is line 1
```

```
Here is line Here is line 2
Here is line Here is line Here is line 3
```

Each invocation of `doAfterBody` picks up the output from the previous invocations, along with the current contents of the tag body.

Here's another variation of repetitively processing the tag body, using a combination of tags:

```
<mytaglib:repeatline repeat="3" >
<mytaglib:formatLine reverse="false" fontSize="5" color="green">
Here is line
</mytaglib:formatLine>
</mytaglib:repeatline>
```

You can see that one tag is coded within another; the `mytaglib:formatLine` is a *child* tag of `mytaglib.repeatline`.

You do not need to change the tag handlers for the preceding two tags to use the tags as coded. The output for the above pair of tags is the same as the output for the `mytaglib:repeatline` tag with the line of text as the tag body.

The body of the outer tag `mytaglib:repeatline` is the result of processing the inner tag, `mytaglib:formatLine`. When the `doAfterBody` method coded for the outer tag invokes its `getBodyContent` method, the invocation *does not* return the code for the inner tag. The `getBodyContent` method returns the result of processing the inner tag.

You can reverse the order of the parent/child relationship as follows:

```
<mytaglib:formatLine reverse="true" fontSize="5" color="green">
<mytaglib:repeatline repeat="3" >
Here is line
</mytaglib:repeatline>
</mytaglib:formatLine>
```

Here is the generated HTML from this tag combination:

```
<font color=green><font size=5>
>rb<3
enil enil si sihT
>rb<2
enil enil si sihT
>rb<1
enil enil si sihT
</font></font>
```

If you were to test this example, this generated HTML would appear as a single green line in your browser.

Tag Library Classes, Interfaces, and Components

As you might imagine, there's plenty of behind-the-scenes activity to get the JSP container to understand a custom tag. For a custom tag to work you need to:

1. Indicate in the JSP page that it uses custom tags using the `taglib` directive;

2. Tell the JSP container what to do when the it sees the tag by creating a `tag handler` class;

3. Tell the JSP container what class to use when it processes the tag by creating a *tag library descriptor file* for the tag library.

Let's examine these coding constructs one at a time, starting with the `taglib` directive.

Using the taglib Directive

The `taglib` directive has a simple syntax shown below:

```
<%@ taglib uri=where_taglib_descriptor_is prefix=some_prefix %>
```

The `uri` attribute names the location of the tag library descriptor file. You can read about this file later in this chapter. For now, know that the tag library descriptor file does what its name says — it describes the characteristics of the tag library. The `taglib` directive tells your JSP page that the page uses a *library*, not a particular tag. The library might only contain a single tag, or it might contain dozens.

A natural question arises — how do you reference an individual tag in the library in your JSP page? The answer is that you use a *prefix*, identified as the value of the second attribute of the `taglib` directive, with the name of an individual tag in the library. For example, the `taglib` directive shown below identifies all tags prefixed with `mytaglib` to be an individual tag within the tag library described by `formatlib.tld`.

```
<%@ taglib uri="formatlib.tld" prefix=mytaglib %>
```

Here's how such a tag may be referenced in a JSP page:

```
<mytaglib:FormatLine fontSize="5"
                     fontColor="blue"
                     reverse="true">
Here's another line
</mytaglib:FormatLine>
```

In the preceding example, `formatlib.tld` is in the same directory as the JSP page containing the `taglib` directive.

Caution   If the `uri` attribute value is an absolute reference (that is, one that begins with a slash), some servers, such as Tomcat, map that absolute value to a file on the local system. You may be confused if you specify a `uri` value that doesn't exist and notice that your JSP pages may still locate the tag library descriptor file.

Now that you can tell your JSP page that it uses custom tags, you need to tell the JSP container what to do with the tag.

Examining the Tag Interface

You implement the functionality of your tag by coding a *Tag Handler Class*. Your tag handler class implements the `javax.servlet.jsp.tagext.Tag` interface. This interface contains constants and methods that the container invokes during the life cycle of your tag, including methods to perform at the start tag and the end tag.

In practice, your tag handler class does not usually implement the tag interface directly. Instead, you can extend a convenience class of the tag interface named `TagSupport` if your tag is empty. You can also extend `BodyTagSupport` if you want to process the tag's body. This class already implements the `BodyTag` interface, which extends the tag interface.

The tag interface defines four constants that govern the disposition of the tag body. Methods you code that describe the actions that occur when the JSP container encounters your tag should return the appropriate constant. Table 7-1 lists these constants and their meanings.

Table 7-1: Constants in the Tag Interface

Constant	Description
SKIP_BODY	The server should not process the body of the tag.
SKIP_PAGE	The server should not process the remainder of the JSP page.
EVAL_BODY_INCLUDE	The server should evaluate the tag body.
EVAL_PAGE	The server should process the remainder of the JSP page.

Before delving into the details of the preceding — named classes and other classes required to implement a custom tag — let's take a look at the last component required to implement a custom tag: the tag library descriptor file.

Creating a Tag Library Descriptor File

The *tag library descriptor file* is a file in XML format that describes the class that implements the functionality of the custom tags in your JSPs. The tag library descriptor file, or `tld`, contains the names of the tags with additional information.

You can find the official DTD describing the elements and attributes of a `tld` at http://java.sun.com/dtd/Web-jsptaglibrary_1_1.dtd for JSP, release 1.1.

You cannot create a `tld` unless you have a basic understanding of XML. If you haven't read Appendix D yet, read it now.

Next, you can see how to code a `class`, `tld`, and `taglib` directive for a custom tag. Let's start with coding the `tld`.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Using Tags Versus Using JavaBeans

In Chapter 6, you read about using the `jsp:useBean` action to access a bean from a JSP page. However, this action is limited to accessing the bean's exposed properties. Within your bean, your mutator methods influence the values of your bean properties. For example, the JSP code below accesses a bean that generates HTML of a certain font size and color. In addition, the bean has a property that, when true, reverses the text. The entire JSP page does not need to be shown; the relevant code illustrates the point.

```
<jsp:useBean id="exbean"  class="chapter7.ExBean"/>
<jsp:setProperty name="exbean" property="fontSize" value="2" />
<jsp:setProperty name="exbean" property="fontColor" value="red" />
<jsp:setProperty name="exbean" property="reverse" value="false" />
<jsp:setProperty name="exbean" property="htmlline" value="This is <i>Line 1</i>" />
<p>HTML line <br>
<jsp:getProperty name="exbean" property="htmlline" />

<jsp:setProperty name="exbean" property="fontSize" value="5" />
<jsp:setProperty name="exbean" property="fontColor" value="blue" />
<jsp:setProperty name="exbean" property="reverse" value="true" />
<jsp:setProperty name="exbean" property="htmlline" value="Here's another line" />
<p>Another HTML line<br>
<jsp:getProperty name="exbean" property="htmlline" />
```

Here's the code within the bean class `ExBean` that sets the value of bean property `htmlline`. As with the preceding listing for the JSP page, the entire bean doesn't need to be shown because the `set` method for the `htmlline` property suffices.

```
public void setHtmlline( String line1 ) {
  String locFontSize = getFontSize() ;
  String color = getFontColor()
  String linetext = (getReverse())?
    ((new StringBuffer(line1)).reverse()).toString() : line1 ;

  htmlline = "<font size=" + locFontSize + ">" +
            "<font color=" + color + ">" + linetext +
            "</font></font>";
}
```

Figure 7-1 shows the page.
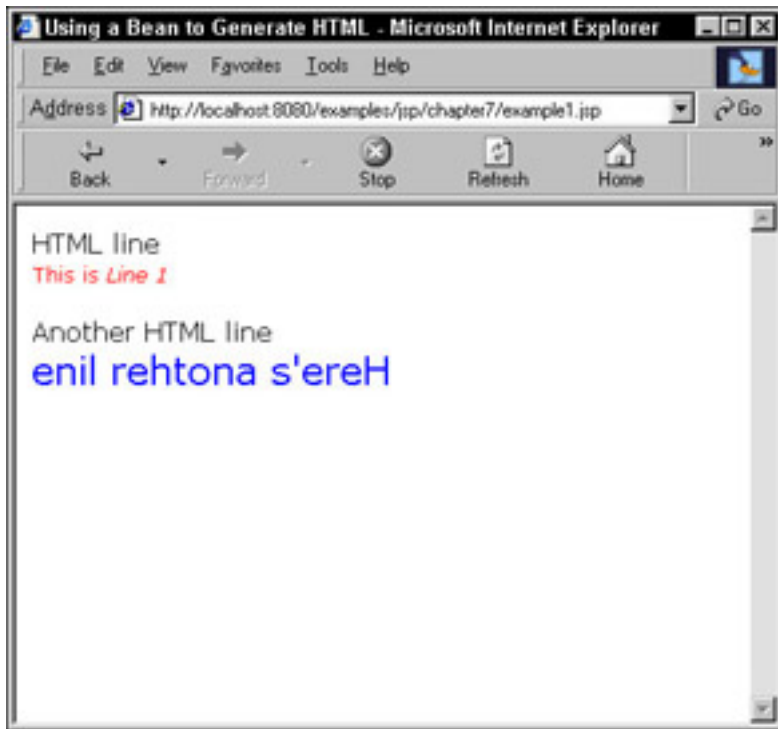
Figure 7-1: Using bean properties to generate HTML

Compare how clumsy and inelegant the preceding example is with the following code:

```
<mytaglib:FormatLine fontSize="2"
                     fontColor="red"
                     reverse="false">
This is <i>Line 1</i>
</mytaglib:FormatLine>
<mytaglib:FormatLine fontSize="5"
                     fontColor="blue"
                     reverse="true">
Here's another line
</mytaglib:FormatLine>
```

Notice how natural the syntax of the custom tag fits with a page of HTML or XML text. By comparison, the JSP code for using the bean seems archaic. Getting, setting, and using bean property values in your JSP pages is a worthwhile and powerful feature. Sadly, using beans and the associated `jsp:setProperty` and `jsp:getProperty` actions clutters your pages with counterintuitive coding structures.

Yes, you can code scriptlets to generate the content shown in Figure 7-1. However, placing the Java code inside your JSP page could blur the distinction between presentation and logic. You would trade using JSP actions with using Java code.

In summary, you can use JavaBeans with the JSP action commands (or scriptlets) to accomplish much of what you can with custom JSP tags. However, coding custom tags in your JSP pages looks more natural than the alternatives.

Next, you can read about the components that constitute a custom tag followed by the code for a simple tag.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Chapter 6: JSP and JavaBeans

You may recall from Chapter 3 that an essential advantage of using JSP over competing technologies is that JSP enables you to separate the business logic from the appearance of your Web pages. You can separate business logic from presentation by using J2SE and J2EE APIs to code your business logic in Java components and by using static and dynamically generated HTML to code your presentation. Java components called *JavaBeans* are particularly important to the JSP author.

This chapter shows you how to use JavaBeans in your JavaServer Pages, starting with an overview of JavaBeans, including how to access data within JavaBeans in your JSP pages. Several examples of accessing JSP pages and changing data within JavaBeans are shown. At the end of this chapter, you can be ready to use JavaBeans in your JSP pages.

A JavaBeans Primer

JavaBeans is a topic worthy of entire tomes, but you do not need to be a JavaBeans expert to use them in your JSP pages. The purpose of this section is to cover enough about JavaBeans to show you how to integrate JavaBeans with your JSP pages.

JavaBeans Defined

Simply defined, JavaBeans are a standard for writing Java software components. As mentioned in Chapter 1, an important virtue of using software components is the ability to plug in components as needed. To accomplish the all-important separation of logic from presentation, it's necessary to discuss how to plug JavaBean components into JSP pages because JavaBeans encapsulate the business logic.

Note   Do not confuse *JavaBeans* with *Enterprise JavaBeans*. Enterprise JavaBeans are JavaBeans with special characteristics that allow them to work with EJB containers. Enterprise JavaBeans are the focus of the second half of this book. Feel free to read ahead in Chapter 11 for a more complete explanation of Enterprise JavaBeans.

Note   Sun provides the *Bean Development Kit*, or BDK, for Java developers interested in creating JavaBeans. You can download a copy at http://java.sun.com/products/javabeans/software/bdk_download.html.

The JavaBean specifies a component architecture for Java classes where a JavaBean is a *public* class that (minimally) has an empty constructor, no public instance variables, and get/set methods for accessing persistent data stored within the bean. In other words, a JavaBean is an object with a well-defined interface. Good Java programmers already code Java classes with hidden instance variables encapsulated with accessor methods. You are halfway there to coding JavaBeans for use in your JSP pages!

Note   You may encounter the term *bean* as you read about JavaBeans, here and throughout Java literature. A bean is an instance of a class created as a JavaBean. In some circles, a bean may mean an Enterprise JavaBean, too. In this book, *bean* refers to an instance of a JavaBean class and *enterprise bean* refers to an instance of an Enterprise JavaBean.

Professional Java development environments use the Java Programming Language's inherent *introspection* feature to peek inside JavaBeans, enabling the Java developer to access and change the properties of JavaBeans by using a GUI. The environments use structures akin to property sheets to get and set bean property values. You must follow a convention, which you may already know, when naming your accessor methods. This convention is explained later in this chapter in the section "Coding JavaBean Property Accessor Methods." It's important to note that a Java developer can use a JavaBean without knowing anything about the bean's internals; the entire state of the bean is described through its properties, which must be accessible through the bean's accessor methods. After all, that's what writing component software is all about.

Another requirement of all JavaBeans is that they implement either the `Serializable` interface or the `Externalizable` interface. This requirement allows a bean to be persistent, an attribute inherent to all objects that implement either of these interfaces. In this book we will only consider the `Serializable` interface.

You may be muttering to yourself, "Great! I have to become a JavaBean maven in order to use JavaBeans in JavaServer Pages." Don't worry! Everything you need to know about using JavaBeans in your JSPs is in the preceding brief definition. If you create a public class with a zero-argument constructor and no public instance variables, you can use this class in your JSP pages. If you want to save persistent data, you can write this data to a database or implement `Serializable`. Rather than coding blocks of scriptlets or methods in your JSPs, you can code JavaBeans containing the scriptlet or method code and invoke the methods from the bean in your JSPs. Now, that's not so bad, is it?

Before you read about how to code your JSPs to use JavaBeans, a few words about coding JavaBeans are in order.

Coding JavaBeans

Basically, coding JavaBeans resembles coding any Java class; remember to code your bean `public`, write (or enable Java to automatically create) a zero-argument constructor, define no `public` instance variables, and use `get` and `set` methods to provide access to your nonpublic instance variables.

You may already know how to code a public Java class and use a zero-argument class constructor. Whether this is the case or not, here's an example of a public class declaration with a zero-argument constructor:

```
public class SomeClassName {
  //Here's the no — arg constructor
  public SomeClassName() { }
  //Rest of the code for this class follows…
}
```

You can give your instance variables any visibility except `public`, although most of the time you may opt for `private` visibility for your instance variables.

Writing a method in a JavaBean is exactly the same as writing a method in any Java class. Code your methods as you would for any Java class, using the familiar Java language constructs you've come to know and love.

Coding JavaBean Property Accessor Methods

As previously mentioned, you *must* follow a naming convention when coding accessor methods to read values from and write values to instance variables. JavaBean tools follow the naming convention when looking inside the bean. These tools then extract the bean properties, enabling the bean user to change the state of the bean by changing the bean properties through a property sheet.

The naming convention details depend on whether or not the instance variable representing a bean property is an array or not. Bean properties represented as arrays are known as *indexed properties*. First, let's consider the case in which bean properties are not represented as arrays.

Coding Accessor Methods for Non-Array Bean Properties

The following is a description of the naming convention for accessor methods of non-array bean properties:

Given a variable named `anInstanceVariable`, declared as follows:

```
private SomeClassOrPrimitiveType anInstanceVariable ;
```

the `get` method that reads the variable's name can be coded as follows:

```
public SomeClassOrPrimitiveType getAnInstanceVariable() {
    return anInstanceVariable ;
}
```

Notice the following about the variable declaration and the `get` method:

- The variable `anInstanceVariable` is declared `private`. The `private` declaration insures that users of your bean cannot access the bean's `instance` variables at will. The bean user has to access the `instance` variables through an approved interface.

- The `get` method is declared `public.` The `get` method is part of the approved interface the bean exposes to the outside world.

- The `get` method takes no arguments.

- The *name* of the `get` method is the word "get" followed by the name of the instance variable with the first letter of the variable name capitalized. However, when the instance variable is of type `boolean`, you may name the method starting with the word *is* instead of the word *get.*

- The `get` method returns some element of the same class or primitive data type as the `instance` variable.

The preceding list of restrictions is required for bean use. Suppose you coded the `getAnInstanceVariable` method without following the rules? For example:

```
public SomeClassOrPrimitiveType
        getAnInstanceVariable( Class1 objClass1 ) {
    return anInstanceVariable ;
}
```

You could invoke the method to "get" the instance variable. However, JavaBean tools would not know the preceding coded method is a `get` method. The difference in signatures between the no-argument `get` method and the method coded above would "fool" the bean tool. Stated differently, the preceding coded method does not follow the *standard* for coding JavaBeans. To the bean tool, the above method is not related to a bean property.

One common mistake is to forget that the first letter of the `instance` variable name included in the `set` method name must be capitalized. Hence, the method header coded as follows fails the naming standard:

```
public SomeClassOrPrimitiveType getanInstanceVariable()
```

When you access bean properties in your JSPs, you *must* code `get` methods according to the convention described previously, or else the JSP engine, as with bean tools, will fail to recognize the method as a `get` method.

In the preceding example, the `get` method merely returns the `instance` variable. Of course, you may code methods

that do all sorts of useful work before returning the element. You are free to use any elements at your disposal to derive a value for the method to return. For example:

```
public SomeClassOrPrimitiveType getAnInstanceVariable() {
    Class1  objectClass1 = new Class1() ;
    SomeClassOrPrimitiveType aVar =
      objectClass1AnotherMethod ( objectClass1,
                                  anotherElement ) ;
    return aVar ;
}
```

In the preceding example a new object is created and another method queried before the final value, `aVar` is returned to the caller.

You may also have `get` methods that don't query an specific instance variable. For example, if you have a bean with the `boolean` property `networkUp`, the method `isNetworkUp` would query the network and return a result based on the status of the network.

In practice, many `get` methods just return the current value of the `instance` variable. Much of the manipulation of setting `instance` variable values is done in "set" methods. Set, or mutator, methods enable a bean user to change the value of a bean property. Here's an example modeled after the `get` method above:

```
public void setAnInstanceVariable(SomeClassOrPrimitiveType aVar) {
    anInstanceVariable  = aVar ;
}
```

Notice the following about the `set` method:

- The `set` method is declared `public`.

- The `set` method returns `void`.

- The `set` method takes one argument of the same class or primitive type as the `instance` variable.

- The `name` of the `get` method is the word "set" followed by the name of the `instance` variable with the first letter of the variable name capitalized.

Notice that `setAnInstanceVariable, anInstanceVariable` is set to the value of `aVar` which was passed into the method. While it is not a requirement of the `set` method that it contain one or more assignment statements, this is usually the case.

You don't need to write both `get` and `set` methods for a bean property. If your JSP or Java code doesn't need to change a bean property, then you shouldn't create a `set` method for that property.

The following section takes a quick look at coding `accessor` methods for indexed bean properties.

Coding Accessor Methods for Indexed Bean Properties

The basic idea and rationale for coding `get` and `set` methods for indexed properties is the same as those for non-array properties. Needing some mechanism for accessing an array element, consider the following indexed property declaration:

```
private SupportedType[] anArrayOfThisType ;
```

Then the method header would appear as follows:

```
public SupportedType getAnArrayOfThisType( int arrIndex )
```

For example, the method invocation coded below reads the fourth element (or index value 3):

```
aVarOfSupportedType = getAnArrayOfThisType( 3 ) ;
```

The `set` method that writes an indexed property requires two arguments: the array position and the data to be written to the property. Here's the method header for a `set` method for the indexed property declared in the previous code listing:

```
public void setAnArrayOfThisType( int arrIndex,
                                  SupportedType aVar )
```

For example, the method invocation in the following writes some data into the fourth array element:

```
setAnArrayOfThisType( 3, aVarOfSupportedType ) ;
```

Now that you've seen the basic elements that comprise a bean, let's take a look at a simple bean.

Creating a Simple Bean

The code in Listing 6-1 shows a simple bean. Later in the chapter, this bean will be used in a JSP.

Listing 6-1: The CalcBean class

```
1        package cbean ;                        //1
2        public class CalcBean {
3        /**
4           Calculator bean for Chapter 6
5         */
6         private int operand1 = 0 ;            //2
7         private int operand2 = 0 ;
8         private double result = 0 ;
9         private String operation = "" ;
10
11        //No-arg constructor for bean....
12        public CalcBean() {      }            //3
13
14        //Get/Set methods follow
15        public void setOperand1( int op1 ) {     //4
16            operand1 = op1 ;
17        }
18        public void setOperand2( int op2 ) {
19            operand2 = op2 ;
20        }
21        public void setOperation( String oper ) {
22            operation = oper ;
23        }
24        public void setResult( double aResult) {
25            result = aResult ;
26        }
27        public int getOperand1() {            //5
28            return operand1 ;
29        }
30        public int getOperand2() {
31            return operand2 ;
32        }
33        public String getOperation() {
34            return operation;
35        }
36
```

```
37          public double getResult() {
38              return performTheOperation() ; //6
39          }
40          /**
41              Perform the operation.....generate the result
42           */
43          private double performTheOperation( ) {
44               double aResult = 0.0 ;
45
46              if ( operation.equals("+") )
47                  aResult = operand1 + operand2 ;
48              else
49              if ( operation.equals("-") )
50                  aResult = operand1 - operand2 ;
51              else
52              if ( operation.equals("*") )
53                  aResult = operand1 * operand2 ;
54              else
55              if ( operation.equals("/") )
56                  aResult = operand1 / operand2 ;
57
58              return aResult ;
59          }
60      }
```

A cursory examination of reveals its purpose, which is to serve as a simple calculator. `CalcBean` has four properties: two operands, an operation, and the result of the operation applied to the operands. To keep things simple, this bean does not include any code that checks for errors (such as `zerodivide`) or performs any edit checks on property values.

Line `1` shows the bean stored as a package. It is a good Java programming practice to use packages to group related classes together. Here, we have only one class so the use of packages may seem a bit pedantic. Later, you'll see where you should put the `CalcBean` class within the Tomcat server's directory structure.

Line `//2` shows the declarations for the bean properties. Notice that the properties are declared with the `private` visibility modifier. The bean requirement is that bean properties are not declared `public`; you can have declared bean properties as protected or use default package visibility. If our bean has subclasses that need access to the bean's properties, you can make a case for using the `protected` visibility modifier. Of course, you can still declare the properties `private` and use `get/set` methods within the subclasses to gain access to the bean properties. Note that because bean properties are permitted to have initialized values, when this bean is created, its properties are given these initial values.

Line `//3` shows the no-arg constructor. You do not need to explicitly code the empty constructor because, as you recall, Java uses the empty constructor by default. The important point is not to code a constructor that requires arguments.

Line `//4` shows a typical set method and line `//5` shows a typical `get` method. The coding of the `get` and `set` methods follows the convention described in the preceding section. Remember, if you do not code your accessor methods according to the described convention, bean tools and the JSP container recognize the methods as accessor methods.

Line `//6` shows a `get` method that invokes a method to generate the result of the calculation. You could have included the body of the method `performTheOperation` within the `get` method or coded `performTheOperation` in another bean or class.

JavaBeans, as software components, must have the ability to communicate with other components. In the following section, you can read how to code your beans so they can communicate with other beans.

Communicating with Other Beans

The standard Java event delegation mechanism enables you to code JavaBeans to communicate with other objects or beans. Beans that are the source of communication are designated event *sources*, whereas interested beans, ones that care about the activities of source beans, are the event *listeners*.

For example, you may want bean A to know when properties in bean B change. If so, bean B is the event source and bean A is the event listener. The events bean A listens for are *property change events*.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Examining Some JSP Pages

In this section, you'll see a few simple JSP pages and read a brief explanation of the JSP elements and how these elements produce dynamic content.

A Simple JSP Page: Your Name Here and ereH emaN ruoY

This JSP page code in Listing 3-1 requests that the user enter his or her name, after which it invokes a JSP that displays the user's name and the name spelled backward. Listing 3-1 shows an HTML page that contains a reference to a JSP followed by a JSP that generates the HTML containing the entered name and the name in reverse. Listing 3-1 shows the HTML page, `forwardreverse.html`, that requests user input.

Listing 3-1: HTML page requesting a JSP

```
1    <HTML>
2    <HEAD>
3    <TITLE>JSP Page to Display Name, Forward and Reverse</TITLE>
4    </HEAD>
5    <BODY>
6    <P>Enter Your Name Below</P>

7      <FORM METHOD="GET" ACTION="ShowForwardAndReverse.jsp">
8      <INPUT TYPE="TEXT" SIZE="20" NAME="yourname">

9    <INPUT TYPE="SUBMIT">
10   </FORM>
11   </BODY>
12   </HTML>
```

The HTML page above does not contain any JSP. Rather, the `<FORM>` tag on line 7 invokes a JSP named `ShowForwardAndReverse.jsp` when the user clicks the Submit button. If the user enters the name Lou Marco, the `SUBMIT` process encodes the data and passes it as a name-value pair, `yourname=Lou+Marco`, and then passes the following URL to the server: `http://localhost:8080/forwardreverse.html?yourname=Lou+Marco`.

Note   Use the Tomcat Web server, which has a JSP engine. See Appendix C for instructions on installing and configuring the Tomcat server.

Listing 3-2 is the first of two examples that show slightly different ways to code `ShowForwardAndReverse.jsp`.

Listing 3-2: A JSP page that prints name forward and reverse

```
1     <%@ page language="java" %>
2     <%--  JSP Page With Scriptlet --%>
3     <!-- JSP Page With Scriptlet -->
4     <HTML>
5     </HEAD>
6     <TITLE>ShowForwardAndReverse.jsp</TITLE>
7     </HEAD>
8     <BODY>
9     <%-- Here is the JSP scriptlet --%>
10     <% String yourName  = request.getParameter("yourname") ;
11        StringBuffer yourNameReverse   =
12          new StringBuffer(yourName).reverse(  ) ;
13        out.println("<P>You Entered " + yourName ) ;
14        out.println("<P>Your Name Backwards is " + yourNameReverse ) ;
15     %>
16     </BODY>
17     </HTML>
```

The first thing that pops out is that the JSP page contains elements that begin with the characters `<%` and end with the characters `%>`. JSP elements are bracketed with `<%` and `%>`. The JSP engine identifies types of JSP elements with additional characters appended to `<%`. For example, lines 2 and 9, bracketed with `<%--` and `--%>`, are JSP comments. The JSP engine does not include JSP comments during the class file translation and, of course, the JSP engine does not use the JSP comments when compiling the page into a servlet. As an aside, line 3, the HTML comment, is passed to the JSP engine for translation and compilation.

Lines 3 through 8 and 16 and 17 are static HTML, which are passed to the JSP engine for translation and compilation. The resultant servlet writes these lines to the output stream with `out.write` statements.

Line 1 is an example of a JSP *directive.* A JSP directive sets various page parameters that affect the structure and properties of the JSP page. The directive coded on line 2 states that the scripting language used in this JSP page is Java. In the JSP 1.1 and 1.2 specifications, the only defined and required scripting language for the `language` attribute is `java`. However, other JSP implementations support other scripting languages beside Java. Allaire's "Jrun" and Caucho Technology's "Resin" are JSP implementations that support JavaScript and Java as scripting languages.

A few points about coding JSP directives are in order here. Notice that the JSP directive (line 1) is bracketed by `<%@` and `%>`. You may code whitespace between the `@` sign and the directive (`page` in this case). You *cannot* code any whitespace between the attribute and its value (`language="java"` in this case). Chapter 4, "The Elements of a JSP Page," contains more information on JSP directives.

Lines 10 through 15 contain a JSP scriptlet. Scriptlets are pieces of Java code that are inserted into the service method generated by the JSP translator. Scriptlets are sandwiched between `<%` and `%>`. Line 10 shows a Java String declare with a call to method `getParameter` from the predefined *request* object instantiated from class `HttpServletRequest`. As you can read in Chapter 4, JSP pages have access to a set of predefined objects, of which the request object is one. The call to `getParameter` requires a parameter name represented as a string object as an argument. The argument used is the name of the text input box coded on line 2 from Listing 3-1 (`yourname`).

Line 11 shows another string declare, `yourNameReverse`, initialized with a call to `reverse()`, a method from the `StringBuffer` class in the `java.lang` package. You do not need to do anything to have JSPs recognize elements from `java.lang` because JSPs are translated into a Java class and then compiled into a servlet. The compiler that creates the servlet does not need any import statements or special setup to recognize elements from the `java.lang` package. If your JSP needed methods from, say, `java.util.Vector`, you would have to use a JSP directive (the page directive, actually) to generate an import statement that imports the required methods.

Lines 13 and 14 compile to output statements that write the entered name forward and backward. These two lines reference `println` from another JSP predefined object, `out`. As previously mentioned, Chapter 4 has the full scoop

on these JSP predefined objects.

Notice that lines 13 and 14 output an HTML `<P>` tag. Although you can code HTML tags with JSP output statements, you should not do so. It's good design practice to separate the programming logic, provided by coding JSP tags, from page formatting tags. You can read a bit more on the separation of logic from presentation later in this chapter.

Listing 3-3 shows the generated HTML resulting from the JSP page in Listing 3-2.

Listing 3-3: The HTML generated by the JSP page

```
1      <!--  JSP Page With Scriptlet -->
2      <HTML>
3      </HEAD>
4      <TITLE>ShowForwardAndReverse.jsp</TITLE>
5      </HEAD>
6      <BODY>
7      <P>You Entered Lou Marco
8      <P>Your Name Backwards is ocraM uoL
9      </BODY>
10     </HTML>
```

Listing 3-4 shows another variation of the JSP page that can generate the HTML shown in Listing 3-3.

Listing 3-4: A variation of the forward and backward JSP page

```
1      <%@ page language="java" %>
2      <%--  JSP Page With a Declaration and Expressions --%>
3      <HTML>
4      </HEAD>
5      <TITLE>ShowForwardAndReverse.jsp</TITLE>
6      </HEAD>
7      <BODY>
8      <%-- Here is a JSP declaration --%>
9      <%!  String yourName  = request.getParameter("yourname") ; %>
10     <%-- Here are some JSP expressions %>
11     <P>You Entered <%= yourName   %>
<P><%= "Your Name Backwards is " +
12      new StringBuffer(yourName).reverse() %>
13     </BODY>
14     </HTML>
```

Line 2 of Listing 3-4 is a JSP comment and line 1 is the same JSP directive as coded in Listing 3-2. Again, lines 3 through 7 and 13 and 14 are straight HTML. Line 9 is an example of a JSP *declaration.* JSP declarations are bracketed with `<%!` and `%>`. You may code complete Java declare statements and complete Java method code in a JSP declaration. Line 9 shows the declaration and initialization of the string variable containing the string entered in the HTML form.

Lines 11 and 12 are examples of JSP *expressions.* A JSP expression is compiled to an output statement that writes a string. The string is formed by evaluating the expression sandwiched between `<@=` and `%>`. JSP expressions use the `toString()` method to convert non-string objects to strings for output, as in the following expression:

```
<%= yourName   %>
```

produces the same output as the scriptlet below:

```
<% out.println( yourName ) ; %>
```

Line 11 shows a JSP expression containing a variable declared with a JSP declare. Line 12 shows some text concatenated with a Java method invocation. The HTML paragraph tags are outside the JSP expression tag markers. JSP may accept HTML tags coded within the tag markers. However, including HTML tags, which are formatting elements, with logic elements within JSP expressions is usually a poor coding practice.

The generated HTML is identical to that shown in <u>Listing 3-3</u>; no need to be repetitious.

Now that you've seen a few simple JSP pages, you might wonder about competing technologies. How does JSP stack up against the competition? What features does JSP have that the competition doesn't? What are the advantages and disadvantages of using JSP? The above questions are addressed in the <u>next section</u>.

Chapter 4: The Elements of a JSP Page

In the previous chapter, you explored some simple JSP pages. In this chapter, you learn about all the elements of a JSP page and you see coding examples using these elements. After you read this chapter, you'll have an understanding of how to code all the JSP elements mentioned.

Coding Static Page Content in JSP Pages

You've read that a JSP page is static text, often in the form of HTML or XML tags, combined with programming elements responsible for generating dynamic content. Often, a sizeable chunk of your JSP page is static. To use such static text in your JSP, you code the static text "as is" into the page, using whatever syntax rules apply to the class of text. If your static text is HTML, for example, you code the HTML as you always would, using the known and familiar syntax rules for proper HTML formation.

The static text you code in the JSP page ends up as Java `println` statements in the generated servlet. The generated servlet eventually passes the text back to the client, where the browser displays the text together with generated dynamic page content. Typing text in a JSP page is a plus compared with writing `println` statements to generate the text in a servlet.

Before you leave the subject of including invariant text in your JSP pages, know that HTML or XML comments will pass through unchanged like any other static text and display in the browser, but JSP comments will not.

The remainder of this chapter describes the programmable elements of a JSP — those elements responsible for generating dynamic content.

Top

Prev   Next

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

JSP is useful in Web application programming because it allows us to create dynamic Web pages that leverage the full power of the Java programming language by mixing static HTML and JSP tags. JSP pages are created in a text editor and are interpreted by a JSP container. The JSP container translates JSP pages into Java class files, which are compiled into servlets. JSP offers distinct advantages over competing technologies but is not without its shortcomings.

In this chapter we've seen our first JSP page examples. In the following chapter we'll break down the parts of a JSP page and examine them each in turn.

Programmable JSP Elements

Programmable JSP elements are divided into five categories: directives, scripting elements, declarations, expressions, and actions. Let's take a look at the elements in the first category.

JSP Directives

JSP directives are JSP elements that send messages to the JSP container. Directives affect the overall structure of the servlet generated from the JSP page. JSP directives do not produce any output to the generated document.

The general format of a JSP directive is:

```
<%@ directiveType attributelist %>
```

The first word, `directiveType`, is one of three values: `page`, `include`, and `taglib`. The second word, `attributelist`, is one or more name-value pairs; the *name* part is the name of the attribute relevant to the directive type and the *value* is a quoted string relevant to the attribute name. If the directive contains more than one name-value pair, the distinct pairs are delimited by one or more spaces, such as the following JSP directive:

```
<%@ page buffer="8k" language="java" %>
```

Note   The syntax for name-value pairs is the same as the syntax for XML name-value pairs. See Appendix D, "XML Overview," for an overview of XML syntax.

Pay heed to the "at" (@) sign after the `<%` at the start of the directive; pay equal attention to the *absence* of the "at" sign at the end of the directive. Also note that the space between the start of the directive and the `directiveType` is *not required.*

The page Directive

The `page` directive enables you to communicate a number of important attributes to the generated servlet. Using the `page`

directive, you can direct the servlet to import classes, define a general error reporting page, or set a content type. The `page` directive follows the general form shown below:

```
<%@ page attributelist %>
```

You can code one or more `page` directives in your JSP page. However, with one exception, take care to ensure that you code only one *name-value pair* per page. The exception is that you may code more than one `import` attribute, the use of which is explained shortly.

You may code page directives anywhere in your JSP page. By convention, `page` directives are coded at the top of the JSP page.

JSP also permits a coding style that follows XML syntax rules. The following is an example of a page directive coded with the XML style:

```
<jsp:directive.page buffer="8k" />
```

The preceding page directive is the same as the example shown here:

```
<%@ page buffer="8k" %>
```

Table 4-1 lists the allowable list of attributes with a short description of each.

Table 4-1: Allowable Page Directive Attributes

Page Directive Attributes	Short Description
`buffer`	Specifies a buffering model for the output stream
`autoFlush`	Controls the behavior of the servlet output buffer
`contentType`	Defines the character encoding scheme
`errorPage`	Defines the URL of another JSP that reports on Java unchecked runtime exceptions
`isErrorPage`	Indicates if this JSP page is a URL specified by another JSP page's `errorPage` attribute
`extends`	Specifies a superclass that the generated servlet must extend
`import`	Specifies a list of packages or classes for use in the JSP as the Java `import` statement does for Java classes
`info`	Defines a string that can be accessed with the servlet's `getServletInfo()` method
`isThreadSafe`	Defines the threading model for the generated servlet
`language`	Defines the programming language used in the JSP page.
`session`	Specifies whether or not the JSP page participates in HTTP sessions

Now let's explore the attributes listed in Table 4-1 in more detail.

The buffer Attribute

The `buffer` attribute specifies buffering characteristics for the server output response object. You'd want a buffer for efficiency because buffered output is usually quicker than unbuffered output. Writing to the browser requires resources. The server sends buffered output in large blocks a few times, which requires less resources than sending unbuffered output in small blocks more often. You may code a value of "none" to specify no buffering so that all servlet output is immediately directed to the response object. You may code a *maximum* buffer size in kilobytes, which directs the servlet to write to the buffer before writing to the response object. Here are a few coding examples:

To direct the servlet to write output directly to the response output object, use the following:

```
<%@ page buffer="none" %>
```

Use the following to direct the servlet to write output to a buffer of size *not less than* 8 kilobytes:

```
<%@ page buffer="8kb" %>
```

The exact size depends on the server. The behavior of the buffer when full is indicated by the value of the `autoFlush` attribute, which is described in the [next section](). The default value of the `buffer` attribute depends on the server implementation.

The autoFlush Attribute

The `autoFlush` attribute controls the behavior of the buffer or, more specifically, what happens to the output buffer when the JSP is buffered and the buffer is full (see the `buffer` attribute above). The `autoFlush` attribute takes a Boolean as a value. Some examples accompanied by a brief description of the directive follow.

The following directive causes the servlet to throw an exception when the servlet's output buffer is full:

```
<%@ page autoFlush="false">
```

This directive causes the servlet to flush the output buffer when full:

```
<%@ page autoFlush="true">
```

Usually, the `buffer` and `autoFlush` attributes are coded on a single page directive as follows:

```
<%@ page buffer="16kb" autoflush="true" %>
```

The preceding directive establishes an output buffer not greater than 16 kilobytes that is flushed automatically when full.

This directive generates an *error.* It doesn't make sense to code the `autoFlush` attribute without a buffer:

```
<%@ page buffer="none" autoflush="false" %>
```

The default value of the `autoFlush` attribute is true.

The contentType Attribute

The `contentType` attribute sets the character encoding for the JSP page and for the generated response page. Put differently, the `contentType` attribute tells the browser how to render the generated page. You can code a `mime type` or a `mime type` and `charset`. Some examples follow.

The following statement directs the browser to render the generated page as HTML:

```
<%@ page contentType="text/html" %>
```

The following statement directs the browser to render the generated page as plain text:

```
<%@ page contentType="text/plain" %>
```

Note   Internet Explorer seems to ignore the `contentType` value of `text/plain`. If your JSP page contains HTML, Internet Explorer will render the page as HTML.

The following directive sets the content type as a Microsoft Word document. Listing 4-1 is an example of a JSP page that specifies a `contentType` of Microsoft Word.

```
<%@ page contentType="application/msword" %>
```

Listing 4-1: JSP page specifying page content is a Microsoft Word document

```
<%@ page contentType="application/msword" %>
<html>
<head>
<title>Showing the contentType Attribute</title>
</head>

<body>
Here's some<B> Bolded Text </b>followed by some
<i>Italicized Text</i>.

<P>Here's some more text
</body>

</html>
```

Displaying this page in Internet Explorer gives rise to the File Download prompt, shown in Figure 4-1.
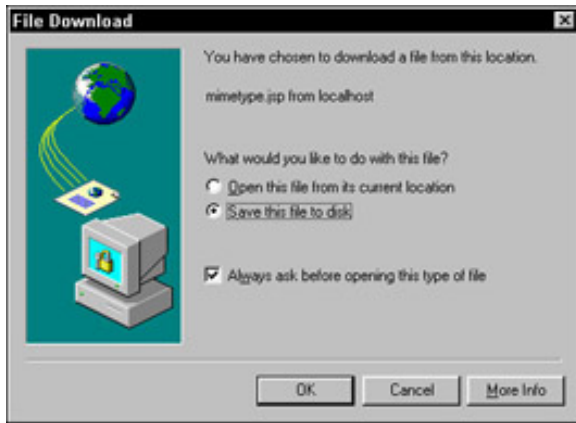
Figure 4-1: Internet Explorer asks you what to do with this MIME type.

Were you to open the file from its present location, Internet Explorer would display the file as a Word document. Were you to save the file, Windows would save the file as a Word document. Netscape would behave in a similar fashion.

You can set a character set for encoding as the following example illustrates:

```
<%@ page contentType="TYPE=text/plain;CHARSET=ISO-8859-1" %>
```

Set the type to plain text using the character set `ISO-8859-1`.

The errorPage Attribute

The `errorPage` attribute defines another JSP page as one that handles unchecked runtime exceptions. The value of the `errorPage` attribute is a *relative* URL.

Note   Relative URLs, when coded with `/` as the first character (called a context-relative path), are referenced from the *application;* when coded without `/` as the first character, they are referenced from the *JSP page.* You cannot code an absolute URL reference in JSP pages.

For example, the following directive displays `MyErrorPage.jsp` when all uncaught exceptions are thrown:

```
<%@ page errorPage="MyErrorPage.jsp" %>
```

`MyErrorPage.jsp` is stored in the *same* directory as the page containing the above page directive. If you code the directive as follows:

```
<%@ page errorPage="/MyErrorPage.jsp" %>
```

then MyErrorPage is stored in the root directory of the *application*.

You can use the `setAttribute()` method of class `javax.servlet.jsp.ServletException` to pass the exception object

to the error page. Since JSP error pages are meant to inform on *uncaught* exceptions, you'll use error pages to report on such exceptions rather than attempt recovery.

The JSP page `MyErrorPage.jsp` must have a page directive with the `isErrorPage` attribute set to true.

The isErrorPage Attribute

The `isErrorPage` attribute indicates whether or not the JSP page is the URL coded in another JSP page's `errorPage` attribute. The value of `isErrorPage` is either true or false. The default value of the `isErrorPage` attribute is false.

The extends Attribute

The `extends` attribute specifies a superclass that the generated servlet must extend. For example, the following directive directs the JSP translator to generate the servlet such that the servlet extends `somePackage.SomeClass`:

```
<%@ page extends="somePackage.SomeClass" %>
```

Do you see a potential problem with coding a value for the `extends` attribute? The server may be using a superclass already, or may require that generated servlets extend another, *different* superclass than the superclass coded as a value for the `extends` attribute. If you use the `extends` attribute, be careful.

The import Attribute

The `import` attribute serves the same function as, and behaves like, the Java import statement. Classes coded as values for the `import` attribute are made known to the generated servlet.

If you do not code any page directives with `import` attributes, the generated servlet imports, *at a minimum,* the following classes:

- `java.lang.*`

- `javax.servlet.*`

- `javax.servlet.jsp.*`

- `javax.servlet.http.*`

Different servers may import additional classes, but the preceding list of classes is required to have a functioning JSP environment.

You should not rely on your JSP pages having access to classes other than the ones listed *without importing the classes yourself!* Wouldn't you be embarrassed if your carefully coded JSP pages executed perfectly on server A but choked big time when the company switched to server B, a different Java-enabled server?

The following example allows you to code unqualified references in a JSP page for classes in `somePackage`:

```
<%@ page import="somePackage.*" %>
```

The `import` attribute is the *only* attribute that may be coded multiple times in a JSP.

Be advised that different Web servers use different directory structures. Therefore, in all likelihood, server A may force you to store your custom classes in a different directory than server B. Some servers require that you store classes used by your JSP pages in a different directory than the directory used to store custom classes for your servlets. The moral of the story is that a cursory glance at the server documentation is worth an hour of directory code examination and trial and error.

The info Attribute

The `info` attribute enables you to make a string available to your JSP pages by invoking the JSP page implementation of the `Servlet.getServletInfo()` method. The string can be pretty much anything you desire. The following is a coding example:

```
<%@ page info="This JSP Page Written By Lou Marco" %>
```

The isThreadSafe Attribute

The `isThreadSafe` attribute lets the JSP container know how to dispatch requests to the page. The value of this attribute is a `boolean`. When the value is false, the JSP container dispatches one request at a time; when the value is true, the JSP container dispatches all outstanding requests simultaneously.

Servlets usually create a thread per user request. Multiple requests result in the servlet dispatching multiple threads, each thread accessing the `service()` method of the same servlet. The underlying assumption is that the servlet is *thread safe.* The servlet synchronizes access to data so that threads do not "step on" each other.

Assigning a value of true to `isThreadSafe` does not make your code thread safe. The `isThreadSafe` attribute is merely a statement about your code's ability to handle multiple threads. Although you should write your code to assume correct execution in a multithreaded environment, you may encounter a situation in which a class you need to use is not thread safe. Hence, you may, at times, code a value of false for the `isThreadSave` attribute. The default value is true.

The language Attribute

The `language` attribute indicates the programming language used in scripting the JSP page. The JSP specification requires that a JSP implementation support a value of `java` for the `language` attribute. However, other JSP implementations may, and do, support other values for the `language` attribute.

The session Attribute

The `session` attribute indicates whether or not the JSP page uses HTTP sessions. A value of true means that the JSP page has access to a `builtin` object called *session;* a value of false means that the JSP page cannot access the `builtin` session object. Put another way, the following directive allows the JSP page to use any of the `builtin` object session methods, such as `session.getCreationTime()` or `session.getLastAccessTime()`:

```
<%@ page session="true" %>
```

When the `session` attribute has a value of false, any attempt to access the `builtin` object session causes an error during JSP to servlet translation.

In [Chapter 5](#), "JSP Web Sessions," sessions are covered in detail. You should recall that an advantage of using servlets over traditional CGI is that servlets allow for sessions that maintain information about the client across multiple Web pages whereas CGI does not.

To wrap up the plethora of attributes for the page directive, I want to note again that these attributes do not direct any output to the eventually displayed page. You may code multiple page directives in your JSP page, but, with the exception of the `import` attribute, attributes can appear *at most once in the page.* In addition, you may code more than one attribute in a single page directive, with some combinations (such as `buffer="none"` and `autoFlush="false"`) not permitted.

JSP has two other directives: the `include` and the `taglib` directives, which are covered in the following two sections.

The include Directive

You probably have an idea of the purpose of the `include` directive. The `include` directive enables you to bring external code into your JSP at the point of reference. The syntax is as follows:

```
<%@ include file="relativeURL" %>
```

Unlike page directives, the placement of the `include` directive is critical. The JSP translator copies the code stored at `relativeURL` into your JSP page starting at the location of the `include` directive. The URL specification in the value of the

`file` attribute is a *relative* URL.

A few caveats concerning the use of the `include` directive are in order. First, be aware that the file to be included may (and usually does) contain JSP code. If the to-be-included file contains page directives, the restriction about the multiple occurrences of page directive attributes applies. In other words, you cannot have an included file contain a page directive with an attribute already coded in the JSP page (except the `import` attribute).

Another caveat is that when you change an included file, you must update *all* the JSP files that include the changed file. What you need is a JSP or server command that tells the JSP translator to retranslate, but such a command is not currently available. The JSP translator detects when JSP pages require translation based on modification dates. An admittedly primitive but effective workaround is to change a comment in your pages that uses included files to force a retranslation.

Later in this chapter, in the section titled "Coding JSP Standard Actions," you'll read about the `<jsp:include` action that also brings external code into your JSP pages.

The taglib Directive

In Chapter 7, "JSP Tag Extensions," we explore using custom tags in JSP pages. Without giving too much away before then, know that a custom tag is part of a *tag library.* A tag library is a set of user-defined tags that implement custom behavior. In short, you are extending the functionality of JSP pages by creating a set of tags that implement new behaviors for your JSP pages. While this sounds a bit like using JavaBeans in your pages (see Chapter 6, "JSP, JavaBeans, and JDBC," for JavaBean usage in JSP pages), the tag library usage plays a slightly different role than JavaBean usage does. Because of the scope of the topic, you'll have to wait until you read Chapters 6 and 7 for the full story.

Given that you have a "thing" called a tag library for use in your JSP page, you have to tell the JSP container about the library. You do so by using a JSP `taglib` directive.

The `taglib` directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page. Here's an example of a `taglib` directive:

```
<%@ taglib uri=http://joestags.tld prefix="joe" %>
```

Here, the `uri` attribute provides an *absolute* `uri` containing the code that implements the custom tag's behaviors. The value of the `uri` attribute can be an absolute (shown above) or relative reference. The `prefix` attribute associates the custom tag coded in the JSP page with the library name coded as a value of the `uri` attribute.

Note   The term `uri` means *Universal Resource Identifier*. Think of a `uri` as a URL or a file reference.

The following is an example of a custom tag from `http://joestags.tld` referenced by the value of the `prefix` attribute coded in the `taglib` directive:

```
<joe:doMyCalc />
```

You *must* code the `taglib` directive before coding any references to the custom tags from the tag library.

Well, that's the story on coding JSP directives. Worth repeating is that JSP directives do not produce output per se; JSP directives communicate certain parameters and set certain attributes that affect the generated servlet and resultant output page.

What about JSP commands that *do* result in output? You'll read about these commands next, starting with JSP scripting elements.

JSP Scripting Elements

JSP scripting elements enable you to insert code into the JSP page that results in code in the generated servlet. Scripting elements range from one-sentence variable declares to entire Java methods. I examine three categories of scripting elements: *expressions, declarations,* and *scriptlets.*

JSP Expressions

A JSP expression inserts data in the resultant output page. A JSP expression has the following syntax:

```
<%= javaExpression %>
```

The JSP translator evaluates `javaExpression`, converts `javaExpression` into a string, and places the resultant string directly in the output page. If the expression cannot be converted to a string, the runtime throws a `ClassCastException`. For example, the following JSP expression generates the date and time the JSP page was requested:

```
<%= new java.util.Date() %>
```

Or, using a page directive that allows for an unqualified reference to the `Date` class with the expression:

```
<%@ page import="java.util.*" %>
<@= new Date() %>
```

JSP expressions are evaluated at the time the JSP page is *requested* (by entering the name of a JSP page in the location bar of a browser or clicking a hyperlink that references the JSP page), or at *runtime,* not at JSP translation time. The result of the evaluation being done at runtime is that variables or objects referenced in the expression have access to any information about the request.

You may use the XML form of a JSP expressions as follows:

```
<jsp:expression>
    Java expression
```

```
</jsp:expression>
```

The engineers at Sun Microsystems make life a bit easier for the JSP programmer by providing the programmer access to predefined environment objects called *implicit objects.* These objects are accessible from JSP expressions and JSP scriptlets. You read about these objects later in this chapter.

JSP Declarations

As with any programming language, JSP uses variables to hold program data or code that performs various tasks. One JSP feature that makes variables or program code known to the JSP page is called a JSP *declaration.*

JSP declarations have the following format:

```
<@! JspDeclaration @>
```

In <u>Listing 4-2</u> you see a simple page with a JSP declaration and expression.

Listing 4-2: Simple JSP page with a declaration and expression

```
<%@ page contentType="text/html" %>
<html>
<head>
<title>Simple JSP Declaration Example</title>
</head>

<body>
<%-- Here's a JSP Declaration --%>
<%! int counter = 0 ; %>
<P>This page has been accessed <b>
<%-- Here's a JSP Expression --%>
<%= ++counter %></b> times
</body>

</html>
```

JSP declarations *by themselves* do not cause output. Rather, JSP declarations are used with the JSP expressions and scriptlets to cause output. Note that the preceding expression could not have generated a value for counter without the `counter` variable being declared.

Worthy of mention is that instance variables declared in JSP declarations need not be declared static because instance variables are shared among separate page requests.

JSP declarations may include *entire methods.* Listing 4-3 shows a small JSP illustrating a method `declare`.

Listing 4-3: Coding and invoking a method in a JSP declaration

```
<%@ page contentType="text/html" %>
<html>
<head>
<title>Coding and Invoking a Method</title>
</head>

<body>
<%-- Here's a variable and method declaration --%>
<%! int toInteger = 100 ;
    int sumOfFirstIntegers = addIntegers( 100 ) ;
    public int addIntegers( int to ) {
      return to * (to + 1) / 2 ;
    }%>
<P> The sum of integers from 1 to <%= toInteger %> is:
<B> <%= sumOfFirstIntegers %> </b>
</body>

</html>
```

Listing 4-3 also includes a few JSP expressions.

As you may have guessed by now, you can code JSP declarations in XML style syntax as follows:

```
<jsp:declaration>
     Java Code
</jsp:declaration>
```

JSPs also permit you to code pieces of Java in your page by coding scriptlets, as explained in the next section.

JSP Scriptlets

A *scriptlet* is an arbitrary piece of Java code. The general format is as follows:

```
<% aPieceOfJavaCode %>
```

Understand that a "piece" of code can be entire statements or groups of statements.

Scriptlets are executed at request time. Hence, code contained in scriptlets may modify objects by invoking methods. Listing 4-4 shows a page similar to Listing 4-3 but using a scriptlet.

Listing 4-4: JSP page with a scriptlet

```
<%@ page contentType="text/html" %>
<html>
<head>
<title>JSP scriptlet example</title>
</head>

<body>
<%-- Here's a JSP scriptlet --%>
<%   int toInteger = 100 ;
```

```
    int sumOfFirstIntegers = toInteger * (toInteger + 1 ) / 2;

    int sumByLoop = 0 ;
    for (int counter = 1; counter <= toInteger; counter++ )
        sumByLoop += counter ;
%>
<P> The sum of integers from 1 to <%= toInteger %> by loop is:
<B> <%= sumByLoop %> </b>
</body>

</html>
```

Notice that Listing 4-4 uses *pieces* of Java code as opposed to an *entire* method. Changing Listing 4-4 to use a declaration causes the JSP translator to generate an error, as shown in Listing 4-5.

Listing 4-5: Tomcat reacts to using a JSP declaration where a scriptlet is called for

```
Error: 500
Location: /examples/jsp/loutest/loutest.jsp
Internal Servlet Error:
org.apache.jasper.JasperException: Unable to compile class for
JSPD:\tomcat32\work\localhost_8080%2Fexamples\_0002fjsp_0002floutest_0002floutest_0002ejsploutest_jsp_0.java:24: Type
expected.
            for (int counter = 1; counter <= toInteger; counter++ )
                ^
1 error
```

The actual diagnostic shows a rather long stack trace, but the included top piece of the diagnostic shown in Listing 4-5 should give you the essential flavor of the problem.

JSP scriptlets do not have to be complete Java statements. You may code pieces as long as all the pieces *together* form complete Java statements. Listing 4-6 illustrates the concept.

Listing 4-6: JSP scriptlets with pieces of Java code

```
<%@ page contentType="text/html" %>
<html>
<head>
<title>JSP scriptlet example</title>
</head>
```

```
<body>
<%-- Here's a JSP scriptlet with pieces of code --%>
<%  int toInteger =
        Integer.parseInt(request.getParameter("to") ) ;
    int sumOfFirstIntegers = toInteger * (toInteger + 1 ) / 2;

    if (sumOfFirstIntegers > 100000){
%>
        <b>Big Number</b>
    <% } else { %>
        <i>Small Number</i>
    <% } %>

</body>

</html>
```

As you can see, the `if` statement is broken into pieces with some pieces as HTML and some as Java code. The requirement for a syntactically correct scriptlet is that all the pieces must form syntactically correct Java code.

The following statement uses an implicit object called `request`:

```
int toInteger = Integer.parseInt(request.getParameter("to") );
```

You pass the parameter by entering the name of the JSP with the parameter entry as shown here:

```
http://localhost:8080/examples/jsp/loutest/loutest.jsp?to=50
```

You can code JSP scriptlets as XML tags, as follows:

```
<jsp:scriptlet>
     Java Code
</jsp:scriptlet>
```

You see that JSP allows you, the JSP programmer, to use Java code in your JSP pages in interesting and flexible ways. However, you have additional capabilities that do not involve embedding Java code in your JSP page. JSP supports a variety of *standard actions,* which are covered in the next section.

Chapter 11: A First Look at EJB

Overview

You've spent some time reading about JavaServer Pages, sometimes referred to as the front door to J2EE
applications. In Part III, you can read about Enterprise JavaBeans. As mentioned in Chapter 1, "Enterprise Computing
Concepts," Enterprise JavaBeans are a server-side software component architecture. In other words, the Enterprise
JavaBeans specification describes how to develop distributed objects and how to deploy these objects in a distributed
computing environment.

This chapter provides an introduction to Enterprise JavaBeans (EJBs). This chapter's first order of business is to
dispense with the belief that Enterprise JavaBeans are related to JavaBeans. Then you can read about the ambitious
goals of the EJB architecture. EJB release 1.1 is discussed here, along with features of EJB release 2.0 (Sun released
the final draft on October 25, 2000).

This chapter also introduces the important topic of *EJB Roles* and how these roles enable the development of
compatible EJBs by different vendors.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Part III: Enterprise JavaBeans

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

You have now seen a complete JSP application. This application has combined JSP pages and JavaBeans to provide
dynamic Web page content. JavaBeans have been used to access database resources and provide client information.
JSP error handling, which we discussed in the previous chapter, was used to handle incorrect user input. You should
now have a better understanding of how these different parts of a JSP application work together. In the coming
chapters we'll learn how to use Enterprise JavaBeans, making our Internet applications even more powerful.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Chapter 12: The Elements of an EJB

Overview

At this point, you've been introduced to the rationale for Enterprise JavaBeans (EJBs), which is the need for server-side distributed components. You've learned about the players in the world of EJBs and about the lofty goals of the EJB architecture. Now, you're ready to learn about the nuts and bolts of an Enterprise JavaBean.

This chapter discusses the components of an EJB. You can read about the required interfaces for implementing and constructing an enterprise bean. You can discover that EJB supports three different bean types — entity beans, session beans, and (new with EJB 2.0) message-driven beans. And you can learn why you need different enterprise bean types in an application. You can also read about the environment required by enterprise beans for living, working, and playing, including how to deploy your beans once developed.

First, let's take a look at the makeup of an enterprise bean followed by a description of the bean's components.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

You've just taken a quick tour of Enterprise JavaBeans. You've learned that EJBs are distributed, server-side software components that live inside an abstraction called a container. You've been introduced to the goals of the EJB architecture. You've learned that the EJB specification provides for three different types of enterprise bean — the session bean, the entity bean and, with release 2.0, the message bean. You've learned that clients do not access enterprise beans directly; rather, they access enterprise beans by going through the bean's home or remote interface. You've read about the deployment descriptor, which enables you to customize an enterprise bean's behavior without changing the bean's source code. You've also been introduced to the concept of EJB roles and how these roles are important to the software component market. This should be enough to let us begin to dig into EJBs in detail with some coding examples in the next chapter.

.

Examining EJB Release 1.1

Sun released Enterprise JavaBeans 2.0 as a final draft in late October 2000. However, this book concentrates on Enterprise JavaBeans 1.1, because, at the time of this writing, no commercially available application servers support the EJB 2.0 specification. In Part III of this book, EJB 2.0 features are discussed where appropriate.

The best place to start when examining the functionality provided by release 1.1 is with the core of EJB — the enterprise bean itself. This chapter also covers other core concepts and key architecture elements that you need in order to understand how enterprise beans work.

Examining the EJB and Its Environment

The EJB specification describes two types of enterprise beans: a session bean and an entity bean. A session bean is a bean that models a process; an entity bean is a bean that models data. A considerable portion of Part III of this book is devoted to describing these bean types.

Note   Release 2.0 of the EJB specification describes a third enterprise bean type — the *message-driven bean* (or, more simply, a message bean). Message beans were specifically designed to handle incoming JMS messages. Chapter 12, "The Elements of an EJB," covers the new message bean in more detail.

Clients do not directly access session or entity beans. Clients (which can be other enterprise beans) usually access enterprise beans by using a *naming service*, such as JNDI. The EJB architecture describes the relevant interfaces used to access enterprise beans, about which more is said throughout this chapter.

Enterprise beans of all types live within EJB containers. The EJB container is not a physical piece of hardware or software but rather an abstract entity that manages the various instances of enterprise bean classes. Most industry EJB containers are packaged as part of a larger software product, like IBM's *WebSphere* and Allaire's *JRun*. The container provides support for contained enterprise beans through a set of interfaces defined in the architecture specification.

Note   Release 1.0 of the EJB specification stated that support for entity beans within EJB containers was *optional*. Since release 1.1, support for entity beans within EJB containers is *mandatory*.

The containers live within an EJB server. You can think of the EJB server as the outermost box containing all the elements of an EJB environment. The server is responsible for providing access to services and resources, such as threads and processes, the network, storage devices, and memory.

Because the client does not directly access enterprise beans, it must access enterprise beans through interfaces. Here's a brief overview on these interfaces.

The *home* interface describes the methods responsible for managing the life of the enterprise bean. The home interface has descriptions for methods that enable a client to create, locate, and destroy EJB instances. The object that implements the home interface for an enterprise bean is referred to as the *Home Object*, or *EJB Home*.

The *remote* interface describes methods that the enterprise bean uses to do its work. Clients can call methods of the remote interface. The object that implements the remote interface is referred to as the *EJB Object*.

Figure 11-1 depicts clients accessing EJB instances by going through, or using, the home and remote interfaces.
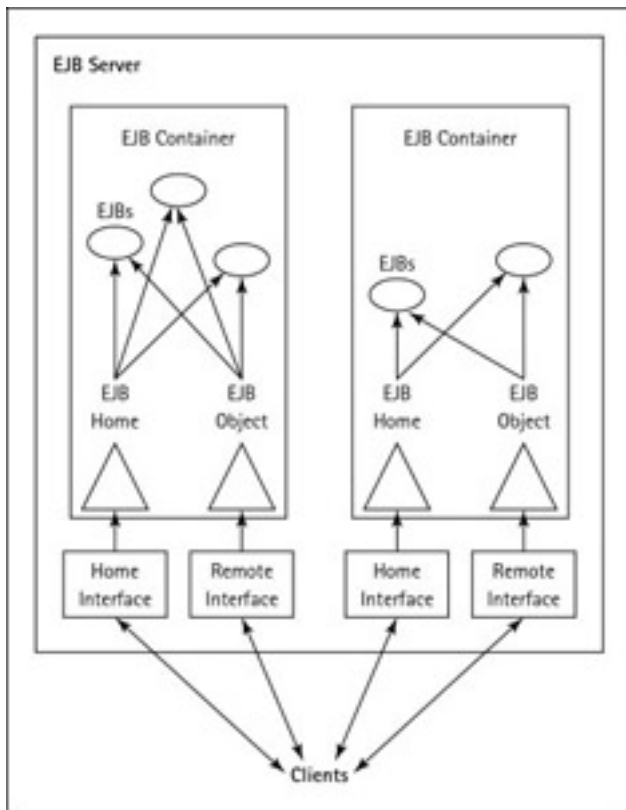
Figure 11-1: Accessing enterprise beans through the home and remote interfaces

The basic idea is that clients, which can be Java applets, Java servlets, or other enterprise beans, can access EJB instances only through the home or remote interfaces. Notice that clients access both bean types through the home and remote interfaces.

The EJB instances are objects from the actual *enterprise bean class*, or bean class. The bean class contains implementations of the business methods, which do the actual work of the bean. However, the bean class does not implement the methods described in the remote interface. To further complicate matters, the bean class methods must have signatures that match those found in the bean's remote interface and several methods found in the bean's home interface. Chapter 12, "The Elements of an EJB," discusses in detail how the home interface, remote interface, and bean class interrelate.

While session, entity, and message beans are different animals, they have much in common. The next section discusses essential enterprise bean properties that are shared by session and entity beans.

Essential Enterprise Bean Properties

The EJB specification describes essential bean properties as follows:

- **Containers manage enterprise bean activities.**

  The EJB container has the responsibility for creating, destroying, and otherwise manipulating the enterprise bean. Various runtime services required by the bean are *not* directly accessible by the bean; these services are provided *to* the bean by the EJB container. Each container may house multiple enterprise beans, and an EJB server may

contain multiple EJB containers.

The EJB architecture *requires* that enterprise beans be deployed in suitable containers. Stated differently, EJBs must live within containers that are capable of providing needed services.

- **Enterprise beans can, and usually are, customized during deployment.**

The most elegantly crafted enterprise bean is useless unless deployable in a customer's environment. Given that enterprise computing environments differ in unknown ways, customers who deploy enterprise beans need an easy way to adapt the bean's properties to fit with the peculiarities of an environment. The architecture calls for enterprise beans to be packaged with a file called a *deployment descriptor*. The deployment descriptor is a text file in XML format that can be edited to suit the needs of the customer. Hence, the customer does not need to have access to Java source code to change the behavior of an enterprise bean to fit his or her environment.

Note   Deployment descriptors are in XML format. Appendix D provides an overview of XML.

- **The client that accesses enterprise beans does not care about the server or container particulars.**

Because a client accesses enterprise beans through the home and remote interfaces, and not directly through the server or container, the client does not need to be concerned with the server or container particulars. The home and remote interfaces are not dependent on server or container particulars above those required by the EJB specification. The upshot is that enterprise beans are highly portable among servers and containers that follow the EJB specification.

Examining EJB Roles

The EJB architecture describes a *role* as a set of responsibilities (often called a contract) assumed by a party in the development and deployment of an enterprise bean. The basic idea is that delineating the responsibilities of each role ensures that the outputs of one party are compatible with the inputs of others.

The six roles described by the EJB release 1.1 specification, and a seventh introduced in EJB release 2.0, are described in the following.

The Enterprise Bean Provider

The enterprise bean provider supplies software components (enterprise beans) that customers may purchase for use in their computing environments. The provider is responsible for supplying the class files that implement one or more processes, the definitions of the bean's home and remote interfaces, and a deployment descriptor that describes how the bean can be adjusted to fit into the customer's environment.

The output of the enterprise bean provider is one or more enterprise beans packaged into *ejb-jar files*, which contain the class files and bean deployment descriptors. You can probably imagine bean providers marketing enterprise beans that perform useful tasks, like processing credit card transactions, which could be plugged into a customer's Internet application as needed.

The Application Assembler

The application assembler combines enterprise beans and possibly other software components and elements into a deployable application. Typically, the application assembler must be adept in the firm's business and with the particulars of how the application's components work together.

The inputs received by the application assembler are the ejb-jar files produced by the enterprise bean provider. The outputs delivered are these ejb-jar files with deployment descriptors modified to reflect the environment's particulars.

The application assembler can be one or more programmers that write glue code that binds together several classes provided by the bean provider. Possibly, the application assembler needs to tweak a user interface to fine-tune a particular process. A good description of the application assembler's ultimate responsibility is to do whatever must be

done to turn a group of EJBs into something deployable.

The Deployer

The deployer performs the actual application deployment of the combined components into a suitable environment. The environment is one or more EJB containers resident on one or more EJB servers. To do its job, the deployer needs inputs from the application assembler, the *EJB container provider,* and the *EJB server provider.*

The output of the deployer is one or more EJBs or an application consisting of one or more EJBs on a specific EJB container. The deployer knows about the details of the EJB containers and EJB servers. Typically, the deployer is knowledgeable of the firm's computing environment.

The EJB Container Provider and the EJB Server Provider

The EJB container provider provides various system-level services available to applications composed of EJBs, such as transaction support, security, storage, and memory.

The EJB server provider supplies the EJB-enabled application server that provides all the system services required by the EJB containers. The EJB architecture does not specify a distinction between the container provider and the server provider. In practice, one organization typically acts as both the container and server provider. In other words, an organization providing EJB-enabled application servers also provides suitable EJB containers.

EJB servers usually contain tools that assist the deployer in deploying EJBs to containers and customizing the enterprise beans if required.

The System Administrator

The system administrator is responsible for managing and maintaining the infrastructure needed for continual operation of the application. The EJB specification does not require any specific inputs from other parties. Typically, the system administrator uses various monitoring and system tuning tools available from the server providers.

The Persistence Manager

The EJB 2.0 specification has introduced a new role — the persistence manager provider. The persistence manager is a new participant in the world of Enterprise JavaBeans dealing with entity beans. A discussion of the persistence manager and the persistence manager provider role can be found in Chapter 15, "EJB Entity Beans."

Wrapping up the Roles Discussion

One party may assume several roles, as is the case with container and server providers. As time goes on, it is easy to envision a marketplace with vendors assuming one or more roles in providing enterprise beans, container, server, and administrative services to customers. When vendors follow the EJB standard, customers can be reasonably sure that different vendor products can operate well with each other.

In addition, the EJB specification may further delineate responsibilities, either by providing additional roles (the persistence manager role in release 2.0, for example) or by defining distinctions between container and server providers.

In short, the purpose of defining the above roles is to define areas of responsibilities to ensure the production of quality, portable enterprise beans that adhere to the EJB specification.
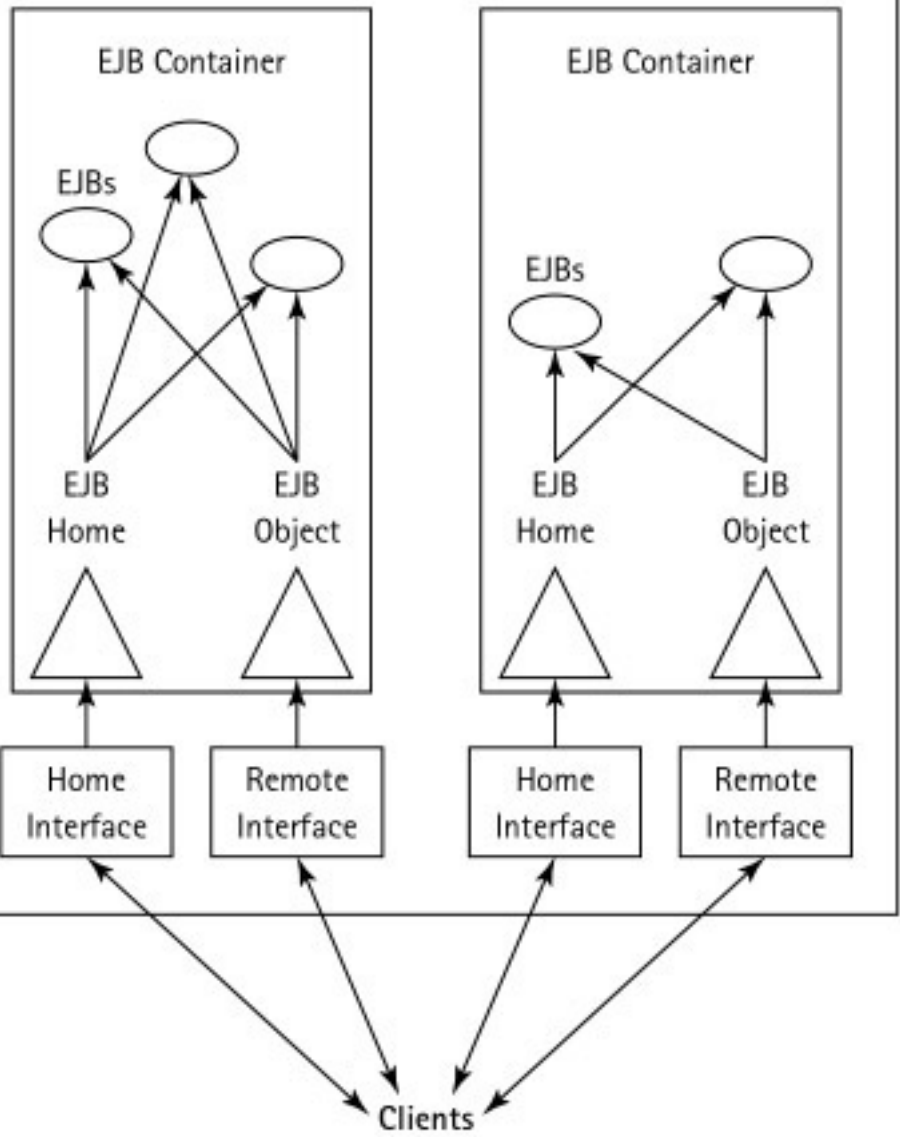
EJB Server

EJB Container

EJBs

EJB Home

EJB Object

EJB Container

EJBs

EJB Home

EJB Object

Home Interface

Remote Interface

Home Interface

Remote Interface

Clients

Chapter 15: EJB Entity Beans

You've read a bit about entity beans in previous chapters, but in this chapter you'll further explore the vitally important entity bean. You'll read about characteristics of entity beans and the entity bean interface, in addition to learning about container-managed and bean-managed persistence. You'll also learn about the life cycle of an entity bean, in addition to examining some code that implements an entity bean. Let's start by learning about the characteristics of entity beans and their uses.

Examining Entity Bean Characteristics

Entity beans provide an application with a consistent interface for accessing and manipulating data. A client interacts with the application's data by invoking entity bean methods defined in the remote interface, or by invoking session bean methods, which, in turn, invoke entity bean methods, again, through their respective remote interfaces.

Many of the concepts and methods needed to implement entity beans should be familiar to you. In Chapter 14, "EJB Session Beans," I covered bean activation and passivation, which are relevant to entity beans. Entity beans have a context object — an instance of the `EntityContext` class — that a bean uses to interact with the EJB container, like a session bean's `SessionContext` object. In short, entity beans have much in common with stateful session beans.

It should come as no surprise that you still code and implement a `home` and `remote` interface to represent the client view of an entity bean. The client still uses JNDI to locate a reference to the bean's `home` interface, as with session beans. The client still invokes methods contained in the `remote` interface to communicate with the container, then the bean, as with session beans.

A key difference between implementing a session bean and implementing an entity bean is that the session bean class implements the `javax.ejb.SessionBean` interface, and the entity bean implements the `javax.ejb.EntityBean` interface. Throughout this chapter, you will read about the methods in the `EntityBean` interface that require implementation.

Using Entity Beans for Persistent Data

You, the EJB developer, use entity beans to model some data relevant to the application. The data should be persistent; that is, the data source should be a persistent data store such as a database. The data should have meaning and use for more than a single client, or the data should be useful in a distributed application environment.

Accessing Entity Beans with a Primary Key

Because entity beans are an in-memory representation of some database data, it makes sense to use a key to access entity beans, just as you would for database data. A primary key used to access an entity bean is, of course, an instance of a Java class.

Primary key objects usually map to a key in the underlying database. However, you are not limited to defining primary key classes as objects of a relational database type, such as `varchar` or `integer`. You may construct your primary key as any Java object as long as the primary key class is serializable.

Looking at the Entity Bean Life Cycle

Entity beans, representing persistent data, have a long lifetime — as long as the data the bean represents. In reality, entity bean instances are pooled and the container may manipulate bean instances to service multiple clients. However, from a client perspective, the entity bean *is* the data. As you'll see in the code examples in this chapter, when a client takes action that impacts the state of an entity bean, the persistent data the bean represents also changes. In short, the life cycle of an entity bean closely mirrors that of the life cycle of the persistent data it models.

Later in this chapter, you learn about the states that an entity bean assumes throughout its lifetime.

Examining Entity Bean Client Scenarios

Entity beans are not bound to a particular client as stateful session beans are. The data being modeled by an entity bean has relevance and use to multiple clients (barring exceptional situations in which only one client has privilege to access some data). Hence, a container strategy that instantiates an entity bean and weds that bean to a single client may suffer performance bottlenecks.

You can imagine a scenario in which multiple clients require access to the same data, modeled as entity beans. Would a container vendor instantiate multiple beans, each bean representing the same data? Would a container vendor instantiate a single bean and write code that threads multiple requests to the entity bean's methods? Would the burden of writing thread-safe code fall on the EJB developer's shoulders?

The EJB specification requires that *all* enterprise beans must contain one and only one thread. The promise of thread-safe execution within the container is a big plus for the EJB architecture and helps ensure that the container and the enterprise application are stable.

Containers from different vendors can pool entity bean instances before filling these bean instances with data. As with session beans, the pooling strategy undertaken by the container vendor is completely transparent to the client and to the EJB developer. When several clients access the same data, some containers may instantiate several entity beans and present each bean to a separate client. These clients could access the bean instances simultaneously. But do you see a problem with the aforementioned approach?

The process of several clients accessing the same data at the same time raises data consistency issues. Client A accesses data. Client B accesses and changes the same data. However, client A has its copy of the data, which does not reflect the changes made by client B. At this point, client A has "stale" data. Also, if client A changes the data, then the changes made by client B may be overwritten and lost. In short, multiple clients accessing the same data requires additional housekeeping to ensure that the clients do not step on each other's changes.

These problems are not specific or unique to entity beans; they are the classic problems faced by database application developers. As you'll read later in this chapter, entity beans have means to ensure that these types of problems do not occur. For now, know that EJBs contain callbacks that help guard against lost update scenarios by synchronizing the bean instance with the database.

Because of the close relationship between entity beans and the data that they model, a short discussion of data persistence is in order.

Data Persistence

Actions taken on an entity bean affect data in the database. When the container creates a new entity bean, the data corresponding to the new bean must be inserted into the database. When a bean property corresponding to a column in the database changes, that changed value must be altered in the database as well. Some agency must be responsible for ensuring that changes in the bean are properly reflected in the database.

In the multiple-client, same-data scenario presented in the , you read that EJBs contain callbacks that help synchronize bean changes and database changes. In some cases, the EJB developer must write code to maintain data consistency, or persisting changes; in other cases, the container handles this chore.

The division of responsibility for persisting changes from the beans to the database results in different classifications for entity beans. Beans that can have the container manage persistence are called *container-managed persistence* (CMP) beans. Beans that contain code that manages persistence are called *bean-managed persistence* (BMP) beans.

Like stateful and stateless session beans, the interfaces you code to implement CMP beans and BMP beans are pretty much the same. The client accessing the bean has no knowledge of whether or not the entity bean is a CMP bean or a BMP bean. As they say, the devil's in the details. Let's take a closer look at these two bean classifications, starting with CMP beans.

Container-Managed Persistence Entity Beans

CMP beans are a boon to the EJB developer. By developing CMP beans, you can insert, update, or delete rows from multiple tables in a relational database *without writing a single line of SQL!* In other words, you write code that manages the activity of the CMP bean and the container automatically reflects your bean activity in the database.

If a client invokes a `create` method in the CMP bean's `remote` interface, the container is responsible for inserting the row(s) into the table(s). If a client invokes a set method to change the name of a CMP bean property assigned to a database column, the container is responsible for updating the corresponding value in the database. EJB folk call bean properties that map to columns in a table *container-managed fields*. A container-managed field can be any Java primitive type or any serializable object

To sweeten the deal, the container for CMP beans also handles data consistency problems previously mentioned, by using the same mechanism to apply database updates based on client invocations of bean set methods.

You may wonder how a CMP bean can possibly apply changes to a table in a relational database without SQL written by the bean developer. The container vendor must provide a tool that enables a bean developer to map CMP bean properties to columns in the database and to associate SQL with CMP bean actions, such as bean creation and removal.

The code in the CMP bean class does not contain any SQL. Also, the CMP bean is defined independently of the database used to store the bean's state. This independent definition and separation of the CMP bean from the database helps in making the bean reusable. The CMP bean contains business logic — not transaction details such as commits and rollbacks — and, with any luck, applies to multiple business scenarios within the organization.

By now you can see that, given a choice, you should strive to develop CMP bean entity beans. Even if you are a "go it alone" type, you should understand the wisdom in having the container manage the details of maintaining consistency between the CMP bean's state and the data contained in the database. Even if you rather enjoy writing SQL, you may see the wisdom in having the container issue any SQL needed, behind the scenes, to maintain the data.

CMP in EJB 2.0

Container-managed persistence will undergo significant changes in the upcoming EJB release 2.0. Sun and industry participants deemed the changes necessary

to clarify some ambiguities in the EJB 1.1 specification. For example, the following code defines a Customer CMP bean that relies on a dependent class:

```
public class Customer implements javax.ejb.EntityBean {
    public   int        custID ;
    public   String     custName ;
    public   Address     custAddress ;
    //Other fields and methods to complete the bean definition…
}
```

The `Address` class instantiates regular (non-EJB) objects:

```
public class Address {
    public String     street ;
    public String     city    ;
    //Other fields and methods that complete the class definition
}
```

The problem is that the EJB 1.1 specification is unclear as how to handle the persistence of objects of class `Address` that are instance variables of the Customer CMP bean. Are the `Address` objects serialized and saved? Are the instance variables (fields) within the `Address` objects mapped?

EJB 2.0 uses the *persistence manager* to alleviate this ambiguity. The persistence manager requires that the CMP bean developer code the relationships between objects of class `Address` (the dependent class) and the `Customer` bean class by using new tags in the deployment descriptor. The container vendor is responsible for providing tools that use the information in the deployment descriptor to generate the required access (`get`/`set` and `create`) methods that the bean instances need from the dependent objects. The following code shows a likely deployment descriptor that describes the relationship between the bean class and the dependent class:

```
<ejb-jar>
... .
   <enterprise-beans>
... .
   </enterprise-beans>

   <dependents>
      <dependent>
         <description>Address dependent class</description>
         <dependent-class>Address</dependent-class>
         <dependent-name>Address</dependent-name>
         <cmp-field><field-name>street</field-name></cmp-field>
         <cmp-field><field-name>city</field-name></cmp-field>
      </dependent>
   </dependents>

   <relationships>

   <!--  One to One: Customer Address -->
      <ejb-relation>
         <ejb-relationship-name>Customer-Address</ejb-relationship-name>
         <!--  defines role relationship from bean perspective -->
         <ejb-relationship-role>
            <ejb-relationship-role-name>
               customer-has-address
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <role-source><ejb-name>Customer</ejb-name></role-source>
            <cmr-field><cmr-field-name>custAddress</cmr-field-name></cmr-field>
         </ejb-relationship-role>

         <!--  defines role relationship from dependent object perspective -->
         <ejb-relationship-role>
            <ejb-relationship-role-name>
               address-belongs-to-customer
            </ejb-relationship-role-name>
            <multiplicity>One</multiplicity>
            <role-source><dependent-name>Address</dependent-name></role-source>
            <cmr-field><cmr-field-name>customer</cmr-field-name></cmr-field>
         </ejb-relationship-role>

      </ejb-relation>

   </relationships>

...

<ejb-jar>
The container uses the information coded in the deployment descriptor to generate the required accessor methods for both
classes.
The dependence on a container tool to map database columns to container-managed fields is a drawback to using CMP beans.
As of yet, container tools are not mature enough to address a wide range of mapping scenarios. Of course, tools never go
the distance; organizations will always have the odd requirement or two that baffles the current state of automation. In
these rare cases, the EJB developer must forgo CMP beans and write the code to have the bean manage the data's
```

persistence. Such entity beans are called *bean-managed persistence* (BMP) beans and are the subject of the next section.

Bean-Managed Persistence Entity Beans

As the name implies, a BMP bean must have code that maintains consistency between the state of the bean and the state of the data in the database. In other words, the BMP bean developer is responsible for writing code that explicitly manages the persistence logic of the database.

Naturally, the BMP bean developer must know about the database, the columns in the tables, the SQL used to access and modify the data, and when to issue the SQL statements. When compared to the work required to code a CMP bean, coding a BMP bean seems like a real effort!

Later in this chapter, I provide code for a CMP bean and show the required changes to transform the CMP bean into a BMP bean. First, you need to take a look at the `javax.ejb.EntityBean` interface, used to define both CMP bean and BMP beans.

.

Chapter 14: EJB Session Beans

Although you've been introduced to session beans in previous chapters, now it's time to learn more about the vitally important session bean. This chapter discusses the characteristics of session beans and the session bean interface. You can also learn about stateful and stateless session beans and their respective life cycles, including the myriad states of a session bean. In addition, you can examine some actual code that implements a session bean.

Understanding Session Beans

Session beans provide the generic components of an application that represent tasks or business processes. Session beans perform actions on behalf of a single client and serve as the entry for the client to the enterprise application. A client interacts with the application by invoking session bean methods defined in the remote interface, which access the functional behavior and services of the application.

The EJB developer uses session beans to model some process or task relevant to the application. These tasks range from the type you might expect, such as performing a computation or billing a customer account, to the type you might not expect, such as issuing an arrest warrant.

Session Bean Life Cycle

Because session beans are typically bound to a client, the lifetime of the session bean is usually as long as that of the client session. Often, an EJB server (or container depending on the implementation) has a time limit for inactive sessions, meaning the container may purge session beans after a period of inactivity. However, one of the benefits of the EJB architecture is that the EJB developer does not need to be concerned with bean-purging activities because such activity is one of the many services provided to the EJB developer.

I mentioned that usually session beans live only as long as the client session. One exception to this rule exists when EJB containers use *bean pools*. With bean pooling, the container or server creates multiple instances of session beans and reuses bean instances to service multiple clients at different times.

One Client, One Thread, One Session Bean, One Time

One ramification of a session bean being bound to a single client is that the EJB container ensures that only a single client has access to a single session bean instance. When a client invokes a session bean method through its remote interface, the EJB container (or server, depending on the implementation) guarantees that no other client can use the bound session bean instance. The benefit to you, the EJB developer, is that you do not have to write reentrant or thread-safe session bean code.

The EJB container is under no obligation to create bean instances for every client. The EJB container has a session bean instance ready for each and every client. The container takes care of the messy details of making session bean

instances available. The container may pull a session bean instance from a pre-created pool of beans, create a new one, or load a previously serialized session bean from storage. The details are not important; all you need to do is code your session beans to solve your business problems and let the EJB container take care of the rest.

As you've already read in Chapter 12, "The Elements of an EJB," session beans come in two flavors: *stateless* and *stateful*. Let's find out more about stateless session beans.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Summary

This chapter introduced the importance of EJB contexts. The nature of Java is to empower programmers by providing
tools that Java programs may use to discern relevant information about the program's environment. Bean developers
can glean useful information about the all-important EJB container during runtime execution from the context objects
discussed in this chapter. In the next two chapters we will continue to examine context objects as we discuss session
and entity beans in turn.

.

Understanding the EJB Container

Chapter 11, "A First Look at EJB," introduced the EJB roles that guide the responsible parties in the development, deployment, use, and maintenance of Enterprise JavaBeans and applications that use beans. In this section, I delve further into the EJB container. In particular, you read about the requirements for a functioning, compliant EJB container, as well as the EJB container requirements under the proposed EJB 2.0 release.

The Container's Responsibility

The container is responsible for providing the deployment tools and for managing the bean instances at runtime. By deployment tools, I mean a tool suite that generates various bean classes and helps the deployer make entries into a deployment descriptor.

The EJB specification does not define any API between deployment tools and the container. Therefore, you do not specify what entity provides the deployment tools. However, the *container* must have tools to support the generation of enterprise bean implementation classes.

Some EJB implementation classes and interfaces are used by all two (EJB 1.1) or three (EJB 2.0) bean types while other EJB implementation classes and interfaces are specific to the different bean types. I won't go into a laundry list of these classes now as you'll encounter all the bean classes and interfaces in subsequent chapters.

The Required EJB Container Runtime Environment

The requirements described here are considered to be the minimal requirements for an EJB 1.1–compliant container. A container provider is free to provide additional functionality that is not required by the EJB specification. As usual, when you stray away from the requirements, you run the risk of limiting your portability.

Java APIs Required for EJB Container

A complex environment that supports EJB, namely a container, requires more than the Java 2 Standard Edition. A Java 2 platform-based EJB container must make the following APIs available to the enterprise bean instances at runtime:

- Java 2 Standard Edition APIs

- EJB 1.1 or 2.0 APIs

- JNDI 1.2

- JTA 1.0.1

- JDBC 2.0

- JavaMail 1.1

The Required J2SE APIs

The EJB container must provide the full set of Java 2 Standard Edition APIs. However, the EJB container is allowed to, and usually does, make certain Java functionality unavailable to the enterprise bean instances by using the Java 2 platform security policy mechanism. The container must take steps to protect its environment and to prevent the enterprise bean instances from interfering with the container's functions.

JNDI 1.2

The container must provide a Java Naming and Directory Interface (JNDI) API name space to the enterprise bean instances. The container must make the name space available to an instance when the instance invokes the `javax.naming.InitialContext` no-arg constructor.

The container needs to make the home interfaces of other enterprise beans available by using the JNDI API. The code snippet that follows shows how a client can use the JNDI API to locate a home object:

```
Context initialContext = new InitialContext();
CartHome cartHome =
        (CartHome)
         javax.rmi.PortableRemoteObject.narrow(
                initialContext.lookup("java:comp/env/ejb/cart"),
                                      CartHome.class);
```

In Chapter 18, "Working with Persistent Data," I discuss the coding of client programs that access enterprise beans, so let's defer discussion of the details until then. For now, note that the preceding code uses the default no-arg constructor for the `InitialContext` class to construct the JNDI context.

The container also needs JNDI to locate resources used by the enterprise beans. The code snippet that follows shows how a bean method would use the default no-arg constructor to locate a resource. Here, the resource is an object of `javax.sql.DataSource`, or a database:

```
InitialContext initCtx = new InitialContext();
myData = (DataSource)initCtx.lookup("java:comp/env/mydatabase");
```

JTA 1.0.1

The container must include the Java Transaction API (JTA) 1.0.1 extension. The relevant JTA interface for EJB is the `javax.transaction.UserTransaction` interface.

Recall that the `EJBContext` interface defined three methods dealing with transactions. The `setRollbackOnly` and `getRollbackOnly` methods of `EJBContext` help developers of container-managed transaction beans control some behaviors of transactions. For developers of bean-managed transaction beans, these two methods are of no help. Bean-managed transaction developers rely on methods in the `javax.transaction.UserTransaction` interface to provide similar functionality.

The third method of the `EJBContext` interface, `getUserTransaction`, is also available to developers of bean-managed transaction beans. This method returns an object of a class that implements the `javax.transaction.UserTransaction` interface.

The container is not required to implement the other interfaces defined in the JTA specification. The other JTA interfaces are low-level transaction manager and resource manager integration interfaces and are not intended for direct use by enterprise beans.

JDBC 2.0 extension

The container must include some of the JDBC 2.0 extension and provide its functionality to the enterprise bean instances. The container makes the J2SE package `java.sql.*` available, as well as the J2EE package `javax.sql.*`, which allows the container to connect to a database and pool database connections.

Of particular interest is the `javax.sql.DataSource` interface, which allows you to connect to a database. Objects derived from the `DataSource` interface are located by using JNDI — the second code snippet in the section "JNDI 1.2" demonstrates a typical use of a `DataSource` object.

The JDBC 2.0 extension includes packages that perform low-level distributed transaction (XA) and connection pooling interfaces. These low-level interfaces are intended for integration of a JDBC driver with an application server, not for direct use by enterprise beans.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Understanding EJBs

From reading about JavaBeans in Chapter 6, "JSP, JavaBeans, and JDBC," you know that JavaBeans are software components. Because EJBs are software components, and the "JB" in EJB stands for "JavaBean," you may conclude that an EJB is an enhanced JavaBean. While this is a reasonable assumption, it is incorrect.

Both JavaBeans and Enterprise JavaBeans are standards for the development of software components, but that's where the similarity ends. A JavaBean is a software component that simply follows the naming conventions for properties and accessor methods, contains a no-argument constructor, and is persistent. EJBs are *deployable* components. Java developers can combine several Enterprise JavaBeans, and possibly other application parts, to form *deployable* components. In other words, an EJB provides some base functionality useful in a distributed computing environment.

Another distinction between JavaBeans and EJBs is that you do not need a special runtime, other than a Java Virtual Machine (JVM), to use JavaBeans, whereas you need a special runtime to use an EJB. The special runtime provides services to the EJBs, such as creating new instances or removing unused instances of a component class. You may recall from Chapter 1, "Enterprise Computing Concepts," that these special runtimes are called *containers*.

JavaBeans are components accessed within the context of a single process that exists in a single address space. Your Java component accesses a JavaBean as an instance of a JavaBean class by name. EJBs are distributed components accessed remotely within multiple contexts. Your application uses some sort of directory service to locate and use an EJB.

Before completing this discussion of differentiating JavaBeans and EJBs, it is worthy of mention that you can, and probably will, use JavaBeans as components when building EJBs.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Goals of the EJB Architecture

Sun Microsystems and its business partners set out to achieve several goals when they jointly developed the EJB
architecture. This section introduces these goals and offers some insights into the underlying intentions and
meanings.

The EJB specification document, available at http://java.sun.com/products/ejb/docs.html, contains the
goals for the EJB architecture. Our discussion in this section will focus on the general goals of the architecture. Later
in this chapter we will discuss goals added in new releases of the EJB specification.

1. **The Enterprise JavaBeans architecture will be the standard component architecture for building
   distributed object-oriented business applications in the Java programming language.**

   According to this goal, any organization developing server-side distributed software components in Java must
   follow the standards set by the EBJ specification. Perhaps, more accurately stated, this goal alerts organizations
   that are developing a distributed object software component architecture that they are wasting their time because
   the architecture has already been developed.

   Of course, all the players in the industry realize that the EBJ specification in its current form is not the last word on
   developing Java-based distributed software components. EJB specification will undergo revisions and
   enhancements as it is further developed. However, the foundations of the EJB architecture probably will not
   change for a significant amount of time.

2. **The Enterprise JavaBeans architecture will make it possible to build distributed applications by
   combining components developed using tools from different vendors.**

   A key feature of the EJB architecture is the construction of applications from software components (enterprise
   beans). The EBJ architecture defines *what* the enterprise beans are made of, including the classes and interfaces
   enterprise beans extend or implement. The EJB architecture does not define *how* a vendor constructs enterprise
   beans; the vendor is free to use whatever tools and technologies are at its disposal. The intention behind this goal
   is to create a rich marketplace in which vendors can reuse server-side distributed components and an industry of
   customers can mix and match different vendor offerings, secure in the knowledge that the developed enterprise
   beans work according to the EJB architecture.

3. **The Enterprise JavaBeans architecture will make it easy to write applications. Application developers will
   not have to understand low-level transaction and state management details, multi-threading, connection
   pooling, and other complex, low-level APIs.**

   As mentioned in Chapter 1, "Enterprise Computing Concepts," developing multi-tiered applications is not a trivial
   task. The application development team must be concerned with many important but nonbusiness services, such
   as application security, transaction management, and data integrity. The EJB architecture defines the
   responsibilities of the *EJB Container*, which provides the preceding, along with other, services to enterprise beans.
   The application development team, freed from having to understand and provide such services, can concentrate
   on solving the business problems they are paid to solve.

4. **Enterprise JavaBeans applications will follow the "Write Once, Run Anywhere" philosophy of the Java programming language. An enterprise bean can be developed once, and then deployed on multiple platforms without recompilation or source code modification.**

   Because the EJB architecture is a standard, as opposed to a product, any vendor that adheres to the standard can develop enterprise beans deployable on any suitable Java-enabled platform, such as an application server. When a vendor states that a piece of software is Java-enabled, the vendor *must* put the software through a series of tests described in the *Java Compatibility Kit*. An enterprise bean developed according to the EJB architecture must run on such a platform.

   Note the preceding goal states that the enterprise bean can be deployed without recompilation or source code modification. Given the wide variety of differences among enterprise computing environments, the goal may, at first, sound impossible to attain. As you can read later in this chapter, you can change some particulars of an enterprise bean without recompilation by using a *deployment descriptor*. The deployment descriptor, as the name implies, is a file that describes the particulars of how the enterprise bean is to be deployed. Think of a properties file and you get the essential idea.

5. **The Enterprise JavaBeans architecture will address the development, deployment, and runtime aspects of an enterprise application's life cycle.**

   This goal is an acknowledgment of the vitally important role distributed software components play in the enterprise application. However, this goal is somewhat vague in *how* the architecture addresses these aspects of the application. One way of looking at this goal is to remember that every phase of the application development cycle, in some manner, deals with the EBJ architecture. However, another way of looking at this goal is to remember that the EJB architecture has something to say about every phase of the application development cycle. Nonetheless, the importance of distributed software components cannot be understated, a fact known to the creators of the EJB architecture.

6. **The Enterprise JavaBeans architecture will define the contracts that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime.**

   As previously mentioned, the architecture defines a standard that enables developed enterprise beans to combine to form applications. A Java-based enterprise application consisting of enterprise beans is not static, such as an executable file formed by a compile/link cycle. A Java-based enterprise application is dynamic, with classes creating, loading, and disposing of instances during execution. To facilitate this dynamic behavior, the EJB architecture defines how enterprise beans interoperate at runtime.

   Above and beyond the stated behavior of communicating enterprise beans during application, execution is how these enterprise beans are created to permit this behavior. The EJB architecture defines what parts of enterprise beans should be exposed to the software development tools the vendors use to empower enterprise beans with the preceding behavior.

7. **The Enterprise JavaBeans architecture will be compatible with existing server platforms. Vendors will be able to extend their existing products to support Enterprise JavaBeans.**

   To create an EJB-capable server, the vendor must provide an environment in which enterprise beans can execute. The environment is called an EJB container. The architecture accomplishes this apparent feat of magic by providing a set of APIs that the EJB container must implement.

8. **The Enterprise JavaBeans architecture will be compatible with other Java programming language APIs.**

   This goal enables application developers to use other Java Language APIs in the construction, deployment, and execution of enterprise beans. Developers may use the JavaBeans API to construct beans that form pieces of enterprise beans. Developers may use the Java servlet API to write servlets that invoke enterprise beans, perhaps by using another Java API, such as RMI. Enterprise beans may use the JDBC API to manipulate persistent data. Developers use the JNDI API to locate enterprise beans in the distributed environment.

9. **The Enterprise JavaBeans architecture will provide interoperability between enterprise beans and non-Java programming language applications.**

Let's face it: For a software component to be truly useful, the component must communicate with pieces of software written in other programming languages. Most customers are not going to scrap their existing, *functioning* applications any time soon. The creators of the EJB architecture realized this sobering fact and, in response, established guidelines to ensure that an enterprise bean must have the ability to communicate with existing applications written in other programming languages. The architecture provides mechanisms to enable this cross-language interoperability.

10. **The Enterprise JavaBeans architecture will be compatible with the CORBA protocols.**

As you read in Chapter 2, "J2EE Component APIs," CORBA (the Common Object Request Broker Architecture) is the granddaddy of distributed object architectures. CORBA is more encompassing than Enterprise JavaBeans and has been around for over a decade. The EBJ architects recognized the importance of CORBA interoperability and have provided the Java IDL to enable Java programmers to represent Java objects in general, and enterprise beans in particular, as CORBA objects.

In short, the goals of the EJB architecture are ambitious but, for the most part, realized with the current release of the architecture. Sun Microsystems and its business partners are continually enhancing the EJB architecture with new releases. The following section discusses, release by release, the major features of Enterprise JavaBeans.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

.

Chapter 18: Creating EJB Clients

In previous chapters you learned about the elements of session and entity beans. You also read about EJB security and transactions and examined code that implements some enterprise beans. In this chapter, you explore the coding constructs required to create EJB clients. I begin by describing some general principles, followed by the requirements that EJB clients must follow. In addition, I describe the particulars of invoking session bean and entity bean methods.

Examining the Rules of EJB Clients

This section examines some properties that EJB clients share, regardless of the bean type the client accesses.

The Client Never Talks to the Bean or to the Container

I've covered the fact that an EJB client never interacts directly with an enterprise bean. Instead, the client interacts with the EJB object, which requests the EJB container to invoke bean methods on behalf of the client.

Let's review quickly the relationship between the client, the EJB object, the container, and the enterprise bean. The EJB object is defined by methods in the `remote` interface. It is through this `remote` interface that the client accesses the enterprise bean. The object that implements this `remote` interface is the EJB object.

The EJB specification does not define a client-container API, which is a ten-dollar way of saying that the client *never* interacts directly with the container. The workings of the container are transparent to the client. Good thing, too, given that one of the primary reasons developers are interested in distributed object technologies such as EJB is that the container provides numerous services to clients *in a transparent manner*.

The Client Acquires References to EJB Objects Through a home Object

Before the client can access an EJB object, the client needs a reference to the object. Regardless of the type of bean (session, entity, message-driven) accessed by the container by way of the EJB object, the client gets a reference to the EJB object the same way: by initially referencing a home object.

The home object (called an *EJB object factory* in the EJB specification) is an object from a class that implements the bean's `home` interface. As you've read, the `home` interface defines methods that allow a client to request the creation, location, and destruction of enterprise beans.

The Client Does Not Know Where the Bean Lives

The client has no knowledge of the location of the enterprise bean. That's the goal of a distributed object technology

such as Enterprise JavaBeans — the location of the bean is unknown to the client because the client accesses the bean regardless of the bean's location on the network. In other words, the client does not know what JVM the bean resides on.

The Client Does Not Know How the Bean Is Implemented

The client has no knowledge of the enterprise bean implementation. The bean may be an entity bean representing relational database data or a wrapping of a CICS (Customer Information Control System) transaction. The client accesses the bean in the manner dictated by the EJB specifications, not by the particulars of the implementation.

The technology used by the container provider is of no interest or importance to the client. Once again, the client accesses enterprise beans according to the EJB specifications, not according to any implementation-specific details of the container.

The Client Accesses Session Beans Much as It Does Entity Beans

A client, for the most part, accesses a session bean much as it does an entity bean. You've read that session beans have different uses and capabilities than entity beans. Of course, the client invokes different methods to utilize particular features of an entity bean instead of a session bean. That said, the methods and techniques to perform common operations on both bean types, such as locating, creating, and destroying the bean, are done by the client the same way.

The Client May Be One of Several Java Objects

An EJB client can be, but need not be, a Java application, through the standard use of a method:

```
public static void main( String[] args ){ ... }
```

A client can also be an applet or a Java servlet. A client can even be a CORBA client not necessarily written in Java.

A client can be another enterprise bean. (It is common for one enterprise bean to invoke the methods in another bean.) The client bean may or may not reside in the same container or the same JVM as the server bean.

Whatever the makeup of the client, the methods used by the client to access the enterprise bean are the same.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

The EJB component architecture was designed to provide a component architecture for enterprise-level applications.
Transactions are inherent to applications of the enterprise level, and with them come the difficulties of ensuring data
integrity. In this chapter we've read about transaction management mechanisms built into the EJB architecture. These
mechanisms give you the tools to control transactions at the level that best suits the nature of your enterprise
application. As you leverage the power of EJB transaction support, you will be able to build more robust applications
at the same time as you reduce your application development time.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Isolating Transactions at Different Levels

Transactions not only make completion of a unit of work atomic, but they also isolate the units of work from each other, provided that the system enables concurrent execution of multiple units of work (recall the isolation property of ACID).

The *isolation level* describes the degree to which the access to a resource manager by a transaction is isolated from the access to the resource manager by other concurrently executing transactions.

Isolation Conditions

Certain conditions, called *isolation conditions*, are a by-product of transactional systems when multiple transactions operate on the same data. These conditions are *dirty reads* and *phantom reads*.

Dealing with Dirty Reads

Dirty reads are a problem common to systems where multiple transactions can act on the same data. Suppose transaction B reads the *uncommitted* results from transaction A. If transaction A is rolled back, a dirty read has occurred because now transaction B is using data that is out-of-state, or invalid. Depending on the nature of your application, you can choose to handle dirty reads using one of two different types of strategies, known as repeatable reads and nonrepeatable reads.

Repeatable reads address the problem of dirty reads by forcing the data to look the same if read multiple times within the same transaction. The resource manager may *lock* the data, thereby denying access to that data to any other transactions. Another strategy to implement a repeatable read is to take a *snapshot* of the data. With the snapshot, other transactions may change the data, but the current transaction does not see the changes; the current transaction sees the snapshot.

Your application requirements determine if locking or taking a snapshot is necessary. It is noteworthy that most database managers implement locking at the row level.

The opposite of repeatable reads are *nonrepeatable reads*. When using nonrepeatable reads, different reads of the same data by the same transaction may yield different results, possibly from actions performed by different transactions. Nonrepeatable reads may not always be a problem. Your application may require that the same data be changed by multiple transactions and that these changes be seen by a single transaction.

Dealing with Phantom Reads

A phantom read occurs when a transaction detects new data between two separate read operations. The difference between a phantom read and a nonrepeatable read is that the data is always new in phantom reads, whereas the data in nonrepeatable reads may be new or changed.

The describes some locking techniques used in dealing with the problems resulting from dirty and phantom read problems.

Using Database Locking Strategies

The most popular database locking strategies are listed below:

- **Write locks:** Write locks prevent other transactions from changing the data until the current transaction completes. Typically, other transactions can read the uncommitted changes.

- **Exclusive write locks:** Exclusive write locks prevent other transactions from looking at or changing the data. This type of lock effectively eliminates the dirty read problem.

- **Read locks:** Read locks eliminate nonrepeatable reads by preventing transactions from changing just-read data until the reading transaction ends. The data can be read by other transactions but not updated by them.

You may think that the best solution is to code as restrictive a lock as possible (exclusive write lock), thereby avoiding all sorts of dirty and phantom read problems. Sadly, the more you lock your data, the slower your application may execute. If you have multiple users who need access to the same data (and who doesn't?), excessive locks cause the transaction manager to roll back or suspend transactions, thereby degrading application performance.

Until the application has been up and running for a while, you may not know which is the best locking strategy to use. That said, the offers some guidelines for managing isolation levels in enterprise beans.

Managing Isolation Levels

The following are guidelines for managing isolation levels in enterprise beans:

- The API for managing an isolation level is resource-manager specific. Therefore, the EJB architecture does not define an API for managing isolation levels.

- If an enterprise bean uses multiple resource managers, the Bean Provider may specify the same or different isolation level for each resource manager. As a result, if an enterprise bean accesses multiple resource managers in a transaction, access to each resource manager may be associated with a different isolation level.

- The Bean Provider must take care when setting an isolation level. Most resource managers require that all accesses to the resource manager within a transaction be done with the same isolation level. An attempt to change the isolation level in the middle of a transaction may cause undesirable behavior, such as a commit of the changes done so far.

- For session beans and message-driven beans with bean-managed transaction demarcation, the Bean Provider can specify the desirable isolation level programmatically in the enterprise bean's methods, using the resource-manager specific API. For example, the Bean Provider can use the `java.sql.Connection.setTransactionIsolation` method to set the appropriate isolation level for database access.

  The following code snippet sets an isolation level for a database connection:
  ```
  String urlForDB = … ;
  DataSource myDSrc = (javax.sql.DataSource)
  jndiCTX.lookup( urlForDB ) ;
  Connection myConn = myDsrc.getConnection ;
  MyConn.setTransactionIsolation
  ( Connection.TRANSACTION_READ_COMMITTED ) ;
  ```

- For entity beans with container-managed persistence, transaction isolation is managed by the data access classes that are generated by the persistence manager provider's tools. The tools must ensure that the management of the isolation levels performed by the data access classes will not result in conflicting isolation level requests for a resource manager within a transaction.

- Additional care must be taken if multiple enterprise beans access the same resource manager in the same transaction. Conflicts in the requested isolation levels must be avoided.

You cannot control isolation levels for container-managed transaction beans. Sun Microsystem's reasoning is that vendors had numerous difficulties implementing isolation levels at the transaction component level. The only real alternative is to use JDBC's isolation control facilities, as shown in the preceding code snippet.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Supporting Transactions in EJBs

Enterprise JavaBeans may be, and typically are, transactional and execute within a distributed transaction-processing environment. Being transactional means that enterprise beans, together with the EJB container, may implement the ACID transaction properties. The beauty of using Enterprise JavaBeans is that the EJB container performs many of the tasks needed to implement the ACID properties and the concepts described previously. The containers and EJB servers implement the low-level APIs needed to provide transaction support services and protocols, thereby freeing the application developer from tending to the details of coding these services. In short, EJB hides much of the complexities of transaction support in a distributed transactional environment from the developer.

Enterprise JavaBeans support transactions that adhere to the OMG Object Transaction Service (OTS) for *flat transactions*. The OTS is a distributed transaction processing service specified by the Object Management Group. The OMG specification extends the CORBA model and defines a set of interfaces to perform transaction processing across multiple CORBA objects. As of release 2.0, Enterprise JavaBeans do not provide support for nested transactions.

Managing EJB Transactions

You, the EJB developer, freed from much of the mundane details of managing transactions, rarely have to deal with low-level transaction APIs. Your code never has to directly deal with the underbelly of the transaction manager. Your job is to write application code that addresses the needs of your customers. You tap the power of EJB by deciding how EJB manages your transactions.

What choices do you have? You can tell EJB to use a *programmatic transaction demarcation* or use a *declarative transaction demarcation*. Let's explore these two choices next.

Managing Transactions at the Bean Level

With programmatic transaction demarcation (often called bean-managed transaction demarcation), the EJB developer is responsible for defining the boundaries of the transaction. Your code should expose methods that start the transaction, perform all required operations, and commit or abort the transaction. In short, bean-managed demarcation is the traditional way of coding transaction-driven applications.

You might be wondering what, with bean-managed transaction demarcation, is the benefit to using Enterprise JavaBeans. Well, you still have the benefits afforded by a distributed object manager, such as location transparency. Also, you need not fret over coding the details of having the transaction manager coordinate efforts with the resource managers during the execution of the transaction operations. You write code invoking methods that implement the operations required to complete the transaction; the EJB container and server handle the coordination of the system components needed to complete (or abort) the transaction.

With bean-managed transaction demarcation, the enterprise bean code demarcates transactions using the `javax.transaction.UserTransaction` interface. All resource manager accesses between the `UserTransaction.begin` and `UserTransaction.commit` calls are part of a transaction. Hence, your beans

would invoke the `UserTransaction.begin` method of the current transaction object to start a transaction and invoke the `UserTransaction.commit` or `UserTransaction.rollback` method to end a transaction. <u>Listing 17-1</u> has a template that illustrates the use of the aforementioned methods.

Listing 17-1: Using bean-managed transaction demarcation

```
public void doMyTransaction( SomeClass1 anObjClass1,
                             SomeClass2 anObjClass2,
                             …
                           ) {
  try {
    //Get a reference to the current transaction
    //
    javax.transaction.UserTransaction myTrans =
                    ejbContext.getUserTransaction() ;
    //start the transaction
    //
    myTrans.begin() ;
    //Execute the operations that constitute the transaction
    //
    doOperation1( anObjClass1 ) ;
    doOperation2( anObjClass2 ) ;

    …
    //Commit the transaction. If any of the doOperation()
    //methods fail, we may assume the failed operation passed
    //an exception back to this method.
    myTrans.commit() ;
  }
  catch (Exception anExc) {
    //Tend to exceptions. You would need logic to
    //determine if a rollback is called for.
    //
    myTrans.rollback();
  }

}
```

Notice how the code in <u>Listing 17-1</u> explicitly sets the transaction boundaries.

Because you make use of the Java Transaction API for bean-managed transaction demarcation, perhaps a few words on the API is in order.

Using the Java Transaction API

The Java Transaction API consists of two elements: an application-level transaction demarcation interface and a standard Java mapping of the X/Open XA protocol. You've seen the application-level transaction demarcation interface in <u>Listing 17-1</u>. <u>Listing 17-2</u> shows the `javax.transaction.UserTransaction` interface.

Listing 17-2: The javax.transaction.UserTransaction interface

```
package javax.transaction ;
interface UserTransaction{

  //
  public void begin()
        throws NotSupportedException,SystemException ;
```

```
  public void commit()
        throws RollbackException, HeuristicMixedException,
             HeuristicRollbackException,
             java.lang.SecurityException,
             java.lang.IllegalStateException,
             SystemException ;
  public void rollback()
          throws java.lang.IllegalStateException,
               java.lang.SecurityException,
               SystemException ;
  public void setRollbackOnly()
          throws java.lang.IllegalStateException,
               SystemException ;
  public void getStatus()
          throws SystemException ;
  public void setTransactionTimeout()
          throws SystemException ;

}
```

Note The `javax.transaction.UserTransaction` package in earlier releases of J2EE contained status codes. In the latest J2EE release, Sun moved the status codes out of the `UserTransaction` interface and placed the codes in a separate interface, `javax.transaction.Status`.

The `begin` method creates a new transaction and associates it with the current thread.

The `commit` method completes the transaction associated with the current thread by running the two-phase commit process. When `commit` completes, the thread becomes associated with no transaction.

The `rollback` method rolls back the transaction associated with the current thread. When `rollback` completes, the thread becomes associated with no transaction.

The `setRollBack` method modifies the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction. You can use `setRollBack` to determine what your transaction components do without committing any changes to permanent storage.

The `getStatus` method obtains the status of the transaction associated with the current thread.

The `setTransactionTimeout` method modifies the value of the timeout value that is associated with the transactions started by the current thread with the begin method. If an application has not called `setTransactionTimeout`, the transaction service uses some default value for the transaction timeout.

Managing Transactions at the Container Level

With declarative transaction demarcation (often called container-managed transaction demarcation), the EJB container is responsible for defining the boundaries of the transaction. Your code never invokes any method that explicitly starts a transaction, or commits or rolls back a transaction. The EJB container does all the necessary work for you.

Just how does EJB accomplish the necessary tasks of demarcating your transactions? The EJB container intercepts any client request that involves a transaction and automatically starts, manages, and ends the transaction. As you've probably guessed, you tell the EJB container what to do by setting various parameters in the bean's deployment descriptor. The parameter you must set to direct the EJB container to participate in container-managed transaction demarcation is the `transaction` attribute. The next section describes this vitally important parameter, providing examples of deployment descriptors that inform the EJB container of the `transaction` attribute value.

The transaction Attribute

A `transaction` attribute is a value associated with a method of a session or entity bean's `remote` or `home` interface. The `transaction` attribute specifies how the container must manage transactions for a method when a client invokes the method via the enterprise bean's `home` or `remote` interface.

For a session bean, the `transaction` attributes must be specified for the methods defined in the bean's `remote` interface and all the direct and indirect superinterfaces of the `remote` interface, excluding the methods of the `javax.ejb.EJBObject` interface. `transaction` attributes must not be specified for the methods of a session bean's `home` interface.

For an entity bean, the `transaction` attributes must be specified for the methods defined in the bean's `remote` interface and all the direct and indirect superinterfaces of the `remote` interface, excluding the `getEJBHome`, `getHandle`, `getPrimaryKey`, and `isIdentical` methods; and for the methods defined in the bean's `home` interface and all the direct and indirect superinterfaces of the `home` interface, excluding the `getEJBMetaData` and `getHomeHandle` methods.

Providing the `transaction` attributes for an enterprise bean is an optional requirement for the application assembler. For a given enterprise bean, an application assembler must either specify a value of the `transaction` attribute for all of the methods for which a `transaction` attribute must be specified, or an assembler must specify none. If the `transaction` attributes are not specified for the methods of an enterprise bean, the deployer must specify them.

Enterprise JavaBeans define the following values for the `transaction` attribute:

- **`NotSupported`:** If you set a bean or method's `transaction` attribute to `NotSupported`, the bean cannot be involved in a transaction. If a method is currently involved in a transaction to invoke a bean method coded as `NotSupported`, the container should suspend the transaction until the `NotSupported` method completes.

- **`Required`:** If you set a mean or method's `transaction` attribute to `Required`, the bean or method always runs in some transaction context. Any method of a bean with the `Required` transaction attribute, invoked by a bean currently involved in a transaction, participates in that transaction. If a method of a bean not involved in a transaction invokes a bean with the `Required transaction` attribute, the bean will initiate a new transaction.

- **`Supports`:** If you set a bean or method's `transaction` attribute to `Supports`, then the bean runs in a transaction only if the invoking method or client is already involved in a transaction. If the invoking method is not involved in a transaction, the bean with a `transaction` attribute of `Supports` executes outside of a transaction context.

- **`RequiresNew`:** Code a bean or method's `transaction` attribute to `RequiresNew` if you always want a new transaction to start when a method of the bean is invoked. If a client or another bean that invokes the `RequiresNew` bean is already involved in a transaction, the container should suspend the already running transaction and start a new transaction. The new transaction lasts as long as the method of the `RequiresNew` bean executes. After the method of the `RequiresNew` bean completes execution, the container resumes execution of the suspended transaction.

- **`Mandatory`:** Use a value of `Mandatory` to instruct the EJB container to include the bean in an already executing transaction. Hence, a transaction must already be executing when a bean with a transaction attribute of `Mandatory` is invoked. If no transaction is executing, the invocation throws a `javax.transaction.TransactionRequiredException`.

- **Never:** Beans coded with a transaction attribute of `Never` cannot be a part of a transaction. If a bean or client involved in a transaction invokes a bean with a transaction attribute of `Never`, the invocation throws a `RemoteException`.

Note   Message-driven beans (EJB 2.0) support only the `Required` and `NotSupported` transaction attribute values and must be coded for the Message bean's `onMessage` method.

Coding the Transaction Attribute

As previously mentioned, you code a value of the `transaction` attribute in the bean's deployment descriptor. EJB enables you to provide a value of the `transaction` attribute for the entire bean or for methods within beans. Also, you may code different values of the attribute for different methods of the same bean.

You use the container-transaction elements to define the `transaction` attributes for the methods of session and entity bean's `remote` and `home` interfaces and for the `onMessage` methods of message-driven beans. Each `container-transaction` element consists of a list of one or more `method` elements, and the `trans-attribute` element. The `container-transaction` element specifies that all the listed methods are assigned the specified `transaction` attribute value. It is required that all the methods specified in a single `container-transaction` element be methods of the same enterprise bean.

The `method` element uses the `ejb-name`, `method-name`, and `method-params` elements to denote one or more methods of an enterprise bean's `home` and `remote` interfaces. The three legal styles of composing the `method` element are as follows:

```
Style 1:
<method>
<ejb-name> EJBNAME</ejb-name>
<method-name>*</method-name>
</method>
```

This style is used to specify a default value of the `transaction` attribute for the methods for which there is no Style 2 or Style 3 element specified. There must be at most one `container-transaction` element that uses the Style 1 `method` element for a given enterprise bean.

```
Style 2:

<method>
<ejb-name> EJBNAME</ejb-name>
<method-name> METHOD</method-name>
</method>
```

This style is used for referring to a specified method of the `remote` or `home` interface of the specified enterprise bean. If there are multiple methods with the same overloaded name, this style refers to all the methods with the same name. There must be at most one `container-transaction` element that uses the Style 2 `method` element for a given method name. If there is also a `container-transaction` element that uses Style 1 element for the same bean, the value specified by the Style 2 element takes precedence.

```
Style 3:

<method>
<ejb-name> EJBNAME</ejb-name>
<method-name> METHOD</method-name>
<method-params>
<method-param> PARAMETER_1</method-param>
...
<method-param> PARAMETER_N</method-param>
</method-params>
</method>
```

This style is used to refer to a single method within a set of methods with an overloaded name. The method must be one defined in the `remote` or `home` interface of the specified enterprise bean. If there is also a `container-`

`transaction` element that uses the Style 2 element for the method name, or the Style 1 element for the bean, the value specified by the Style 3 element takes precedence.

The optional `method-intf` element can be used to differentiate between methods with the same name and signature that are defined in both the `remote` and `home` interfaces.

Listing 17-3 shows the specification of the `transaction` attributes in a deployment descriptor. The `updatePhoneNumber` method of the `EmployeeRecord` enterprise bean is assigned the transaction attribute `Mandatory`; all other methods of the `EmployeeRecord` bean are assigned the attribute `Required`. All the methods of the enterprise bean `AardvarkPayroll` are assigned the attribute `RequiresNew`.

Listing 17-3: Deployment descriptor showing transaction attribute values

```
<ejb-jar>
...
<assembly-descriptor>
...
<container-transaction>
<method>
<ejb-name>EmployeeRecord</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
<method>
<ejb-name>EmployeeRecord</ejb-name>
<method-name>updatePhoneNumber</method-name>
</method>
<trans-attribute>Mandatory</trans-attribute>
</container-transaction>
<container-transaction>
<method>
<ejb-name>AardvarkPayroll</ejb-name>
<method-name>*</method-name>
</method>
<trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

Another important transaction property supported by EJB is the property of *Isolation Level*. The next section provides some details about this property.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Transaction Properties

Many of us have an intuitive understanding of a transaction, which is a set of operations that transforms data from one consistent state to another. The set of operations that constitutes the transaction is usually initiated by a client and consists of client requests for data access, data manipulation, and data removal.

Although describing a transaction is simple, implementing a transaction isn't quite as simple. The difficulty often arises when the set of operations must be implemented across a group of distributed components using data from several locations. For example, if you write an enterprise bean that represents an online shopping cart, it may need to make entries in your warehouse database in Chicago, your accounting database in Dallas, and your customer database in New York. The operations involved in these transactions must be coordinated and possibly synchronized — this can be a source of difficulty.

Our goal in this chapter is to understand transactions and how EJBs help ease these and other difficulties inherent in implementing and managing transactions.

Understanding Transaction Terminology

Before you can explore transactions, it is necessary to become acquainted with the most commonly used terms found in the world of transactions.

Processing Transactions

To process transactions, a system needs access to several components, namely *application components,* a *resource manager,* and a *transaction manager.* Let's take a look at these vitally important terms.

Application Components

The application consists of one or more application components. Application components are clients for the transactional resources, or the programs with which the application developer implements business transactions. As you might have guessed, enterprise beans are the application components of an EJB-based transactional system. Application components operate on data by using the services of a resource manager.

Resource Managers

The resource manager supports distributed transactions by managing resources. In the context of a transactional system, a resource may represent a permanent data store or a message queue.

Resource managers implement a transaction resource interface, such as a database driver to communicate with a relational database. The X/Open XA protocol, supported by nearly all the major database players, is another example

of a resource manager.

Transaction Managers

The transaction manager is the heart and soul of a transaction-processing environment. Its primary responsibilities are to create transactions when requested by application components and to manage the actions resulting from the success or failure of transactions by working with the resource managers.

The transaction manager may act as a traffic cop, temporarily halting or suspending one transaction and enabling another to proceed. Perhaps one transaction requires a resource held by another, currently executing transaction. If so, the transaction manager may abort the one transaction or suspend it until the transaction holding the needed resource completes. The transaction manager coordinates activities with the resource managers to make the previously held resource available to the waiting transaction. Once available, the transaction manager allows the suspended transaction to resume execution.

These three previously mentioned components are discussed later in this chapter, specifically in the section "Transaction Processing Concepts." Now, it's time to look at the four defining characteristics of a transaction: the ACID properties.

Learning the ACID Properties

Transactions have several important properties known as *ACID* properties, which are as follow:

- **Atomicity:** Transactions implement several operations, which must be completed in order for the transaction to be successful. If any one of the constituent operations fails, the entire transaction fails. We say that transactions are *atomic* in nature, meaning the operations of the transaction must be considered as an indivisible unit when deciding whether or not the transaction is successful.

- **Consistency:** Transactions must ensure that the system remains in a consistent state, regardless of whether or not the transactions successfully execute. The consistency of the system is defined by business rules, which the application must enforce and implement.

- **Isolation:** The results of transaction execution should be independent of other executing transactions. Stated differently, the effects of one transaction should not be visible to any other transaction until the transaction completes. Although several transactions may be executing in parallel, the net effect is as if the transactions were executing in serial.

- **Durability:** Changes made by a transaction must be permanent. Once the transaction successfully completes its work, the system ensures that subsequent failures (software, hardware, network, and so on) cannot undo the changes to the system resulting from a transaction.

When you develop a single-user program, implementing the ACID properties is simple. However, when you are engaged in the development of enterprise-wide applications whereby hundreds or thousands of users have access to common, distributed data stores, implementing the ACID properties is anything but simple.

In the past, the onus for supporting and implementing the highly desirable ACID properties fell to the application programmer. With EJB, you can see that some of this burden is lifted from the programmer and placed on the back of the EJB container. However, the application programmer is still left with a fair share of the load.

Differentiating Transaction States

A transaction results in one of two states: *committed* or *rolled back*. A transaction is committed if all its component operations successfully execute and the changes in system state are persisted. In other words, the transaction successfully implemented the ACID properties previously mentioned.

A transaction is rolled back if any one of its component operations fails. You may recall that the atomicity property of a transaction requires that all of the transaction's operations successfully complete. If one or more operations fail, the system must be put back into a consistent state. Recall that the consistency property requires that the transaction must leave the system in a consistent state. A good strategy for leaving the system in a consistent state after a transaction operation failure is to undo all changes made by the transaction's component operations (if any). The process of undoing the changes made by the component operations is called rolling back the transaction.

A rolled-back transaction should remove all traces of its existence except for, perhaps, entries in various log files. Please realize that a rolled-back transaction may not necessarily result from an error; it may be the result of the implementation of a business rule or a change of state not present at the transaction's onset.

Before continuing the discussion on transactions and transaction management, let's describe some transaction processing concepts.

Learning Some Transaction Concepts

In the world of transaction processing, many important concepts exist, such as *transaction context*, *two-phase commit, resource enlistment, transaction demarcation*, and *transaction models.*

Transaction Context

Because multiple application components and resources participate in a transaction, the transaction manager needs to establish and maintain the state of the transaction as it occurs, usually in the form of a transaction context.

The transaction context is an association among the transactional operations on the resources, and the application components invoking the operations. During the course of a transaction, all the components participating in the transaction share the transaction context. Thus, the transaction context logically envelops all the operations performed on transactional resources during a transaction. The transaction manager is responsible for managing the transaction context in a manner transparent to the applications and resource managers.

Two-Phase Commit

The two-phase commit protocol between the transaction manager and all the resources enlisted for a transaction ensures that either all the resource managers commit the transaction or they all abort.

In this protocol, when the application requests for committing the transaction, the transaction manager issues a "get ready to commit" request to all the resource managers involved. The "get ready" request is the initial part of phase one in the two-phase commit protocol.

Each resource manager may send a reply to the transaction manager indicating whether or not the resource manager is ready to commit the changes made by the currently executing transaction. The reply is the second part of phase one in the two-phase commit protocol. Only when all the resource managers reply that they are ready for a commit does the transaction manager issue a commit request to all the resource managers. The request is (you guessed it) phase two of the two-phase commit protocol. If a single resource manager vetoes the commit request made by the transaction manager, the transaction manager issues a rollback request and the transaction is rolled back.

Resource Enlistment

The ten-dollar phrase used to describe how the resource managers notify the transaction manager of their involvement in a transaction is *resource enlistment*. The transaction manager must have a mechanism to keep track of the specific resource managers (and their underlying resources) used in each transactions. Also, the transaction manager needs to keep tabs on the resource managers to implement the two-phase commit protocol.

At the end of a transaction (after a commit or rollback), the transaction manager *delists* the resources, which severs any relationship between the transaction and the resource(s).

Transaction Demarcation

Transaction demarcation involves defining the players that are involved in the transaction, including who starts the transaction, who performs an update or a delete, and who commits or rolls back the transaction. Transaction demarcation is a way of marking groups of operations as a transaction.

Transaction Models

A transaction model is a generalized framework that describes a class of transactions. The two most popular, or often-used, models are the flat and the nested transaction models.

The Flat Transaction Model

The flat transaction model is a direct implementation of a series of operations that constitute a transaction. The operations are applied in a specific order. If any single operation fails, the entire transaction is aborted, or rolled back, in accordance with the atomicity property of transactions.

The resource managers enlisted in the processing of a flat transaction typically hold off making permanent changes to the underlying resource until the transaction manager directs the managers to commit the changes. This way, the transaction can be safely rolled back if an operation fails and the system is left in a consistent state.

The Nested Transaction Model

In contrast to the flat model, the nested transaction model enables a transaction to contain multiple units of work. The multiple units of work combine to form the complete unit of work required to complete the transaction.

The classic example of a nested transaction is the trip-planning problem. Consider booking a trip that contains several legs. The entire trip is a transaction; each of the legs may be considered a contained unit of work. The transaction manager could book each leg separately, or independently of the other legs. If all legs are successfully booked, the transaction manager could direct the resource managers to commit the transaction.

So far, the previous description of planning a trip may resemble a flat transaction. However, consider what a transaction manager would do if one of the legs were currently unavailable for booking. If this trip were implemented as a flat transaction, the transaction manager would have no choice but to direct the resource managers to roll back the transaction. However, with a nested transaction, the currently unavailable trip is a separate unit of work contained within the larger transaction. The transaction manager can put the transaction on hold and try to book the unavailable leg later. If the later booking is successful, the trip transaction can be committed.

The application program can be written to try alternate legs when a leg cannot be immediately booked. If this is the case, the transaction manager suspends the trip transaction while the application attempts to locate an alternative (perhaps to complete the leg by boat instead of by jet). If an alternative is located, the transaction can be committed.

The application can also be written to book the "hard-to-book" legs first. With nested transactions, you are not necessarily tied to a rigid order of operations as with the flat model. Of course, all the legs eventually must be booked in order for the transaction to be committed. The nested transaction model affords the application developer more flexibility in designing transactions.

Now that you have some terminology under your belt, it's time to discuss how Enterprise JavaBeans support transactions.

Chapter 17: EJB and Transaction Management

Overview

If you had to pick a property common to all enterprise-wide applications, surely the use of a database would be at the top of the list. A database does more than store data, however. A database helps ensure that the application has access to *accurate* data, although data accuracy is not the sole responsibility of the database. The database needs help from the application, and from various pieces of system software, to ensure that the data housed in the database is indeed accurate. The strategy used to ensure data accuracy includes the use and management of *transactions*.

Transactions are a proven technique for simplifying application programming. Transactions give the application programmer the freedom from dealing with the complex issues of failure recovery and multi-user programming.

If the application programmer uses transactions, the programmer divides the application's work into units called transactions. The transactional system ensures that a unit of work fully completes or is fully rolled back. Furthermore, transactions enable the programmer to design the application as if it were running in an environment that executes units of work serially.

You may recall that a prime motivation for using a distributed component architecture to develop applications is to take advantage of the services provided by the architecture. One of the vitally important services provided by the Enterprise JavaBeans architecture is *transaction* support. This chapter discusses how Enterprise JavaBeans provide transaction support, starting with a description of the transaction properties, and then following with a recap of the Java APIs involved in transactions. Next, you can explore the services relating to transactions provided by EJB, and you can finish the chapter with an examination of code that provides transaction support.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

Security is a complex topic that continues to grow in importance each and every day. In this chapter, you've glanced at Java's security features with emphasis on what Enterprise JavaBeans have to offer in the information security realm. Thanks to the advanced architecture of EJB, you can build on the security features of Java to leverage security in the construction of your EJBs.

Even though EJBs may not be the dominant piece of your applications, EJBs typically are the gateway to mission-critical corporate data. This data is the target of hackers, crackers, and other scurrilous individuals with a not-too-nice agenda. Fortunately, you have the tools and the technology to prevent most of these disasters.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

.

Securing the Rest of Your Environment

Systems that are not 100 percent pure Java (and that means just about every large system out there) have additional security concerns. Some techniques for addressing security in non-Java systems are SSL, HTTPS, IPSEC, and, of course, using passwords.

SSL, or Secure Sockets Later, is a technology that enables a network administrator to secure transmissions between two sockets. Once the network has established a channel connecting the two sockets, the transmissions are encrypted.

HTTPS is a secure version of HTTP. The deal here is that HTTPS sends encrypted requests from Web hosts to servers.

IPSEC is perhaps the strongest level of network security. In IPSEC, *all* IP messages sent over the network are encrypted.

While encryption has an impact on network performance, the protection of information, corporate or private, can be far more important.

.

EJB Players and Security

Here is a brief rundown of the EJB players (roles) and their responsibilities in the security arena.

The Enterprise Bean Provider

The bean provider should implement neither security mechanisms nor hard-code security policies in the enterprise bean's business methods. Rather, the bean provider should rely on the security mechanisms provided by the EJB container, and should let the application assembler and deployer define the appropriate security policies for the application.

The bean provider and application assembler may use the deployment descriptor to convey security-related information to the deployer. The information helps the deployer to set up the appropriate security policy for the enterprise bean application.

The bean provider has methods available in the EJB API to glean security-related information. The `javax.ejb.EJBContext` interface provides two methods that enable the bean provider to access security information about the enterprise bean's caller:

```
java.security.Principal getCallerPrincipal();
boolean isCallerInRole(String roleName);
```

The main purpose of the `isCallerInRole()` method is to enable the bean provider to code the security checks that cannot be easily defined declaratively in the deployment descriptor using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The Application Assembler

The application assembler (which could be the same party as the bean provider) may define a *security view* of the enterprise beans contained in the `ejb-jar` file. Providing the security view in the deployment descriptor is optional for the bean provider and application assembler.

The security view consists of a set of *security roles.* A security role is a semantic grouping of permissions that a given type of users of an application must have in order to successfully use the application.

The application assembler defines *method permissions* for each security role. A method permission is a permission to invoke a specified group of methods of the enterprise bean's `home` and `remote` interfaces. In other words, the application assembler is responsible for the creation of roles and placing this information in the deployment descriptor as described in previous sections.

The Deployer

The deployer is responsible for ensuring that an assembled application is secure after it has been deployed in the target operational environment.

The deployer is responsible for assigning the security domain and principal realm to an enterprise bean application. Security domains and realms are security components from Java 2, not EJB.

Typically, the deployer is more knowledgeable of the EJB's operational environment than the other parties. Hence, the deployer may use the security view defined in the deployment descriptor by the bean provider and application assembler merely as "hints" and may change the information whenever necessary to adapt the security policy to the operational environment.

Providing security information may fall upon the deployer's shoulders. The EJB specification states that the creation of security information (roles, method permissions, and so on) by the application assembler is not mandated. If the application assembler fails to provide security information, the deployer must pick up the slack.

The EJB Container Provider and Server Provider

The EJB Container Provider is responsible for providing the deployment tools that the deployer can use to perform security-related tasks described in this chapter.

The deployment tools read the information from the deployment descriptor and present the information to the deployer. The tools guide the deployer through the deployment process, and present him or her with choices for mapping the security information in the deployment descriptor to the security management mechanisms and policies used in the target operational environment.

The deployment tools' output is stored in an EJB container–specific manner, and is available at runtime to the EJB container.

The System Administrator

The system administrator's security responsibilities usually fall outside the scope of EJB. That is, the system administrator is involved with security of the network, such as granting users' access to the network and adding and removing user network accounts. In addition, if the EJB container provides an audit trail facility, the system administrator is responsible for its management.

Note   The EJB specification defines no security requirements for the Persistence Manager.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Securing EJB Applications

So far you've taken a high-level look at the security features of the Java language and JVM. Enterprise JavaBeans bring additional security requirements to the table because EJBs are distributed software components. The two overriding concerns with designing EJBs are *identifying and authorizing* the client and *managing access to resources.*

Identifying and Authorizing Clients

During the design of EJBs, the designer might have groups of users that can access the bean. These groups of users may be classified into *roles.* For example, *managers*, *tellers*, and *customers* can access a bank account EJB and a specific role can be assigned to each of the three groups. The bean designer decides the business methods that each role has access to within the bean. Table 16-1 shows a possible grid that describes who has access to what methods.

Table 16-1: Method Permissions in a Bank Account EJB

Method	Manager Role	Teller Role	Customer Role
createAcct	Yes	Yes	No
removeAcct	Yes	Yes	No
auditAccount	Yes	No	No
creditAcct	No	No	Yes
debitAcct	No	No	Yes
transferFunds	No	No	Yes

The assignment of the role to the method is usually done through an EJB deployment tool. If you are hard-core, you may code the permissions by role in the bean's deployment descriptor. Listing 16-1 shows a piece of a deployment descriptor with a role and method-permission instructions.

Listing 16-1: Deployment descriptor snippet showing role and method permissions

```
<assembly-descriptor>
  <security-role>
    <description>This role represents bank branch managers
    </description>
    <role-name>manager</role-name>
  </security-role>

  <method-permission>
```

```
        <role-name>manager</role-name>
        <method>
            <ejb-name>BankAccountBean</ejb-name>
            <method-name>createAcct</method-name>
            <method-name>removeAcct</method-name>
            <!-- and so on --!>
        </method>
    </method-permission>

    <method-permission>
        <role-name>customer</role-name>
        <method>
            <ejb-name>BankAccountBean</ejb-name>
            <method-name>transferFunds</method-name>
            <method-name>debitAcct</method-name>
            <!-- and so on --!>
        </method>
    </method-permission>

    <container-transaction>
        <!-- We don't care about transaction attributes for now --!>
    </container-transaction>
</assembly-descriptor>
```

EJB uses a default role called, appropriately enough, *everyone*.

Managing Access to Resources in EJBs

The EJB specification indicates that permissions can be declaratively assigned to EJB methods via the deployment descriptor. As previously mentioned, your EJB tools should make such assignments a bit easier by providing a graphical interface. Table 16-2 shows the EJB 1.1 security restrictions as delineated in Section 18.2.1.1 of the specification.

Table 16-2: EJB Security Rules

Permission	EJB Rule
`java.security.AllPermission`	deny
`java.awt.AWTPermission`	deny
`java.io.FilePermission`	deny
`java.net.NetPermission`	deny
`java.util.PropertyPermission`	grant `read, *`; deny others
`java.lang.reflect.ReflectPermission`	deny
`java.lang.RuntimePermission`	grant `queuePrintJob`; deny others
`java.lang.SecurityPermisison`	deny
`java.lang.SerializablePermission`	deny
`java.net.SocketPermission`	grant `connect, *`; deny others

EJB security uses policy files with the EJB Security Manager first enforcing the permissions contained in the policy file. If those checks pass successfully — the user has permission to execute the given method as delineated by the permissions in the file — then the additional preceding checks are enforced if the operation is being attempted in the context of an EJB.

Recall from , "A First Look at EJB," that the EJB specifications cite seven players in the EJB arena. Each player has a role in the development, installation, and configuration of an EJB. In the following, you can read which player is responsible for what piece of EJB security.

EJB & JSP: Java On The Edge, Unlimited Edition

 by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

.

Reviewing Java Language Security Features

To understand what EJB offers in the way of security, it is necessary to briefly discuss the offerings of Java, the programming language and runtime environment. An interesting aspect of Java is that many of the language's features that help programmers write better applications also assist us in writing applications that are more secure. We will look at several of these features in this section.

The overriding property of the Java language and runtime is that Java manages most memory-related tasks. Not only does Java do the thankless work of managing memory, Java *forbids* the programmer from doing this thankless work. The engineers at Sun Microsystems realized that the vast majority of computing hacks come from clever folk who outfox the operating system by performing memory sleight of hand with pointers, pointer arithmetic, improper object casting, and extending arrays beyond declared bounds.

An example of a security language feature is the Java requirement that the programmer can cast objects to and from super and subclasses only. This casting requirement ensures that the programmer uses valid references to objects. Casting of objects outside a hierarchy chain, as allowed in other programming languages, could cause the runtime to overlap or corrupt memory as the runtime attempts to interpret dissimilar objects.

Java checks the appropriateness of an object cast at *compile* time. The compile-time check finds errors early and is a feature that enables programmers to write better code more quickly, without waiting for the runtime to crash and burn. In addition, Java also checks some objects cast at *runtime*. So, it's very difficult to outfox Java in the area of object casting.

Java does not allow a programmer to access memory by address. The Java programmer cannot *explicitly* use pointers. Of course, the Java programmer causes the JVM to use pointers internally each time an object is referenced. However, the object reference is not a pointer in the classic sense; the reference cannot be manipulated by performing arithmetic or bit-shifting. The only real manipulation of the reference itself (as opposed to the underlying object) is an object cast, and as we read in the previous paragraph, Java has object casts well secured.

The prohibition of explicit pointer use in Java stops a host of bugs and has the wholesome effect of making Java code secure against memory-related attacks.

Java does not allow a programmer to explicitly allocate and free memory chunks. You cannot find dangling pointers in a Java program. The runtime detects the presence of unused objects and sweeps them up automatically (garbage collection).

Try as you may, you cannot access an array outside its allocated extents. Again, the overriding theme is Java's control over memory. You may know that some programming languages enable array access past declared bounds and that such access can corrupt code and data.

The watchword here is *predictability*. Memory-related errors behave in unpredictable ways. Such unpredictability is the bane of anyone concerned with security. By stopping memory errors, Java enables programmers to improve the predictability of applications.

Speaking of predictability, Java also uses a structured exception-reporting mechanism. While Java does not force programmers to use this mechanism to handle all exceptions, nearly all of them take advantage of Java's exception-reporting mechanism. As a result, programmers improve the predictability of a program in the event of an error.

Out-of-the-box Java includes an impressive array of features that enable a programmer to engineer well-developed and secure systems. However, Java security encompasses more than the features of the language. The Java runtime, or JVM, also contains a few features that affect system security, which is discussed in the following.

Using the JVM Security Features

Of course, the Java language security features, and the Java language itself, cannot exist without the JVM, which has a few features that affect system security.

The JVM contains a *bytecode verifier*, which is the initial line of defense. In brief, the verifier scans the bytecode representing a Java software object and checks to determine if the bytecode has been altered before executing it.

For example, the following program produces bytecode with the output string imbedded in the file:

```
public class BytecodeTest {
   public static void main(String [] a ) {
      System.out.println("Tamper with my Bytecode") ;

   }
}
```

If you open the class file containing the bytecode with a text editor and change the word "with" to "WITH" and then run the bytecode, here's how JVM responds:

```
java.lang.ClassFormatError: BytecodeTest (Illegal constant pool type)
      at java.lang.ClassLoader.defineClass0(Native Method)
      at java.lang.ClassLoader.defineClass(ClassLoader.java:486)
      at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:111)
      at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
      at java.net.URLClassLoader.access$100(URLClassLoader.java:56)
      at java.net.URLClassLoader$1.run(URLClassLoader.java:195)
      at java.security.AccessController.doPrivileged(Native Method)
      at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
      at java.lang.ClassLoader.loadClass(ClassLoader.java:297)
      at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:286)
      at java.lang.ClassLoader.loadClass(ClassLoader.java:253)
      at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:313)
Exception in thread "main"
```

You can see that even an apparently benign change in the bytecode causes the JVM to react.

Another feature of the JVM, apparent in the above diagnostic, is the *class loader*. The class loader controls how and when classes are added to the JVM during execution. A JVM contains many class loaders. The stack trace above shows a `URLClassLoader`, a `SecureClassLoader`, and a `ClassLoader`. Class loaders are responsible for determining when classes may be added to the Java environment.

The JVM also makes use of the s*ecurity manager*. The security manager is responsible for controlling how classes use accessible interfaces. The security manager performs runtime checks and can issue security exceptions to veto potentially dangerous or unauthorized actions. The security manager works hand-in-hand with the various class loaders to check for access violations at runtime.

Up to this point in the chapter you've read about Java and JVM features that enable you to write more secure software, but there's more to tell regarding Java security features. Java contains a set of packages and tools that specifically address security, which is covered in the following section.

Looking at the Java Security Packages

The relevant Java packages that deal with security are the `java.security`, `java.security.cert`, `java.security.interfaces`, and `java.security.spec` packages, as well as a few classes in the base package, `java.lang`.

The `java.security` package provides the classes and interfaces for the security framework, including classes that implement a user-configurable, fine-grained access control security architecture. This package also supports the generation and storage of cryptographic public key pairs, as well as a number of exportable cryptographic operations including those for message digest and signature generation. Finally, this package provides classes that support signed/guarded objects and secure random number generation. The cryptographic and secure random number generator classes are provider-based. The implementations themselves may then be written by independent third-party vendors and plugged in seamlessly as needed. Therefore, application developers may take advantage of any number of provider-based implementations without having to add or rewrite code.

The `java.security.cert` package contains classes and interfaces for parsing and managing X.509 version 3 certificates. A *certificate* is a digitally signed statement from one entity, stating that the key (and some other information) of another entity has some specific value. The key is usually a public key, meaning that the key is freely available to any and all who need to receive information from the certificate sender. The X.509 standard is the accepted standard for digital certificates and is maintained by the *Public Key Infrastructure Group*.

The `java.security.interfaces` package contains interfaces for generating RSA (Rivest, Shamir, and Adleman AsymmetricCipher algorithm) keys as defined in the RSA Laboratory Technical Note PKCS#1, and DSA (Digital Signature Algorithm) keys. The ugly details of the algorithm and the specifications of the keys are defined in NIST's Federal Information Processing Standards Publication, number 186.

The `java.security.spec` package contains classes and interfaces for key specifications and algorithm parameter specifications. A key specification is a transparent representation of the key material that constitutes a key. A key may be specified in an algorithm-specific way, or in an algorithm-independent encoding format. This package contains key specifications for public and private keys in RSA, X.509, and other formats.

Security Features Available in the java.lang Package

The `java.lang` package contains classes that address security. These classes implement a security policy by using a security manager to check permissions in conjunction with a policy file. Let's take a look at how these pieces fit together.

The SecurityManager Class

A commonly used class from base Java is the `SecurityManager` class. This class and subclasses enable applications to implement a security policy. The `SecurityManager` class enables an application to determine, before performing a possibly unsafe or sensitive operation, what the operation is and whether it is being attempted in a security context, which would permit the operation to be performed. The application can allow or disallow the operation. When the operation is not allowed, the application may throw a security exception.

The `SecurityManager` class contains methods with names that begin with the word "check," such as `checkRead()` and `checkWrite()`, which check if the application can read from or write to a stream specified in the argument. These methods are called by various methods in the Java libraries before those methods perform certain potentially sensitive operations. For example, the invocation of a `checkRead()` method may resemble the code snippet shown below:

```
//Get a security manager to check permissions
try {
  SecurityManager security = System.getSecurityManager();
  if (security != null) {
      //Here's one permission…
      security.checkRead( myStringFileArg );
      //Do the operation…
      success = doRead( myStringFileArg ) ;
  }
catch (SecurityException sEx) { sEx.printStackTrace() } ;
```

The security manager allows the Java programmer to enforce a fine-grained level of access control by granting permissions on a method-by-method basis.

The class `RMISecurityManager`, a subclass of `SecurityManager`, is used to manage permissions for RMI applications. Use `RMISecurityManager` as you would use `SecurityManager`; get an instance of the `RMISecurityManager` and invoke the `checkXXX` methods.

Using Policy Files

The security manager examines a *policy file* to determine if a given permission is allowable. A policy file is a text file containing a list of allowable operations. Operations not explicitly included in the policy file are forbidden. The basic structure of a policy file is shown below:

```
grant [SignedBy "signer"] [, CodeBase "url"] {

   permission permission_class
      ["target"]  [,"action"]  [, SignedBy "signers"];
   //Additional permission statements may follow

}
```

In the preceding code snippet, if a read permission were not included in the policy file, the code would throw a `SecurityException`.

Assuming that you have a policy file called `mypolicyfile`, you associate the policy file with an application `MyApplication.class` as follows:

```
java -Djava.security.policy=mypolicyfile MyApplication
```

Tip   There are hundreds of Java methods that require permissions. Rather than provide an exhaustive list here, check out the following URL:
http://java.sun.com/j2se/1.3/docs/guide/security/permissions.html
Here, you can find the definitive list of methods and the permissions they require.

In addition to supplying classes as part of the core language, Sun provides several tools to help you navigate the security minefield.

Using Sun JDK Security Tools

The Java JDK contains tools for creating certificates and applying those certificates to your code (sign your code). The relevant tools are `keytool` and `jarsigner`.

The keytool Utility

The `keytool` utility, available since Java 1.2, enables you to:

- Manage a database of keys and names called a *keystore*

- Generate public and private key pairs

- Store these keys in encrypted format

- Assign a password to allow controlled access to the `keystore` database

- Import and export X.509 certificates

The `keytool` utility has numerous command line options. Here's an example of generating a pair of keys and an X.509 certificate with password-protected access. The options appear in italics.

```
keytool –genkey –alias mykeypair –keypass pairpass –keystore mykeystore –storepass mystorepass
```

This command accomplishes the following:

- Creates a `keystore` database called `mykeystore` in the current directory and assigns the password `storepass` to control access.

- Generates a pair of public and private keys and names the pair `mykeypair` and password-protects access to the keys by assigning the password `pairpass`.

- Issues a series of prompts for information required for an X.509 certificate, such as name, company name, address, and other identifying information.

- Generates an X.509 certificate that includes the key associated with the name `mykeypair`.

Note   Most Java IDEs provide a user interface to create certificates, sign code, and import or export certificates.

Tip   While certificates that you create with the `keytool` utility are valid, they may not be trusted by other parties in all instances. There are companies that provide what are known as "trusted" certificates that can be purchased on a subscription basis. For more check out VeriSign (http://www.verisign.com/).

The jarsigner Utility

The `jarsigner` tool is used to sign (apply a certificate to) jar files. The utility works with the keys and certificates generated in the `keytool` utility described earlier. Here's how you create a signed jar file using the information generated with the `keytool` command above.

- Create your application, applet, or Java software object

- Create a jar of the class file(s) using Sun's `jar` utility
  ```
  jar cvf MyJavaSoftware.jar MyJavaSoftware.class …
  ```

- *Sign* the jar file with the `jarsigner` tool; command options are in italics.
  ```
  jarsigner –keystore mykeystore
  –signedjar MySignedJavaSoftware.jar MyJavaSoftware.jar mykeypair
  ```

The `jarsigner` tool prompts for the `keystore` password (`mystorepass`) and the key password (`pairpass`). Notice that `jarsigner` need not overwrite your jar file.

Now that you have an idea of the security features available to the Java programmer, let's take a look at EJB security concerns and features.

System Security Overview

In this instance, the term *security* means exercising control over the resources of the computing environment, which involves *physical security* and *information security.*

Physical security deals with controlling access to buildings, rooms, and machines. Today, many companies hire security guards and require employees to display and use encoded identification badges to gain entrance into buildings, rooms, and parking garages. Companies may also install cameras and motion detectors, and they may monitor employee e-mail and, at times, telephone calls — all in the name of providing a more secure environment to conduct business. The high cost of such measures underscores the importance that companies place on physical security.

Information security deals with protecting access to and ensuring the integrity of information, particularly in an electronic form. Not only do security managers need to make sure that the wrong people do not access valuable information, but they need to make sure that the integrity of that information is never compromised!

Regarding information security, a search on http://www.excite.com/ for the string "information security" returns over 5 millions results. Even if a very small percentage of these Web sites addresses the issue of information security, that's still an absurd number of sites on the topic! Clearly, security is a hot topic and it's likely to gain importance in this ever-increasing networked world in which we live.

A search on http://www.amazon.com/ for "information security" lands 376 total matches. That's a large number of books, which, once again, illustrates the high level of interest software folk have in the topic of information security.

Enterprise JavaBeans' security features extend those of the Java programming language. Java, being a modern software technology, has several features that directly address information security. Next, you can read about some of Java's security features.

Chapter 16: EJB Security

Overview

Today, corporate information systems departments are growing more concerned with threats to system security from a Microsoft Word virus to a denial of service attack to a hack resulting in a loss or corruption of data. More than ever before, decision-makers must take steps to secure their systems. Fortunately, decision-makers who have made Java technologies, including EJB, an integral part of their shops are a step ahead of those who haven't.

This chapter discusses the security features available when deploying Enterprise JavaBeans, starting with a brief overview of security and then focusing on security features found in the Java programming language. This chapter also introduces the additional security features EJB brings to the table, featuring a discussion of the security roles of the players in the EJB arena, and concluding with a few thoughts on system security outside the Java environment.

First, let's take a look at the topic of system security and what the Java environment has to offer in the security arena.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

In this chapter you've learned about the use of entity beans to model data in an enterprise computing environment.
You've examined the life-cycle of entity beans and you've looked at the differences between container-managed
persistence and bean-managed persistence. Lastly, you've had the opportunity to create both CMP and BMP entity
beans in our employee bean examples. You should be able to use the information garnered from this chapter to
design and deploy entity beans to model data in future J2EE applications.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Implementing an Entity Bean

The listings that follow show code for an employee entity bean that models an employee in a company. This bean
provides the client with access to information about the employee, including salary, grade, and classification. The
bean also provides the client with methods to reassign or to promote the employee and to give the employee a raise.

In the following sections we'll look at the `remote` interface, the `home` interface, the primary key class, and, of course,
the bean class used to implement this employee entity bean.

Creating the remote Interface

shows the `remote` interface for the employee bean. The `remote` interface has the usual `get` and `set`
methods along with some business methods that represents the client's view of the enterprise bean. Notice that all the
`remote` interface methods throw an `EJBException`.

Listing 15-3: Remote interface for employee bean

```
package chapter15.employee ;

import java.util.* ;
import chapter15.employee.* ;
import javax.ejb.* ;

public interface Employee extends EJBObject {

  //Get methods for Employee Bean Properties
  String getEmployeeID()       throws EJBException ;
  String getEmployeeName()     throws EJBException ;
  Date   getHireDate()         throws EJBException ;
  int    getGrade()            throws EJBException ;
  String getClassification()   throws EJBException ;
  double getSalary()           throws EJBException ;
  String getDeptCode()         throws EJBException ;
  //Set methods for Employee Bean properties (some of
  //them, anyway)
  void   setGrade ( int newGrade )            throws EJBException ;
  void   setSalary( double newSalary )        throws EJBException ;
  void   setDeptCode( String newDeptCode )    throws EJBException ;
  //A few business methods
  void   giveEmployeeAPromotion( double percentIncrease,
                                 int    newGrade        ,
                                 String newClassification )
                                            throws EJBException ;
  void   giveEmployeeARaise( double percentIncrease )
```

```
                                                       throws EJBException ;
  void    reassign( String newDeptCode )              throws EJBException ;
}
```

Next, you take a look at the home interface for the BMP Employee bean.

Creating the home Interface

Listing 15-4 shows the employee bean `home` interface. The `home` interface contains `create`, `remove` and `finder` methods. Notice that `create` methods throw a `CreateException`, the `finder` method throws a `FinderException`, and the `remove` method throws a `RemoveException`.

Listing 15-4: The home interface

```
package chapter15.employee ;

import chapter15.employee.* ;
import javax.ejb.* ;
import java.util.* ;

public interface EmployeeHome extends EJBHome {

  public Employee create( String employeeID,
                          String firstName,
                          String lastName,
                          Date   hireDate,
                          int    grade,
                          String classification,
                          double salary,
                          String deptCode )
        throws EJBException, CreateException ;
  //This create() method is used to assign employees
  //to a default department, classification, grade and salary
  public Employee createTemp( String employeeID,
                          String firstName,
                          String lastName,
                          Date   hireDate )
        throws EJBException, CreateException ;
  //One finder method....
  public Employee findByPrimaryKey ( EmployeePK empPK )
        throws EJBException, FinderException ;
  //And a remove method
  public void remove( EmployeePK empPK )
        throws EJBException, RemoveException ;

}
```


Creating the Primary Key Class

Entity beans need a primary key for access. With BMP beans, the bean code must directly use the primary key to locate beans. Listing 15-5 shows the code for a key class whose objects have values that serve as keys to locate particular entity beans.

Listing 15-5: The primary key class

```
package chapter15.employee ;

import java.io.* ;

public class EmployeePK implements Serializable {
    public String employeeID ;

    public EmployeePK( String empID ) {
        employeeID = empID ;
    }

    //No Arg constructor for CMP beans
    public EmployeePK() {
    }

    public boolean equals( Object empID ) {
        boolean isEqual = false ;
        if ( empID instanceof EmployeePK )
            isEqual =
              ( ( (EmployeePK)empID).employeeID == employeeID ) ;

        return isEqual ;
    }

    public String toString() {
        return employeeID ;
    }

}
```

Notice that the primary key class satisfies the criteria cited earlier. The class is serializable, and overrides the Object.equals and Object.toString methods. It doesn't need to override Object.hashcode because the sole instance variable, employeeID, is a string.

Implementing the Bean Class as a BMP Bean

Listing 15-6 shows the bean class coded as a BMP bean. Notice the `ejbCreate` and `ejbFindByPrimaryKey` methods corresponding to the `create` and `findByPrimaryKey` methods in the `home` interface.

Listing 15-6: The bean class coded as a BMP bean

```
package chapter15.employee ;

import java.util.* ;
import chapter15.employee.* ;
import javax.ejb.* ;
import java.sql.* ;
import javax.sql.* ;
import javax.naming.* ;

public class EmployeeBean implements EntityBean {
  //Copy of EntityContext
  private EntityContext empECtx ;
  //Instance variables for Employee bean
  public    String    employeeID ;
  public    String    firstName ;
  public    String    lastName ;
  public    java.util.Date    hireDate ;
```

```java
public   int      grade    ;
public   String   classification ;
public   double   salary ;
public   String   deptCode ;
//Default instance variables for temporary employees
public   static   final   int      TEMPGRADE            = 30 ;
public   static   final   String   TEMPCLASSIFICATION  = "TEMP" ;
public   static   final   double   TEMPSALARY          = 2000.00;
public   static   final   String   TEMPDEPTCODE        = "T100";

//ejbCreate and ejbPostCreate methods
public EmployeePK ejbCreate( String employeeID,
                             String firstName,
                             String lastName,
                             java.util.Date   hireDate,
                             int    grade,
                             String classification,
                             double salary,
                             String deptCode )
        throws EJBException, CreateException {
  //Assign fields to instance variables
  this.employeeID     = employeeID ;
  this.firstName      = firstName ;
  this.lastName       = lastName ;
  this.hireDate       = hireDate ;
  this.grade          = grade ;
  this.classification = classification ;
  this.salary         = salary ;
  this.deptCode       = deptCode ;
  //Issue SQL to insert into database
  if ( employeeAdded(  employeeID, firstName, lastName,
                       hireDate, grade, classification,
                       salary, deptCode ) )

      return new EmployeePK( employeeID ) ;
  else
      throw new CreateException( "SQL For Employee " +
                                  employeeID + " failed." ) ;


}
public void ejbPostCreate( String employeeID,
                           String firstName,
                           String lastName,
                           java.util.Date   hireDate,
                           int    grade,
                           String classification,
                           double salary,
                           String deptCode )
          throws EJBException, CreateException {
   //Nothing going on here....
}
//This create() method is used to assign Temp employees
//to a default department, classification, grade and salary
public EmployeePK ejbCreateTemp( String employeeID,
                                 String firstName,
                                 String lastName,
                                 java.util.Date   hireDate )
        throws EJBException, CreateException{
   return ejbCreate( employeeID, firstName, lastName, hireDate,
                     TEMPGRADE,  TEMPCLASSIFICATION,
```

```java
                         TEMPSALARY, TEMPDEPTCODE ) ;

}
public void ejbPostCreate( String employeeID,
                           String firstName,
                           String lastName,
                           java.util.Date   hireDate )
         throws EJBException, CreateException {
   //Nothing going on here.....
}
//ejbFind method follows
EmployeePK ejbFindByPrimaryKey( EmployeePK empPK )
          throws EJBException, FinderException {
  PreparedStatement       selectSQL  = null ;
  Connection              selectConn = null ;
  ResultSet               findRS     = null ;
  String                  selectSQLString =
          "select employeeID from emptable where employeeID = ?" ;
   try {
      selectConn = getConnection() ;
      selectSQL = selectConn.prepareStatement( selectSQLString ) ;
      selectSQL.setString( 1, empID.toString() ) ;
      findRS     = selectSQL.executeQuery() ;

      if ( findRS == null )
         throw new EJBException( "Locate of employee " +
                  empID.toString() + " failed." ) ;
      else {
         findRS.next() ;
         employeeID     = findRS.getString( "employeeID" ) ;

      }
   }
   catch( Exception exc ) {
       throw new EJBException ( exc.toString() ) ;
   }
   finally {
     //Close up shop
     try {
        if (selectSQL != null )
            selectSQL.close() ;
          if ( loadRS != null )
             loadRS.close() ;
        if ( selectConn != null )
            selectConn.close() ;
     }
     catch ( Exception exc ) {} ;

}
//Business methods, including get and set methods
String getEmployeeID() throws EJBException {
   return employeeID ;
}
String getEmployeeName() throws EJBException {
   return firstName + " " + lastName ;
}
java.util.Date   getHireDate() throws EJBException {
   return hireDate ;
}
int getGrade() throws EJBException {
   return grade ;
```

```java
}
String getClassification() throws EJBException {
   return classification ;
}
double getSalary() throws EJBException {
   return salary ;
}
String getDeptCode() throws EJBException {
   return deptCode ;
}
void setGrade ( int newGrade ) throws EJBException {
   grade = newGrade ;
}
void setSalary( double newSalary ) throws EJBException {
   salary = newSalary ;
}
void setDeptCode( String newDeptCode ) throws EJBException {
   deptCode = newDeptCode ;
}
//A few business methods
void giveEmployeeAPromotion( double percentIncrease,
                            int    newGrade       ,
                            String newClassification )
      throws EJBException {
   salary          = salary * ( 1.0 + percentIncrease ) ;
   grade           = newGrade ;
   classification  = newClassification ;
}
void giveEmployeeARaise( double percentIncrease )
      throws EJBException {
   salary          = salary * ( 1.0 + percentIncrease ) ;
}
void reassign( String newDeptCode )  throws EJBException {
    deptCode = newDeptCode ;

}
//Required methods for EntityBean declared here
//Save the entity context
public void setEntityContext( EntityContext ectx ) {
      empECtx = ectx ;
}
public void unsetEntityContext( ) throws EJBException {
    empECtx = null ;
}
public void ejbActivate() throws EJBException {
  //Nothing going on here....
}
public void ejbPassivate() throws EJBException {
  //Nothing going on here....
}
public void ejbLoad()      throws EJBException {
  //Load the correct database data to correspond with
  //this bean instance
  EmployeePK  thisPK     = (EmployeePK) empECtx.getPrimaryKey() ;
  String      employeeID = thisPK.employeeID ;
  ResultSet   loadRS     = null ;

  String      selectSQLString =
               "select * from empTable where employeeID = ?" ;

   PreparedStatement      selectSQL  = null ;
```

```
      Connection              selectConn = null ;
      try {
         selectConn = getConnection() ;
         selectSQL = selectConn.prepareStatement( selectSQLString ) ;
         selectSQL.setString( 1, employeeID ) ;
         loadRS     = selectSQL.executeQuery() ;

         if ( loadRS == null )
             throw new EJBException( "Load of Employee " +
                                     employeeID + " failed." ) ;
         else {
            loadRS.next() ;
          employeeID = loadRS.getString( "employeeID" ) ;
          firstName = loadRS.getString( "firstName"  ) ;
          lastName = loadRS.getString( "lastName"    ) ;
          hireDate = (java.util.Date) loadRS.getDate( "hireDate" ) ;
          grade = loadRS.getInt( "grade" ) ;
          classification = loadRS.getString( "classification" ) ;
          salary = loadRS.getDouble( "salary" ) ;
          deptCode = loadRS.getString( "deptCode" ) ;
         }
      }
      catch( Exception exc ) {
          throw new EJBException ( exc.toString() ) ;
      }
      finally {
        //Close up shop
        try {
           if (selectSQL != null )
                 selectSQL.close() ;
            if ( loadRS != null )
               loadRS.close() ;
           if ( selectConn != null )
                 selectConn.close() ;
         }
          catch ( Exception exc ) {} ;
       }

   }
   public void ejbStore() throws EJBException {
       PreparedStatement updateSQL  = null ;
       Connection updateConn = null ;
       String updateSQLString =
           "update emptable set employeeID = ?, firstName = ?, "  +
           "lastName   = ?, hireDate  = ?, "   +
           "grade      = ?, classification = ?, " +
           "salary     = ?, deptCode  = ? ) " ;
       //Issue SQL to update the database
       try {
         //First, get a database connection....
         updateConn = getConnection() ;
         //Second, prepare the SQL statement
         updateSQL = updateConn.prepareStatement( updateSQLString ) ;
         updateSQL.setString( 1, employeeID ) ;
         updateSQL.setString( 2, firstName ) ;
         updateSQL.setString( 3, lastName ) ;
         updateSQL.setDate  ( 4, (java.sql.Date) hireDate ) ;
         updateSQL.setInt   ( 5, grade ) ;
         updateSQL.setString( 6, classification ) ;
         updateSQL.setDouble( 7, salary ) ;
         updateSQL.setString( 8, deptCode ) ;
```

```java
          //Third, issue the SQL insert statement.
          //Save results as boolean for return
          if ( updateSQL.executeUpdate() != 1 )
             throw new EJBException("Update Failed during ejbStore");
       }
       catch ( Exception exc ) {
             throw new EJBException( exc.toString() ) ;
       }
       finally {
          //Close up shop
          try {
                if (updateSQL != null )
                    updateSQL.close() ;
                if (updateSQL != null )
                 updateSQL.close() ;
          }
          catch ( Exception exc ) {} ;
       }

}
public void ejbRemove()     throws EJBException {
      EmployeePK thisPK = (EmployeePK) empECtx.getPrimaryKey() ;
      String employeeID = thisPK.employeeID ;

      String deleteSQLString =
                  "delete from empTable where employeeID = ?" ;
      PreparedStatement deleteSQL  = null ;
      Connection deleteConn = null ;
      try {
       deleteConn = getConnection() ;
       deleteSQL = deleteConn.prepareStatement( deleteSQLString ) ;
       deleteSQL.setString( 1, employeeID ) ;
       if ( deleteSQL.executeUpdate() != 1 )
           throw new EJBException( "Delete of Employee " +
                                    employeeID + " failed." ) ;
      }
      catch( Exception exc ) {
         throw new EJBException ( exc.toString() ) ;
      }
      finally {
       //Close up shop
         try {
            if (deleteSQL != null )
                deleteSQL.close() ;
            if (deleteConn != null )
                 deleteConn.close() ;
         }
        catch ( Exception exc ) {} ;
      }


}
//Utility method to establish a database connection
private Connection getConnection( )
        throws SQLException, NamingException {
      String dbURL = "java:comp/env/empdb" ;

      InitialContext initCtx = new InitialContext();
      DataSource empDB = (DataSource) initCtx.lookup( dbURL );
      Connection con = empDB.getConnection();

    return con ;
```

```java
      }
   //Utility method to insert an employee into the database
   private boolean employeeAdded( String employeeID,
                                  String firstName,
                                  String lastName,
                                  java.util.Date   hireDate,
                                  int     grade,
                                  String classification,
                                  double salary,
                                  String deptCode )
         throws EJBException {

      boolean               insertSuccessful = false ;

      PreparedStatement   insertSQL  = null ;
      Connection           insertConn = null ;
      String               insertSQLString =
            "insert into emptable " +
            " (employeeID,firstName,lastName,hireDate, "  +
            " grade, classification, salary, deptCode ) " +
            "values (?,?,?,?,?,?,?,?)" ;
      //Issue SQL to insert into database
      try {
        //First, get a database connection....
        insertConn = getConnection() ;
        //Second, prepare the SQL statement
        insertSQL = insertConn.prepareStatement(insertSQLString) ;
        insertSQL.setString( 1, employeeID ) ;
        insertSQL.setString( 2, firstName ) ;
        insertSQL.setString( 3, lastName ) ;
        insertSQL.setDate  ( 4, (java.sql.Date) hireDate ) ;
        insertSQL.setInt   ( 5, grade ) ;
        insertSQL.setString( 6, classification ) ;
        insertSQL.setDouble( 7, salary ) ;
        insertSQL.setString( 8, deptCode ) ;
        //Third, issue the SQL insert statement.
        //Save results as boolean for return
        insertSuccessful = ( insertSQL.executeUpdate() == 1 ) ;

      }
      catch ( Exception exc ) {
           throw new EJBException( exc.toString() ) ;
      }
      finally {
         //Close up shop
         try {
           if (insertSQL != null )
               insertSQL.close() ;
           if (insertConn != null )
              insertConn.close() ;
         }
        catch ( Exception exc ) {} ;

          return insertSuccessful ;
      }

   }

}
```

Because this bean class is for a bean-managed entity bean, it contains plenty of SQL code that performs the bean's major duties. The ejbCreate methods require the use of an SQL INSERT statement. The ejbFindByPrimaryKey and the ejbLoad methods require an SQL SELECT statement. The ejbRemove method requires the use of an SQL DELETE statement. The ejbStore method requires an SQL UPDATE statement.

The database connection is acquired by using the JNDI naming context, as in the example of a session bean shown in Chapter 14, "EJB Session Beans." Of course, the usual spate of `accessor` and `mutator` methods is also present, along with business methods.

Let's now take a look at the Employee bean coded as a container-managed bean.

Implementing the Bean Class as a CMP Bean

1.

The differences between the BMP and CMP bean versions are startling. You must have noticed all the SQL in the BMP bean class. Please notice the *absence* of SQL in the CMP bean class. Listing 15-7 shows the bean coded as a CMP bean.

Listing 15-7: The bean class as a CMP bean

```
package chapter15.employee ;

import java.util.* ;
import chapter15.employee.* ;
import javax.ejb.* ;

public class EmployeeBeanCMP implements EntityBean {
  //Copy of EntityContext
  private EntityContext empECtx ;
  //Instance variables for Employee bean
  public   String    employeeID ;
  public   String    firstName ;
  public   String    lastName ;
  public   java.util.Date    hireDate ;
  public   int       grade    ;
  public   String    classification ;
  public   double    salary ;
  public   String    deptCode ;
  //Default instance variables for temporary employees
  public   static   final   int      TEMPGRADE             = 30 ;
  public   static   final   String   TEMPCLASSIFICATION = "TEMP" ;
  public   static   final   double   TEMPSALARY            = 2000.00;
  public   static   final   String   TEMPDEPTCODE          = "T100";

  //ejbCreate and ejbPostCreate methods
  public void ejbCreate( String employeeID,
                         String firstName,
                         String lastName,
                         java.util.Date    hireDate,
                         int     grade,
                         String classification,
                         double salary,
                         String deptCode )
          throws EJBException, CreateException {
    //Assign fields to instance variables
    this.employeeID     = employeeID ;
    this.firstName      = firstName ;
    this.lastName       = lastName ;
```

```java
   this.hireDate       = hireDate ;
   this.grade          = grade ;
   this.classification = classification ;
   this.salary         = salary ;
   this.deptCode       = deptCode ;

}
public void ejbPostCreate( String employeeID,
                           String firstName,
                           String lastName,
                           java.util.Date   hireDate,
                           int     grade,
                           String classification,
                           double salary,
                           String deptCode )
        throws EJBException, CreateException {
   //Nothing going on here....
}
//This create() method is used to assign Temp employees
//to a default department, classification, grade and salary
public void ejbCreateTemp( String employeeID,
                           String firstName,
                           String lastName,
                           java.util.Date   hireDate )
     throws EJBException, CreateException {
     ejbCreate( employeeID, firstName, lastName, hireDate,
                TEMPGRADE,  TEMPCLASSIFICATION,
                TEMPSALARY, TEMPDEPTCODE ) ;

}
public void ejbPostCreate( String employeeID,
                           String firstName,
                           String lastName,
                           java.util.Date   hireDate )
        throws EJBException, CreateException {
   //Nothing going on here.....
}
//Business methods, including get and set methods
String getEmployeeID()       throws EJBException {
   return employeeID ;
}
String getEmployeeName()     throws EJBException {
   return firstName + " " + lastName ;
}
java.util.Date   getHireDate()        throws EJBException {
   return hireDate ;
}
int    getGrade()            throws EJBException {
   return grade ;
}
String getClassification()   throws EJBException {
   return classification ;
}
double getSalary()           throws EJBException {
   return salary ;
}
String getDeptCode()         throws EJBException {
   return deptCode ;
}
void   setGrade ( int newGrade )             throws EJBException {
   grade = newGrade ;
```

```
  }
  void    setSalary( double newSalary )        throws EJBException {
     salary = newSalary ;
  }
  void    setDeptCode( String newDeptCode )    throws EJBException {
     deptCode = newDeptCode ;
  }
  //A few business methods
  void    giveEmployeeAPromotion( double percentIncrease,
                                  int     newGrade       ,
                                  String newClassification )
        throws EJBException {
     salary            = salary * ( 1.0 + percentIncrease ) ;
     grade             = newGrade ;
     classification    = newClassification ;
  }
  void    giveEmployeeARaise( double percentIncrease )
        throws EJBException {
     salary            = salary * ( 1.0 + percentIncrease ) ;
  }
  void    reassign( String newDeptCode )  throws EJBException {
       deptCode = newDeptCode ;

  }
  //Required methods for EntityBean declared here
  //Save the entity context
  public void setEntityContext( EntityContext ectx ) {
        empECtx = ectx ;
  }
  public void unsetEntityContext( ) throws EJBException {
       empECtx = null ;
  }
  public void ejbActivate() throws EJBException {
     //Nothing going on here....
  }
  public void ejbPassivate() throws EJBException {
     //Nothing going on here....
  }
  public void ejbLoad()      throws EJBException {

  }
  public void ejbStore()     throws EJBException {

  }
  public void ejbRemove()    throws EJBException {

  }

}
```

With the container managing the bean instance-to-database relationship, the bean code doesn't need to issue any SQL. With the need for SQL gone, so too is the need for utility methods that get database connections and the like.

Finder methods are absent in CMP beans, too. The container generates primary key values and uses these values to access and manipulate beans and the underlying data.

Notice that the `ejbCreate` methods return a void, not an instance of the primary key class. The container tends to many of the mundane details of creating and accessing database data.

Just how does the container know what to do? All the required information is kept in the deployment descriptor.

Writing the Deployment Descriptor for a CMP bean

shows a piece of the deployment descriptor instructing the container to handle database interactions with instance variables of the employee bean.

Listing 15-8: Deployment descriptor naming container-managed fields

```
<entity>
  <description>
   The Employee entity bean encapsulates access to the
   employee records. The deployer will use container-managed
   persistence to integrate the entity bean with the back-end
   system managing the employee records.
  </description>
  <ejb-name>Employee</ejb-name>
  <home>chapter15.employee.EmployeeHome</home>
  <remote>chapter15.employee.Employee</remote>
  <ejb-class>chapter15.employee.EmployeeBeanCMP</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>chapter15.employee.EmployeeID</prim-key-class>
  <reentrant>True</reentrant>
  <cmp-field><field-name>employeeID</field-name></cmp-field>
  <cmp-field><field-name>firstName</field-name></cmp-field>
  <cmp-field><field-name>lastName</field-name></cmp-field>
  <cmp-field><field-name>hireDate</field-name></cmp-field>
  <cmp-field><field-name>grade</field-name></cmp-field>
  <cmp-field><field-name>classification</field-name></cmp-field>
  <cmp-field><field-name>salary</field-name></cmp-field>
  <cmp-field><field-name>deptCode</field-name></cmp-field>
</entity>
```

The container is smart enough to generate the correct SQL (or whatever language is used by the database) for creates, deletes, and most selects and updates, based on the information contained in the deployment descriptor.

.

Looking at the Life Cycle of an Entity Bean

In this section, I describe the life cycle of an entity bean and discuss some container-invoked methods that execute as
the bean progresses through its life cycle.

Examining the Possible Entity Bean States

An entity bean exists in one of three states:

- **Does Not Exist:** The bean has not been created yet.

- **Pooled:** An entity bean in the pooled state exists but has no identity. In other words, the bean is ready to rock-and-
  roll but cannot be identified by a client.

- **Ready:** An entity bean in the ready state has identity, or can be identified by a client. In other words, a ready state
  bean can perform useful work on behalf of a client.

Figure 15-1 shows the methods invoked by the client and the container that causes the entity bean to transition from
one state to another.
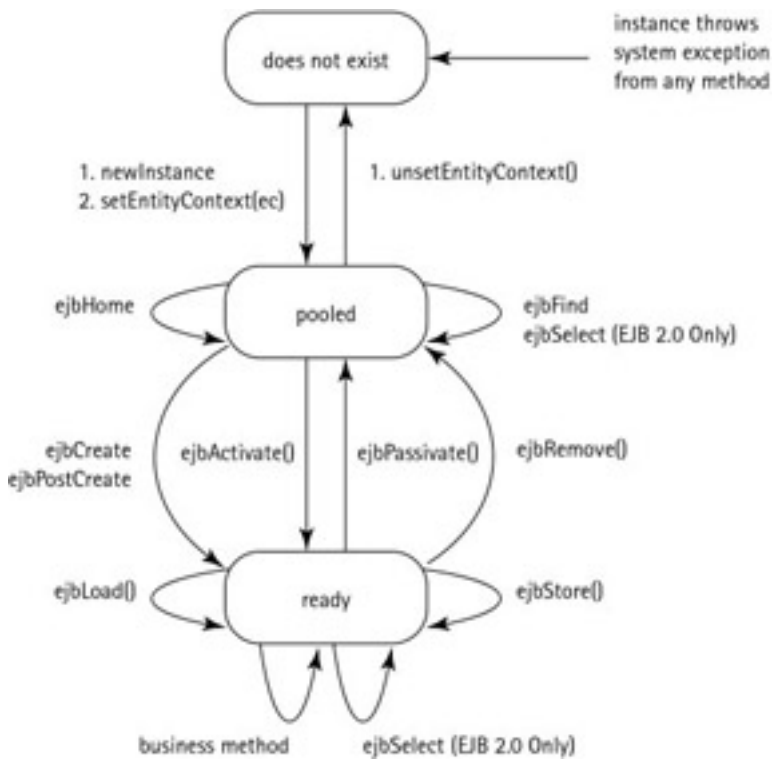
Figure 15-1: Entity bean life cycle

The following sections describe the life cycle of an entity bean instance.

Instantiating an Entity Bean

An entity bean instance's life starts when the container creates the instance using the `Class` method `newInstance`. The container calls the default constructor for your bean class.

Please realize that an EJB container is under no obligation to create a new instance of your bean class for every client. Actually, most containers try to create as few beans as possible because object creation is not cheap. The thrust behind the pooling strategy is to dole out a few bean instances to multiple clients. The same entity bean instance usually services multiple clients. Never develop EJB applications under the assumption that the container will create an entity bean at any particular time or situation.

The container then invokes the `setEntityContext` method to pass the instance a reference to the `EntityContext` interface. The `EntityContext` interface allows the instance to invoke services provided by the container and to obtain the information about the caller of a client-invoked method.

In Chapter 13, "EJB Basics," I discussed the methods in the `EJBContext` interface and in Chapter 14, "EJB Session Beans," I discussed the methods in the `SessionContext` subinterface. Let's take a look at the subinterface of `EJBContext` for entity beans named, appropriately enough, `EntityContext`.

The EntityContext Interface

The `EntityContext` interface provides a bean instance with access to its container-provided runtime context. The container passes the `EntityContext` interface to an entity bean instance after the instance has been created.

The `EntityContext` interface remains associated with the instance for the lifetime of the instance. Note that the information that the instance obtains using the `EntityContext` interface (such as the result of the `getPrimaryKey` method) may change, as the container assigns the instance to different EJB objects during the instance's life cycle.

Listing 15-2 shows the `EntityContext` interface.

Listing 15-2: The EntityContext interface

```
public interface javax.ejb.EntityContext
      extends javax.ejb.EJBContext {

   EJBObject getEJBObject()
        throws java.lang.IllegalStateException ;

   Object getPrimaryKey()
         throws java.lang.IllegalStateException ;
}
```

The `getEJBObject` method returns a reference to the EJB object associated with the bean instance. Hence, your bean can invoke this method only when your bean has identity, or is in the *ready* state. Entity beans may call `getEJBObject` from their `ejbActivate`, `ejbPassivate`, `ejbPostCreate`, `ejbRemove`, `ejbLoad`, `ejbStore`, and business methods.

If you need to code a method that passes a reference of the bean instance to another method, you should use the reference returned by `getEJBObject`. You may think you can pass `this` to refer to the current bean instance. However, recall that the work done by beans is done by the associated EJB object, through the container. You may get unpredictable results by passing `this` around from method to method.

The `getPrimaryKey` method gets the primary key of the EJB object that is currently associated with this instance. The same restrictions for invoking this method as those for invoking `getEJBObject` apply — the bean must be associated with an EJB object.

It is important to save a reference to the entity context. If you don't, you cannot legally invoke the `getPrimaryKey` or `getEJBObject` methods because when your bean is in a legal state to do so, the `setEntityContext` method has already executed and the entity context will go out of scope.

Moving an Entity Bean to the Pool

Once the entity bean is instantiated, it enters the pool of available instances. Each entity bean has its own pool. While the instance is in the available pool, the instance is not associated with any particular entity object identity. All instances in the pool are considered equivalent, and therefore any instance can be assigned by the container to any entity object identity at the transition to the ready state.

While the instance is in the pooled state, the container may use the instance to execute any of the entity bean's `finder` or home methods (shown as `ejbFind` and `ejbHome` in the Figure 15-1, respectively). The instance does *not* move to the ready state during the execution of a `finder` or a `home` method.

Transitioning to the Ready State

An instance transitions from the pooled state to the ready state when the container selects that instance to service a client call to an entity object. There are two possible transitions from the pooled to the ready state: through the `ejbCreate` and `ejbPostCreate` methods, or through the `ejbActivate` method.

The container invokes the `ejbCreate` and `ejbPostCreate` methods when the instance is assigned to an entity object during entity object creation (i.e., when the client invokes a `create` method on the entity bean's home object). The container invokes the `ejbActivate` method on an instance when an instance needs to be activated to service an invocation on an existing entity object. This occurs because there is no suitable instance in the ready state to

service the client's call.

When an entity bean instance is in the ready state, the instance is associated with a specific entity object identity. While the instance is in the ready state, the container can invoke the `ejbLoad` and `ejbStore` methods zero or more times. A business method can be invoked on the instance zero or more times. Invocations of the `ejbLoad` and `ejbStore` methods can be arbitrarily mixed with invocations of business methods.

The purpose of the `ejbLoad` and `ejbStore` methods is to synchronize the state of the instance with the state of the entity in the underlying data source. The container can invoke these methods whenever it determines a need to synchronize the instance's state.

The container can choose to passivate an entity bean instance within a transaction. To passivate an instance, the container first invokes the `ejbStore` method to allow the instance to synchronize the database state with the instance's state, and then the container invokes the `ejbPassivate` method to return the instance to the pooled state.

Returning to the Pooled State

Eventually, the container moves the bean instance from the ready state to the pooled state. There are three possible transitions from the ready to the pooled state:

- After execution of the `ejbPassivate` method.

- After the execution of the `ejbRemove` method. Contrary to the method name, `ejbRemove` does not delete or destroy the bean instance. After the container invokes `ejbRemove`, the bean instance retreats back into the instance pool.

- After a transaction rollback for `ejbCreate`, `ejbPostCreate`,or `ejbRemove` (not shown in Figure 15-1).

The container invokes the `ejbPassivate` method when the container wants to disassociate the instance from the entity object identity without removing the entity object. The container invokes the `ejbRemove` method when the container is removing the entity object (i.e., when the client invokes the `remove` method on the entity object's `remote` interface, or on the entity bean's `home` interface). If `ejbCreate`, `ejbPostCreate`, or `ejbRemove` is called and the transaction rolls back, the container transitions the bean instance to the pooled state. Chapter 17, "EJB and Transaction Management," has more information about transactions and Enterprise JavaBeans.

When the instance is put back into the pool, it is no longer associated with an entity object identity. The container can assign the instance to any entity object within the same entity bean home.

Killing an Entity Bean

You can remove an instance in the pool by calling the `unsetEntityContext` method on the instance. Recall that the container executing `ejbRemove` does not delete the bean instance.
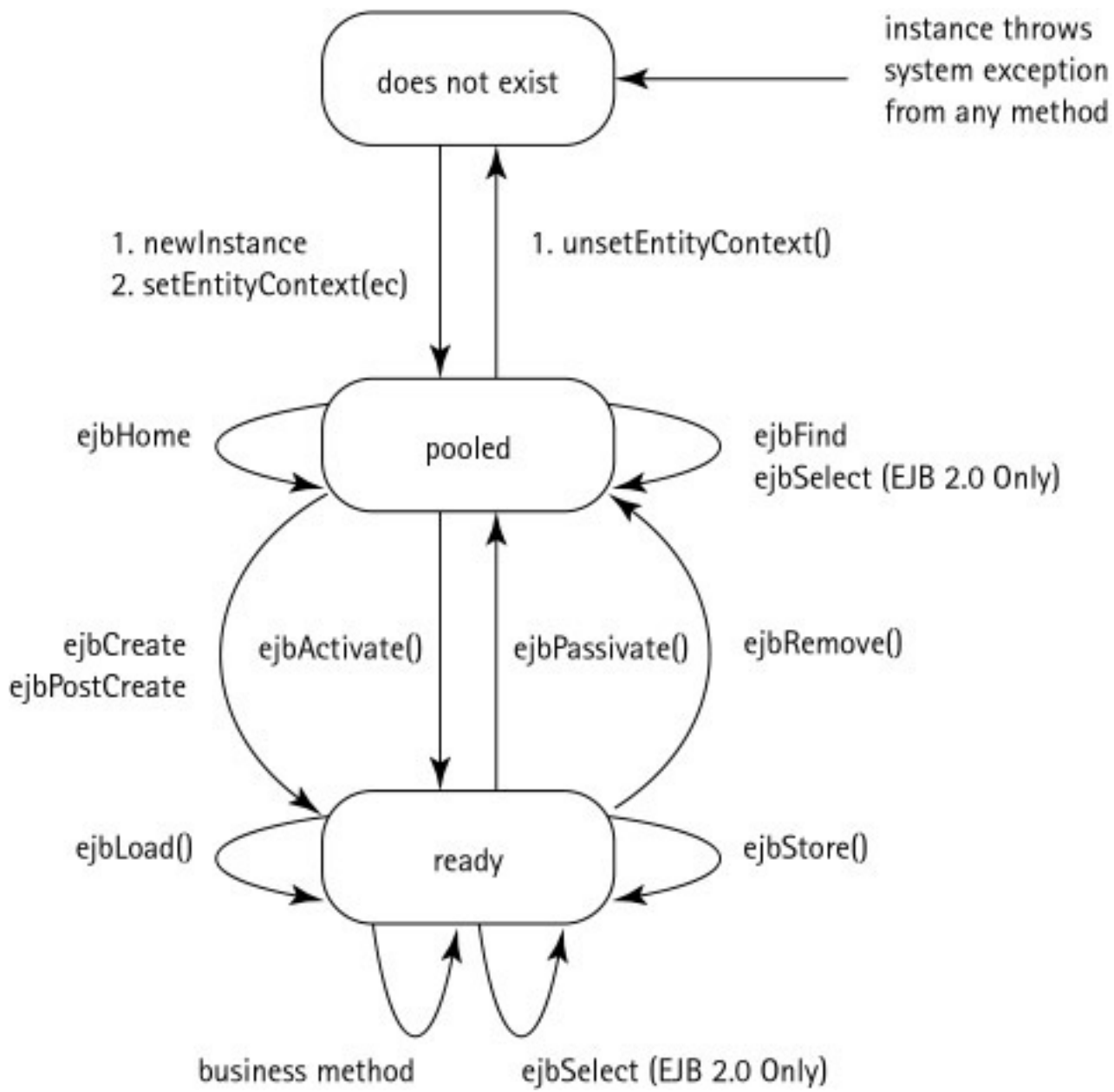
EJB & JSP: Java On The Edge, Unlimited Edition

 by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

.

Chapter 13: EJB Contexts and Containers

The previous chapter described the elements of session and entity beans. You've also read about the bean classes and the home and remote interfaces. In this chapter, you read about the methods available from the `EJBContext` interface. The chapter closes with some comments on the relationship, or *contract*, between the enterprise bean and the EJB container.

Exploring the EJB Context

You know of the two enterprise bean types defined by the EJB 1.1 specification — entity and session beans. You may recall that EJB 2.0 defines a third bean type — the message-driven bean. All these bean types (and others that may be defined in future EJB releases) must provide information about their status and the container in which they reside at runtime.

Enterprise JavaBeans provide developers with several interfaces to gather status or container information. Each bean type supports a *context* interface that supplies this information. The context interfaces for each bean type are:

- The `EntityContext` interface for entity beans. See Chapter 15, "EJB Entity Beans," for more details on the `EntityContext` interface.

- The `SessionContext` interface for session beans. See Chapter 14, "EJB Session Beans," for more details on the `SessionContext` interface.

- The `MessageDrivenContext` interface for message-driven beans (EJB 2.0 only). See Chapter 19, "The Proposed EJB 2.0 Specification," for more details on the `MessageDrivenContext` interface.

The preceding three context interfaces have a parent interface called the `EJBContext` interface. This section discusses the methods available to bean developers from the `EJBContext` interface.

Various Java technologies that utilize containers support the use of a context object that holds, and allows access to, information about the container. Servlets, for example, support a `ServletContext` object that serves much the same purpose as the EJB context objects. However, rarely does an enterprise bean use an object derived from the `EJBContext` interface. More often, the bean uses a context object derived from one of the three child interfaces listed previously. Because of Java's inheritance mechanism, all the methods available in `EJBContext` are available to objects derived from all of the three child interfaces.

Before discussing the methods available through the `EJBContext` interface, let's see how to gain access to a context object.

Gaining Access to a Context Object

With EJBs (and other Java container type technologies), context objects are available at certain times in the bean's life cycle. Chapters 14 and 15 discuss enterprise bean life cycles, so I won't cover the life cycle here. An enterprise bean context object becomes available to the bean when the container creates an instance of the bean.

The fact that the context object is available to your bean does not mean that your bean has automatic access to the context object. You need to write code to save a reference to the context object. The code snippet that follows saves a reference to the EntityContext, which is the context available to entity beans:

```
public MyEntityBean implements EntityBean {
   //Declared outside any method
   private myEntityContext ;

   public void setEntityContext( EntityContext ectx )
             throws RemoteException {
      myEntityContext = ectx ;
   }
   //Rest of bean methods follow
}
```

Now the entity context is available to any bean method through the reference myEntityContext. Methods similar to setEntityContext are available for session beans and message-driven beans. Chapter 15 discusses the setEntityContext method in more detail.

Using EJBContext Methods

Listing 13-1 shows the methods that the EJBContext interface provides. All enterprise bean types have access to these methods. We will discuss each of these methods in turn in the rest of this section.

Listing 13-1: The EJBContext interface

```
public interface javax.ejb.EJBContext {

  public java.security.Principal   getCallerPrincipal() ;

  public boolean                   isCallerInRole(String roleName) ;

  public boolean                   getRollbackOnly()
        throws java.lang.IllegalStateException ;

  public void                      setRollbackOnly()
        throws java.lang.IllegalStateException ;

  public javax.jts.UserTransaction getUserTransaction()
        throws java.lang.IllegalStateException ;

  public javax.ejb.EJBHome         getEJBHome() ;
}
```

Note   EJB release 1.0 defined three methods, since deprecated. The methods and their replacements are: public java.security.Identity getCallerIdentity, replaced by public java.security.Principal getCallerPrincipal; public java.util.Properties getEnvironment, replaced by using JNDI to get bean environment properties; and public boolean isCallerInRole(java.security.Identity role) replaced by public boolean isCallerInRole(String role).

The methods from EJBContext fall into three categories: security, transaction support, and accessing the home

object. Let's examine the methods in these three categories.

EJBContext Security Methods

As a general rule, the EJB container should handle security issues in a manner transparent to the execution of your beans. However, rules are made to be broken. If you have a real need to access security information about the caller of your bean methods, the `EJBContext` interface provides two methods that deal with EJB security: `getCallerPrincipal` and `isCallerInRole`.

The `getCallerPrincipal` and `isCallerInRole` methods can be invoked only in the enterprise bean's business methods for which the container has a client security context. If these methods are invoked when no security context exists, they should throw the `java.lang.IllegalStateException` runtime exception.

Let's examine each of these security methods in turn in the following two sections. In Chapter 16, "EJB Security," we will return to address EJB security issues in more detail.

The getCallerPrincipal Method

The `getCallerPrincipal` method returns an object of a class that implements the `java.security.Principal` interface, which we'll call a *principal*. A principal can be an individual, a corporation, a program thread, or anything that can have an identity.

An enterprise bean can invoke the `getCallerPrincipal` method to obtain a `java.security.Principal` interface representing the current caller. The enterprise bean can then obtain the distinguished name of the caller principal using the `getName` method of the `java.security.Principal` interface.

The meaning of the current caller, the Java class that implements the `java.security.Principal` interface, and the realm of the principals returned by the `getCallerPrincipal` method depend on the operational environment and the configuration of the application.

The isCallerInRole Method

The main purpose of the `isCallerInRole` method is to allow the bean provider to code the security checks that cannot be easily defined in the deployment descriptor using method permissions. Such a check might impose a role-based limit on a request, or it might depend on information stored in the database.

The enterprise bean code uses the `isCallerInRole` method to test whether the current caller has been assigned to a given security role. The application assembler typically defines security roles in the deployment descriptor. The deployer assigns the security roles to principals or principal groups that exist in the operational environment.

Listing 13-2 shows a code snippet that checks if the method caller has a specific security role named `payroll`. If not, the code throws a `SecurityException`. If so, the caller presumably has clearance to perform updates to the data.

Listing 13-2: Invoking isCallerInRole

```
public class PayrollBean implements EntityBean {
   //We use EntityContext because the PayrollBean is an
   //entity bean
   EntityContext ejbContext;
   //Typical EJB business method
   public void updateEmployeeInfo(EmplInfo info) {
      oldInfo = ... read from database;
      //All users who do not have the security role
      //of 'payroll' are booted from the method.
```

```
        if (info.salary != oldInfo.salary &&
            !ejbContext.isCallerInRole("payroll")) {
                throw
                new SecurityException("You cannot update salaries!");
        }
```

As previously mentioned, security roles are declared in the bean's deployment descriptor. Listing 13-3 shows a piece of a deployment descriptor that allows current callers with a role of `payroll` to invoke the `updateEmployeeInfo` method.

Listing 13-3: Deployment descriptor showing security role

```
<security-role>
     <description>
          This role represents Human Resource Staff
          authorized to perform payroll operations
     </description>
     <role-name>payroll</role-name>
  </security-role>

  <method-permission>
    <role-name>payroll</role-name>
    <method>
        <ejb-name>PayrollBean</ejb-name>
        <method-name>updateEmployeeInfo</method-name>
        <!- - Other methods may follow --!>
    </method>
  </method-permission>
```

It bears repeating that, as much as possible, the bean developer should allow the container to manage security matters. After all, if you don't want to take advantage of the services provided by the Enterprise JavaBeans container, why are you developing EJBs?

The EJBContext Methods Dealing with Transactions

The topic of transactions is discussed in detail in Chapter 17, "EJB and Transaction Management." However, a few words on the `EJBContext` methods dealing with transactions are in order. The three methods provided by the context interface are `getRollbackOnly`, `setRollbackOnly`, and `getUserTransaction`.

The getRollbackOnly and setRollbackOnly Methods

Code the `getRollbackOnly` method to test if the transaction has been marked for rollback only. An enterprise bean instance can use this operation to test, after an exception has been caught, whether it is fruitless to continue computation on behalf of the current transaction.

Code the `setRollbackOnly` method to mark the current transaction for rollback and the transaction will become permanently marked for rollback. A transaction marked for rollback can never commit, by the way. Typically, an enterprise bean marks a transaction for rollback to protect data integrity before throwing an application exception because application exceptions do not automatically cause the container to roll back the transaction.

As you'll learn in Chapter 17, "EJB and Transaction Management," you can defer many of the details of transaction management to the container. Beans that have their transaction state managed by the container are called (surprise) *container-managed transaction beans*. Those beans that contain code to manage the nuts and bolts of transactions

are called *bean-managed transaction beans*.

I draw the distinction between bean-managed and container-managed beans in this section because only enterprise beans with container-managed transactions are allowed to use `getRollbackOnly` and `setRollbackOnly`. However, bean-managed beans may invoke the `getStatus` method of the `javax.transaction.UserTransaction` interface to get the same information as `getRollbackOnly` and may invoke the `javax.transaction.rollback` method to roll back a transaction. While these methods in the `javax.transaction` package are not identical in function to the `get` and `set` rollback methods in the `EJBContext` object, the `javax.transaction` package methods achieve the same end results.

The getUserTransaction Method

Code the `getUserTransaction` method to obtain a reference to the current transaction if there is one. Once obtained, your code can take over transaction duties and issue commits and rollbacks.

As you may have guessed by now, beans with container-managed transactions cannot invoke `getUserTransaction`. Having container-managed beans obtain a reference to the current transaction is a bit silly because the container wouldn't be managing the transaction at all. Hence, only enterprise beans that are bean-managed may usefully invoke the `getUserTransaction` method.

The code snippet that follows shows a bean method acquiring a reference to the current transaction using the `getUserTransaction` method. Remember that you need to save the context in a variable known to all methods of the bean class.

```
public MySessionBean implements SessionBean {
  //SessionContext is the session bean subinterface of EJBContext
  SessionContext mySessionContext;

  //Save the session context….
  public void setSessionContext( SessionContext sctx )
              throws RemoteException {
      mySessionContext = sctx ;
   }

  public someBeanMethod() {
        //myTransaction is a reference to the current transaction
        UserTransaction myTransaction =
           mySessionContext.getUserTransaction();
        myTransaction.begin();
        //Do transaction stuff…
        myTransaction.commit();
  }
}
```

At the risk of sounding overly repetitive, boring, and preachy: strive to let the container manage as much of your bean activity as possible. Security was one of the main reasons Sun developed the EJB technology in the first place.

The getEJBHome Method

The `getEJBHome` method returns a reference to the bean's `home` object. I covered a bit about the `home` object in [Chapter 12](), "The Elements of an EJB," but to refresh your memory, the `home` object is the broker between the client and the EJB container. Put another way, the client invokes methods that are defined in the `home` object, and the `home` object delegates client calls to the EJB container, which in turn, invokes the appropriate method in your enterprise bean class.

The `getEJBHome` method allows your client to get a reference to the `home` object. In addition, your bean may use

`getEJBHome` to get a reference to its `home` object. A bean method may need to pass a reference to its `home` object as a parameter to another method in a different class.

By now you may have the strong sense that the EJB container plays an integral part in the development, deployment, and use of enterprise beans. Next, you read about the EJB container and the support services it provides.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Coding Entity Bean Creation Methods

As expected, a client accesses entity beans through the container's implementation of the bean's home interface. The home interface enables the client to create an entity bean, to obtain a reference to an existing bean, and to remove a bean. The creation, location, and removal of entity beans is done by the container (once again, the client never accesses or interacts with an enterprise bean directly).

You, or a client, create an entity bean the same way you would a session bean. The client program issues a call to the create method. The client class then implements `javax.ejb.EJBHome`, which signals the container to invoke a corresponding `ejbCreate` method implemented in the bean class. Recall that for session beans, you must code at least one `ejbCreate` method in your bean class. However, you are not required to code any `ejbCreate` methods for entity beans. For example, an online shopping application may have the product catalog stored in a database accessed through Enterprise JavaBeans. You certainly would want shoppers to peruse the catalog but you would not want shoppers to create new catalog items.

Of course, if the application designers decide not to allow entity bean creation through container invocations of `ejbCreate` methods, the designers need other methods to create entity beans. Fortunately, you can create the data underlying entity beans through conventional means — SQL insert statements, for example.

You code a create method in your client application, which instructs the container to invoke a corresponding `ejbCreate` method, followed by invoking an `ejbPostCreate` method, coded in your bean class. Also, you may code multiple create methods, passing different parameters. For every create method coded in your `home` interface and invoked in your client, you must code a corresponding `ejbCreate` method and a corresponding `ejbPostCreate` method containing the same parameter list.

You code `ejbCreate` methods for bean-managed entity beans a bit differently than `ejbCreate` methods for container-managed beans. However, the client code that invokes the create methods is the same for both entity bean types.

Let's see how to code create methods in the `home` interface and `ejbCreate` and `ejbPostCreate` methods of the bean class for BMP beans and CMP beans.

Coding create Methods

As previously mentioned, you are not required to code any `create` methods when implementing your bean's `home` interface. If you do, the signature will resemble the method signature that follows:

```
RemoteInterfaceName create<optional>( ClassProp1 prop1,
                                      ClassProp2 prop2, … )
        throws CreateException, RemoteException,
               ApplicationDefinedException1, …
               ApplicationDefinedExceptionN ;
```
Because EJB clients interact with the container through a remote interface, any create method should return an object of the bean's remote interface. As described in Chapter 12, "The Elements of an EJB," the remote interface defines the client view of the bean.

You may code a `create` method using the word "create" only or you may code a `create` method that includes optional words, such as a subclass name. For example, the `create` method that follows, when invoked, requests that the container create a special class of account called a large account:

```
Account createLargeAccount(String firstname, String lastname,
                           double initialBalance)
        throws RemoteException, CreateException;
```

Here, `Account` is the name of the bean's remote interface.

The arguments for the `create` methods are one or more properties of the bean class. You can code `create` methods with different arguments to initialize different properties of the bean. For example:

```
//Create account passing name and initial balance
Account create(String firstName, String lastName,
               double initialBalance)
        throws RemoteException, CreateException;

//Create account by passing account number and initial balance
Account create(String accountNumber, double initialBalance)
        throws RemoteException, CreateException,
               LowInitialBalanceException;
```
Finally, create methods coded in the home interface should throw, at a minimum, a RemoteException and a CreateException.

Remember that Enterprise JavaBeans is a distributed object technology and that method invocations are communicated to a container that resides on a different JVM. The mechanism is similar to RMI, whereby a method invocation goes through a stub, across the network, to another JVM. Hence, methods invoked by a client should throw a `RemoteException`.

The `CreateException` is peculiar to EJB `create` methods. You've seen other EJB-specific exceptions, such as `RemoveException` in <u>Listing 15-1</u> for the `ejbRemove` method and the general-purpose `EJBException` for all methods in the bean class.

You may also instruct `home` interface methods to throw one or more application-defined exceptions. The second `create` example in the previous code shows the `create` method throwing a `LowInitialBalanceException`. You learn more about application-defined exceptions later in this chapter.

Your bean classes should have `ejbCreate` and `ejbPostCreate` methods for every `create` method coded in the `home` interface. Let's take a look at how to code methods in the next two sections.

Coding ejbCreate Methods for BMP Beans

The container invokes an `ejbCreate` method after the client calls a `create` method of the home object. Your `ejbCreate` method header would resemble the following for your BMP beans:

```
PrimaryKeyClassName ejbCreate<optional>( ClassProp1 prop1,
                                         ClassProp2 prop2, … )
        throws CreateException, EJBException,
               ApplicationDefinedException1, …
               ApplicationDefinedExceptionN ;
```

When coding `ejbCreate` methods for bean-managed beans, you must return an instance of the *primary key class*. If you think about it, returning the primary key object makes sense from the container's viewpoint because the container will need the primary key object to locate the bean.

The prefix `ejb` of the bean class's `ejbCreate` methods is not a slip of the pen. The container knows how to match bean `ejbCreate` methods with the corresponding `create` methods in the `home` interface. The parameter lists of the `create` and `ejbCreate` methods must match.

For example, the BMP bean `ejbCreate` method corresponding to the `createLargeAccount` method shown in the <u>previous section</u> is as follows:

```
AccountPK ejbCreateLargeAccount(String firstname, String lastname,
                               double initialBalance)
        throws EJBException, CreateException;
```

When the client calls the `createLargeAccount` method, the home object would call the `ejbCreateLargeAccount` method.

Other than the bean method returning an instance of the primary key class and the name of the method prefixed with `ejb`, the remainder of the `create` method in the `home` interface is the same as the `ejbCreate` method in the bean class.

One final point: As of Enterprise JavaBeans 1.1, `ejbCreate` methods should not throw a `RemoteException`. The recommended practice is for `ejbCreate` methods to throw, at a minimum, an `EJBException` and a `CreateException`.

Note   As of the EJB 1.1 specification, EJB containers are still required to support the deprecated use of the `RemoteException`. If a `RemoteException` is thrown, EJB containers are instructed to treat it in the same way specified for treating a thrown `EJBException`.

Coding ejbPostCreate Methods for BMP Beans

As previously mentioned, you should code an `ejbPostCreate` method for each `ejbCreate` method coded in your bean class. The `ejbCreate` and `ejbPostCreate` methods are in pairs; each should have the same argument list and the same return type, and should throw the same exceptions.

The container calls the `ejbPostCreate` method immediately after calling the `ejbCreate` method. You have the chance to perform any initializations not possible in the corresponding `ejbCreate` method.

The syntax for the `ejbPostCreate` method is identical to that of the `ejbCreate` method described previously except for the method name — `ejbPostCreate` versus `ejbCreate`.

Coding ejbCreate and ejbPostCreate Methods for CMP Beans

Most of what you read in the <u>previous section</u> about `ejbCreate` and `ejbPostCreate` methods with BMP beans applies to CMP beans as well. The main difference is that `ejbCreate` and `ejbPostCreate` methods coded in a CMP bean do not return an instance of the primary key; they return *void*.

In BMP beans, the bean needs the primary key to access the bean instance. In CMP beans, the bean doesn't need to know about the primary key to access the bean because the container is responsible for bean access (hence, the name container-managed). It is the responsibility of the container to create a primary key used for CMP beans.

When the container invokes an `ejbCreate` method and the subsequent `ejbPostCreate` method (in response to a client invoking a `create` method), the

container typically inserts a row of data into the database for both BMP and CMP beans. The difference is that, with BMP beans, you must write code (usually an `SQL INSERT` statement) to perform the insert, whereas with CMP beans, you don't need to write any SQL.

Now that I've covered a bit about creating new entity beans, a few words on accessing existing entity beans are in order.

Coding finder Methods in the home Interface and ejbFind Methods in the Bean Class

The `finder` methods locate existing beans based on one or more criteria. Often a `finder` method uses instances of the primary key class to locate existing beans. However, you may code multiple `finder` methods, each passing a different set of parameters.

As with the `create` methods described previously, you code `finder` methods in the `home` interface. All `finder` methods have corresponding `ejbFind` methods in the bean class. As you might imagine, coding `finder` methods is different depending on whether you are coding for BMP beans or for CMP beans. You can read about coding `finder` methods in the bean class as we continue in this chapter, but first, let's take a look at coding `finder` methods in the `home` interface.

Coding finder Methods in the home Interface

Finder methods enable you to access beans in much the same way that `SQL SELECT` statements enable you to access data in relational database tables. Finder methods in the `home` interface have one of two looks, shown here:

```
RemoteInterfaceClass findByPrimaryKey ( PKClass aPrimaryKey)
                    throws RemoteException, FinderException,
                        ApplicationSpecificException ;

Collection    findBy<OneorMoreProperties> (ClassProp1 aProp1Obj,
                                            …
                                        ClassPropN aPropNObj)
                    throws RemoteException, FinderException,
                        ApplicationSpecificException ;
```

Again, because the `finder` method is invoked in a client that communicates to the container through the `remote` interface, the `finder` method returns an object of the `remote` interface.

Using a primary key for bean access should return one instance of the entity bean, just as accessing a relational table or view with a primary key should return one row of data.

Also notice the presence of a new exception class called `FinderException`. If you pardon the lame pun, there's nothing exceptional about `FinderException` (or `CreateException` for that matter). All you need to do is code the method header and throw the proper exception at the proper time. Coding your method to throw application-specific exceptions is not required.

You may wonder if EJB technology enables you to access or locate multiple entity beans, like the user of a relational database would use to access or locate multiple rows of a table or view. You can code multiple `finder` methods in your `home` interface. By accessing different properties of your entity beans, your `finder` method may return zero to many entity beans.

For example, the `finder` methods that follow locate entity beans by using different search criteria:

```
//Find account by using account holder name
Collection findByLastName( String lastName )
        throws RemoteException, FinderException ;

//Find all Large Accounts
Collection findLargeAccounts( double accountLimit )
        throws RemoteException, FinderException;
```

Notice the preceding two `finder` methods return an object of the `java.util.Collection` interface. A good rule of thumb is that `findByPrimaryKey` returns one object; `findBy<other property>` returns a collection.

Now, let's look at how to code the corresponding `finder` methods in the bean class.

Coding ejbFind methods in BMP Beans

As with `create` methods, you should code `finder` methods in your BMP bean class that correspond to the `finder` methods coded in your `home` interface. For every `finder` method coded in your `home` interface, you code an `ejbFind` method in your bean class. The general form should resemble one of the two templates that follow:

```
PKClass ejbFindByPrimaryKey ( PKClass aPrimaryKey)
                    throws FinderException,
                        ApplicationSpecificException ;

Collection ejbFindBy<OneorMoreProperties> (ClassProp1 aProp1Obj,
                                            …
                                        ClassPropN aPropNObj)
                    throws FinderException,
                        ApplicationSpecificException ;
```

As with the `create` methods, the client invokes a `finder` method, which instructs the EJB object to get the container to invoke the corresponding `ejbFind` method from the bean class.

The body of the `finder` methods should contain SQL or whatever database access language code you use to select data from the underlying database. That's why some entity beans are called *bean-managed;* the code in the bean is responsible for creating, accessing, and otherwise manipulating database data.

Not Coding Finder Methods in CMP Beans

In CMP beans, you *do not* code `finder` methods. A virtue of using CMP beans is that the container is responsible for locating beans and generating primary keys. You don't need to code `finder` methods in your CMP bean classes that return primary keys to the client.

Actually, in CMP beans, you *do not* code any database access language code. The container assumes full responsibility for any data manipulation. Later in this chapter, you learn how the container manages this apparent feat of magic.

The Primary Key Class

The earlier discussions in this chapter on the `finder` method and the `ejbRemove` and `ejbLoad` methods both mention the use of a primary key. In EJBs, primary keys are instances of a class which are used to locate particular entity beans. You need keys to access entity bean instances because EJB containers typically pool entity beans and may use a single entity bean to represent several instances of database data.

The primary key enables you to access a particular bean instance among several in the container. As previously mentioned, when you remove beans, you should take pains to ensure that the bean about to be removed (put back in the pool, actually) maps to database data that you intend to delete from the database. Also, when the container invokes an `ejbLoad` method, you should ensure that the data loaded from the database eventually gets assigned to instance variables of the bean instance currently bound to the EJB object.

A primary key class is just an ordinary, serializable Java class — it does not extend a bean class nor implement a bean interface. A primary key class is merely an encapsulation of one or more database columns that serve as the primary key to the data. The only hard and fast requirement of the primary key class is that the class implements the `Serializable` interface.

Primary key classes should override the `equals`, `hashcode`, and `toString` methods from class `Object`. Recall that the standard object equality operator (`==`) compares object *references*, not contents. At times, you'll need to know if two primary key objects have equal contents. The `equals` method from class `Object` offers no help because it just uses `==`; you should override `Object.equals`.

So, too, with the `hashcode` method in class `Object` — the `hashcode` method generates a hash code based on *memory locations*. However, for string objects, the `hashcode` method uses the *value* of the string to generate a `hashcode`. Thus, if the underlying data representation of your primary key is one or more strings, you don't need to override `hashcode`.

Finally, you should override the `Object.toString` method. You'll need a string representation of your primary key, but the default `Object.toString` method appends additional object information, which makes the `Object.toString` method unsuitable for converting a key to a string. Most of the time all you need to return is the instance variable (or variables) of the primary key object as strings.

In summary, a primary key class:

- is serializable

- overrides Object.equals, Object.hashcode, and Object.toString

For an example of a primary key class, please refer to later in this chapter.

Now let's take an in-depth look at the life cycle of an entity bean.

EJB & JSP: Java On The Edge, Unlimited Edition

 by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Chapter 19: The Proposed EJB 2.0 Specification

Throughout the book, you've read references to the latest proposal for Enterprise Java Beans, the specifications for the 2.0 release. With the Sun Microsystems specification document for EJB 2.0 weighing in at 558 pages (up from 314 pages for EJB 1.1), EJB 2.0 is no mere point release! In this chapter, you learn more about this "great leap forward." Sun published the final draft of the EJB 2.0 on October 23, 2000.

The chapter begins with a description of changes from EJB 1.1 to EJB 2.0. As you'll see, EJB 2.0 contains some new features plus refinements of existing (EJB 1.1) features. I explain the new features of EJB 2.0 and compare and contrast the refinements to the EJB architecture brought about by the new release.

EJB 1.1 Versus EJB 2.0: The Major Changes

With over 150 additional pages of specifications, you might expect that EJB 2.0 introduces some significant changes to Enterprise JavaBeans — and you'd be correct. The following is a brief rundown of the major changes:

- **Introduction of the `MessageDrivenBean`:** In addition to supporting session beans and entity beans, EJB 2.0 provides support for a new type of bean called a `MessageDrivenBean`, or message-driven beans. The short story is that a message-driven bean provides the integration of Java Messaging Service (JMS) with Enterprise JavaBeans. The longer story is covered later in this chapter.

- **The new container-managed persistence model:** Recall that container-managed persistence applies to entity beans where the EJB container automatically persists beans to the underlying database. As you read in Chapter 15, "EJB Entity Beans," the manner by which the EJB 1.1 container persisted beans is a bit ambiguous when objects contain other objects. The EJB 2.0 specification tightens up the rules for container-managed persistence by employing the services of the *persistence manager*.

- **The EJB Query Language (QL):** Entity beans are Java objects that are persisted to a database. The users of entity beans use `find` methods on the `home` interface for the bean to create and do searches for these objects. A lot of projects required custom `find` methods to support complex queries efficiently. This was possible using proprietary extensions provided by the application server vendors. These extensions used SQL to specify a `WHERE` clause for the query.

  Sun Microsystems recognized that relying on vendor-specific extensions was in conflict with the spirit of Enterprise JavaBeans. In response, Sun defined a new language for specifying these custom queries called the *EJB Query Language*, or QL. QL enables bean developers to use references to attributes on entity beans rather than columns in the database, further isolating bean developers from how the beans are mapped to the underlying database.

- **Additional methods in `home` interface:** Sun has provided additional support for the `home` interface to implement business logic that is independent of a specific enterprise bean instance by including additional methods.

  Sun also added a run-as security identity functionality for enterprise beans. This functionality allows for the declarative specification of the principal to be used for the run-as identity of an enterprise bean in terms of its

security role.

Sun defined an interoperability protocol based on CORBA/IIOP to allow invocations on session and entity beans from J2EE components that are deployed in products from different vendors.

Let's now take a more detailed look at the preceding enhancements to EJB available with the 2.0 release, starting with a discussion of the message-driven bean.

The MessageDrivenBean Type

A message-driven bean is an asynchronous message consumer. The EJB container invokes a message-driven bean as a result of the arrival of a JMS message. The container controls the life cycle for the message-driven bean. A message-driven bean has neither a `home` nor a `remote` interface. A message-driven bean instance is an instance of a message-driven bean class.

To a client, a message-driven bean is a JMS message consumer running on the server that implements some business logic. A client accesses a message-driven bean through JMS by sending messages to the JMS Destination (Queue or Topic) for which the message-driven bean class is the message listener.

Message-driven bean instances have no conversational state, which means that all bean instances, like stateless session beans, are equivalent when they are not involved in servicing a client message. In other words, message-driven beans have no special identity to the client. Actually, message-driven beans are not even visible to the client.

A bean instance has no state for a specific client. However, the instance variables of the message-driven bean instance can maintain state during the handling of client messages. Examples of state a message-driven bean might maintain include an open database connection or an object reference to an EJB object.

Leveraging Message-driven Beans

Although session and entity beans work well in implementing transactions, they're not perfect. Often, when session and entity beans work to complete a transaction, they do so in a synchronous manner. When dealing with messages, beans may have to wait until a message request is acknowledged. This waiting period could severely impede EJB performance.

To alleviate potential performance bottlenecks arising from the synchronous handling of messages, Sun Microsystems introduced JMS support for EJB in release 1.1. JMS is an asynchronous messaging service, which means that a JMS message sender doesn't need to wait for a response.

For example, if one or more beans are involved with a transaction, the steps required to complete the transaction usually follow a prescribed sequence. In addition, the next step in the transaction should not commence unless the previous step has successfully completed. Any messaging involved in processing a transaction should be synchronous.

In contrast, if the business scenario requires that other beans be notified of the status of the transaction, the notification may be sent asynchronously. An order placement bean may send messages to inventory beans so the inventory beans can adjust the stock items; the inventory bean may not need to message the order bean back.

Although EJB 1.1 supported JMS (and asynchronous messaging), EJB 2.0 wraps JMS inside a bean type and delegates much of the message-driven bean handling to the EJB container. In short, the EJB 2.0 bean developer may take advantage of the asynchronous messaging capabilities of JMS by developing beans as opposed to writing native JMS calls.

How does an EJB developer code a message-driven bean? In the next section, I show you how.

Coding a Message-driven bean

Not surprisingly, you code a message-driven bean by implementing interfaces. The three interfaces used when implementing message-driven beans are the `javax.ejb.MessageDrivenBean` interface, the `javax.ejb.MessageDrivenContext` interface, and the `javax.jms.MessageListener` interface (note that the third interface is in the `javax.jms` package).

Let's first look at the required interfaces that every message-driven bean must implement, and then let's look at a message-driven bean template.

The MessageDrivenBean Interface

All message-driven beans are coded as classes that implement the `MessageDrivenBean` interface. The `MessageDrivenBean` interface, like the `SessionBean` and `EntityBean` interfaces, extends the `EnterpriseBean` interface. The container uses the `MessageDrivenBean` methods to notify the bean instances of the instance's life cycle events.

The `MessageDrivenBean` interface contains two methods:

- **`setMessageDrivenContext`:** The bean container calls `setMessageDrivenContext` to associate a message-driven bean instance with its context maintained by the container. Typically a message-driven bean instance retains its message context as part of its state. Essentially, `setMessageDrivenContext` serves the same function for message-driven beans as `setEntityContext` does for entity beans and `setSessionContext` does for session beans.

- **`ejbRemove`:** When the container decides to remove an instance of a message-driven bean class, the container invokes the `ejbRemove` method. Again, `ejbRemove` serves the same purpose for message-driven beans as it does for entity and session beans.

The MessageDrivenContext Interface

The `MessageDrivenContext` interface, a subinterface of `EJBContext`, provides access to the runtime message-driven context that the container provides for a message-driven enterprise bean instance. The container passes the `MessageDrivenContext` interface to an instance after the instance has been created. The message-driven context remains associated with the instance for the lifetime of the instance.

The `MessageDrivenContext` interface adds no new methods to the `EJBContext` interface. However, message-driven beans must not call the `getCallerPrincipal` method, the `isCallerInRole` method, or the `getEJBHome` method.

The MessageListener Interface

Message-driven beans must implement the `MessageListener` interface. This interface has only one method requiring implementation, the `onMessage` method. The `onMessage` method is called by the bean's container when a message has arrived for the bean to service. The `onMessage` method contains the business logic that handles the processing of the message. The `onMessage` method has a single argument, the incoming message.

Only message-driven beans can asynchronously receive messages. Session and entity beans are not permitted to be JMS `MessageListeners`.

Generic Message-driven Bean Template

is a generic message-driven bean. You supply the implementation of the `onMessage` method to complete the bean.

Listing 19-1: A generic message-driven bean template

```
public class GenericMessageBean implements MessageDrivenBean, MessageListener {

  private MessageDrivenContext mCTX;

  //Sets the MessageDrivenContext.
  public void setMessageDrivenContext(MessageDrivenContext ctx) {
    mCTX = ctx;
  }

  //ejbCreate() with no arguments is required
  // by the EJB 2.0 specification

  public void ejbCreate () throws CreateException {}
  /**
    Supply an implementation of MessageListener by coding the
    onMessage() method below.
   */
   public void onMessage(Message msg) {
     // Implement your onMessage() method inside a try/catch block
     try {
      //Do it here……
     }
     catch(Exception ex) {
       ex.printStackTrace();
     }
  }
  //  Public, no arg constructor

  public MessageLogBean() {}

  //  ejbActivate is required by the EJB Specification

  public void ejbActivate() { }

  // ejbRemove is required by the EJB Specification

  public void ejbRemove() {
    mCTX = null;
  }

  //  ejbPassivate is required by the EJB Specification

  public void ejbPassivate() { }

}
```

Notice that, as with other bean types, several methods are required even though these required methods contain no real method body.

Message-driven Bean Life Cycle

A message-driven bean instance's life starts when the container invokes `newInstance` on the message-driven bean class to create a new instance. Next, the container calls `setMessageDrivenContext` followed by `ejbCreate`. Any client can now deliver a message to the message-driven bean instance.

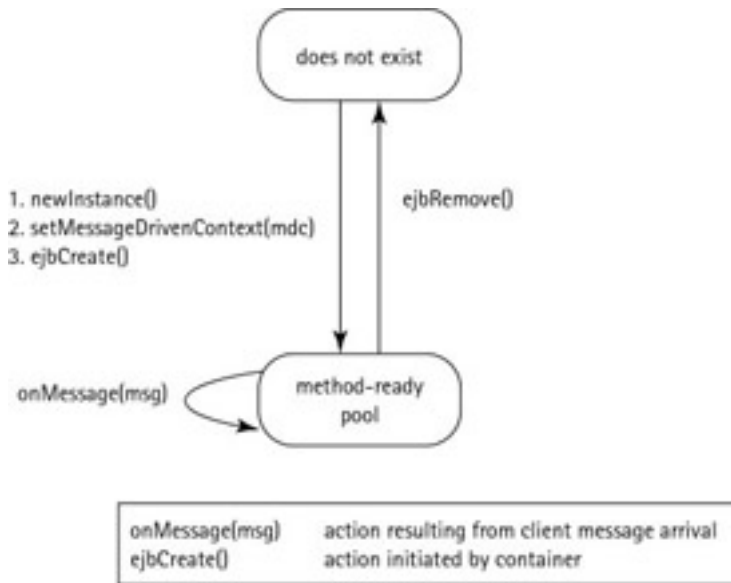shows a state diagram for a message-driven bean's life cycle.

Figure 19-1: Message bean life cycle

When the container no longer needs the instance (which usually happens when the container wants to reduce the number of instances in the method-ready pool), the container invokes the bean's `ejbRemove` method. This ends the life of the message-driven bean instance.

Closing Thoughts on Message-driven Beans

Message-driven beans are necessary to allow EJB application servers to compete on a level playing field against traditional transaction processing (TP) monitors. Modern enterprise applications are increasingly using message-based architectures. The message-driven bean feature now allows EJB application servers to be used out of the box in message-based architectures without resorting to using custom "message-driven beans."

Previously, the lack of support for message synks and transactions contained in JMS and JDBC was a problem when using EJB servers in large applications. With the introduction of the new message-driven bean, this obstacle has been removed.

The New Container-Managed Persistence Model

EJB 2.0 redefines the container-managed persistence model for entity beans. The major features of the new model include the introduction of a new role, the persistence manager; a new technique for defining container-managed fields; and new ways of working with objects of dependent classes and other EJBs.

The Persistence Manager

In EJB 2.0, the persistence manager handles persistence of container-managed entity beans automatically at runtime. The persistence manager is responsible for mapping the entity bean to the database based on a new bean-persistence manager contract called the *abstract persistence schema*. By having the persistence manager handle bean and dependent object relationships instead of the EJB container, entity beans developed under EJB 2.0 should be more portable across multiple containers than their EJB 1.1 counterparts.

The bean provider specifies the persistent fields and the relationships between dependent objects by coding abstract persistence schema statements in the deployment descriptor. The persistence manager provider should have tools to

map the persistent fields to a database according to the rules coded in the abstract persistence schema. The tools should generate all needed classes to provide for persisting the indicated fields *at runtime*.

Listing 19-2 shows a snippet from a deployment descriptor that uses the abstract persistence schema to specify a one-to-many relationship between two fields: `Order` and `LineItem`.

Listing 19-2: Deployment descriptor showing one-to-many relationship between two fields

```
<!-- ONE-TO-MANY: Order LineItem -->
<ejb-relation>
    <ejb-relation-name>Order-LineItem</ejb-relation-name>
    <ejb-relationship-role>
        <ejb-relationship-role-name>
            order-has-lineitems
        </ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <role-source>
            <ejb-name>OrderEJB</ejb-name>
        </role-source>
        <cmr-field>
            <cmr-field-name>lineItems</cmr-field-name>
            <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>

    <ejb-relationship-role>
        <ejb-relationship-role-name>lineitem_belongsto_order
        </ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <cascade-delete/>
        <role-source>
            <dependent-name>LineItem<dependent-name>
        </role-source>
        <cmr-field>
            <cmr-field-name>order</cmr-field-name>
        </cmr-field>
    </ejb-relationship-role>
</ejb-relation>
```

Each `ejb-relationship-role` element describes a relationship role, its name, its multiplicity within a relation, and its navigability. From the perspective of the participating bean (or its dependent object class), `ejb-relationship-role` also specifies the name of the `cmr-field` that is used.

Each relationship role refers to an entity bean or a dependent object class by using an `ejb-name`, a `remote-ejb-name`, or a `dependent-name` element contained in the `role-source` element. The bean provider must ensure that the content of each `role-source` element refers to an existing entity bean, entity bean reference, or dependent object class.

Closing Thoughts on EJB 2.0 Container-Managed Persistence

The EJB 2.0 CMP model helps normalize the CMP vendor offerings and makes beans much more likely to be portable between persistence manager providers. However, the EJB 2.0 CMP model requirements do not define a completely portable persistence mechanism. Because the object-to-datastore mapping is undefined, it remains possible to provide implementations that vary in their capabilities. For example, it is possible for a vendor to supply a CMP implementation in which a given object can map only to a single table. In addition, no interface is defined between the persistence manager and the container, which means that collaboration between the two occurs in a proprietary,

application server–specific way. In general, though, the CMP model defined in EJB 2.0 provides a more robust infrastructure than EJB 1.1 for persisting data.

Using the EJB Query Language

The persistence manager is responsible for implementing and executing `find` methods based on a new query language called *EJB Query Language (EJB QL)*. The structure of the QL statements depends on the `find` methods coded in the `home` interface of the entity bean and the relationships of fields between beans coded in the deployment descriptor.

The EJB QL defines query methods (`find` and `select` methods) for entity beans with container-managed persistence. EJB QL defines query methods so that they are portable across containers and persistence managers. EJB QL is a declarative, SQL-like language intended to be compiled to the target language of the persistent data store used by a persistence manager.

EJB QL is based on a subset of SQL92 and is enhanced by path expressions that allow navigation over the relationships defined for entity beans and dependent object classes.

EJB QL is a specification language that can be compiled to the persistent storage target language used by the persistence manager, such as SQL. This allows the responsibility for the execution of queries to be shifted to the native language facilities provided for the persistent store (e.g., RDBMS). Queries should no longer need to be executed directly on the persistent manager's representation of the entity bean's state. As a result, you can more easily optimize query methods. The methods are also easily portable between different database systems.

The EJB QL uses the abstract persistence schema of entity beans and dependent object classes, including their relationships, for its data model. It defines operators and expressions based on this data model.

The bean provider uses EJB QL to write queries based on the abstract persistence schemas and the relationships defined in the deployment descriptor. EJB QL depends on navigation and selection based on the `cmp-fields` and `cmr-fields` of the abstract schema types of related entity beans and dependent objects.

The bean provider can navigate from an entity bean or dependent object to other dependent objects or beans by using the names of `cmr-fields` in EJB QL queries.

EJB QL allows the bean provider to use the abstract schema types of related entity beans in a query if the abstract persistence schemas of the related beans are defined in the same deployment descriptor as the query. The bean provider can navigate to both locally defined entity beans and to remote entity beans. (In this context, remote entity beans are entity beans with bean-managed persistence, entity beans using EJB 1.1 container-managed persistence, and beans whose abstract persistence schemas are defined in a different deployment descriptor.) Although the abstract persistence schemas of remote entity beans are not available to the bean provider, it is still possible to use EJB QL to navigate to such remote entity beans. In addition, special expressions in the language allow the bean provider to invoke the `find` methods of remote entity beans in queries.

EJB QL Statement Structure

An EJB QL query is a string containing one of the following three clauses:

- A `SELECT` clause, which indicates the types of the objects or values to be selected

- A `FROM` clause, which provides navigation declarations that designate the domain to which the conditional expression specified in the `WHERE` clause of the query applies

- A `WHERE` clause, which restricts the results that are returned by the query

The `FROM` clause is required; `SELECT` and `WHERE` are optional.

EJB SQL supports many of the SQL92 operators used in predicates, such as `in`, `between`, `like`, `not`, and the use of `null`.

An EJB QL Example

Because QL statements are based on the `find` methods in the bean's `home` interface, you need an example `home` interface to work with. [Listing 19-3](#) is a `home` interface for a customer bean.

Listing 19-3: A home interface for a customer bean

```
public interface CustomerHome extends javax.ejb.EJBHome {
...

  public Employee findByPrimaryKey(Integer id)
         throws RemoteException, CreateException;

  public Collection findByZipCode(String zipcode)
         throws RemoteException, CreateException;

  public Collection findByProduct(String ProductName)
         throws RemoteException, CreateException;

}
```

You can use EJB QL to specify how the persistence manager should execute the `find` methods. All entity beans are required to have a `findByPrimaryKey` method. You don't use QL to execute the `findByPrimaryKey` method; you use the fields in the primary key to search the database.

The `findByZipCode` method is used to obtain all the customer beans with a certain zip code. You can use the following EJB QL in the deployment descriptor as follows:

```
FROM customerTable WHERE customerTable.zip = ?1
```

The question mark, reminiscent of the `PreparedStatement` interface in JDBC, is a placeholder for the argument passed by the `findByZipCode` method.

EJB QL expressions are coded in the deployment descriptor. Listing 19-4 provides an example.

Listing 19-4: EJB QL coded in deployment descriptor

```
<query>
   <query-method>
     <method-name>findByZipCode</method-name>
     </method-params>
     <ejb-ql> FROM customerTable WHERE customerTable.zip = ?1
     </ejb-ql>
   <query-method>
</query>
```

Closing Thoughts on EJB QL

EJB QL allows the bean developer to delegate the execution of bean `find` methods to the persistence manager. The EJB developer is not required to use EJB QL, but the straightforward syntax should make EJB QL an essential tool in

the EJB developer's bag of tricks.

Additional Methods in home Interfaces

EJB 2.0 allows bean developers to code additional methods in the `home` interface that do not apply to any single bean instance. Such methods are implemented by corresponding `ejbHome` methods. For example, the method header coded in the following customer bean's `home` interface:

```
public interface CustomerHome extends javax.ejb.EJBHome {

  public void increaseCredit( double increasePercentage )
        throws RemoteException ;
```

would have a method in the customer bean class with the following signature:

```
public void ejbHomeIncreaseCredit( double increasePercentage )
```

`home` interfaces in EJB 1.*x* specifications looked like singletons (that is, there was only one instance) to the developer. They contained `find` methods for their associated bean (each bean has its own home) and methods to create and delete the beans. Developers had a need for methods that applied to all beans represented by the home. There was no standard place to put these in the EJB 1.*x* servers, so developers usually created a stateless session bean and put the method there.

Developers can now add these methods to the `home` interface where they belong. A typical use for these methods is as follows: You have an employee entity bean that holds employee names and salaries. Previously, you would have a method on the employee bean to adjust the salary. If you wanted to change all salaries, the client of the bean wrote code that iterated over all beans and called the method for each one. Such an operation should go in a session bean, which is co-located with the entity bean for best performance. You can now put this method on the `home` interface. This makes much more sense, because it's where one expects to find all "static" methods for a bean.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco

ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

In this chapter, you've read about the elements of an EJB and how these elements relate to one another. You've read about the interfaces that represent server-side EJB components to clients and about the enterprise bean class. Now, you know that EJBs come in three flavors, with each flavor dealing with a specific feature of using objects in a distributed environment. You should be able to use this information to model your application components based on which ones should be entity, session, or message beans. In the next chapter we'll continue to look at EJB code, and we'll examine them in yet deeper detail.

The javax.ejb.EntityBean Interface

The `EntityBean` interface, a subinterface of `javax.ejb.EnterpriseBean`, is implemented by every entity enterprise bean class. The container uses the `EntityBean` methods to notify the enterprise bean instances of the instance's life cycle events. Here you implement any access methods to the bean's data. As you can see, you need not implement most of these methods for CMP beans, as the container does most of the dirty work for you. Listing 15-1 shows the entity bean interface.

Listing 15-1: The EntityBean interface

```
public interface javax.ejb.EntityBean
        extends javax.ejb.EnterpriseBean {

  public void ejbRemove( )
        throws javax.ejb.RemoveException,
        throws javax.ejb.EJBException;

  public void setEntityContext( EntityContext ctx )
        throws javax.ejb.EJBException;

  public void unsetEntityContext( )
        throws java.rmi.EJBException ;

  public void ejbPassivate( )
        throws javax.ejb.EJBException;

  public void ejbActivate( )
        throws javax.ejb.EJBException;

  public void ejbStore( )
        throws javax.ejb.EJBException;

  public void ejbLoad( )
        throws javax.ejb.EJBException;


}
```

Note   Release 1.0 of EJB required that entity (and session) bean methods throw a `RemoteException`. You may see bean method signatures that throw both `RemoteException` and `EJBException`. With newer EJB releases, all you need to throw is `EJBException`, although `RemoteException` may be present for backward compatibility with EJB 1.0.

You've seen some of these methods in Chapter 14, "EJB Session Beans." In particular, the methods `ejbPassivate`, `ejbActivate`, and `ejbRemove` perform much the same function for session beans as for entity beans. No need to

repeat the descriptions of these methods. I'll mention any differences between these methods when used for session and entity beans later.

Let's look at the methods in the entity bean interface.

Using ejbRemove

The `ejbRemove`method behaves a bit differently when invoked from a session bean than from an entity bean. If you flip back to Chapter 14, you'll note that the signature of `ejbRemove` for session beans throws an `EJBException`; if you look at Listing 15-1, you'll note that `ejbRemove` for entity beans throws an `EJBException` *and* a `RemoveException`.

Recall from Chapter 14 that the EJB container invokes `ejbRemove` immediately prior to the container's removal of a session bean instance. The EJB container activity leading up to invoking an `ejbRemove` method for an entity bean is a bit different. The EJB container invokes `ejbRemove` in response to a client invocation of a `remove` method (like session beans). However, the container *does not* remove an instance of an entity bean before, during, or after the invocation of `ejbRemove`. The container, when executing `ejbRemove`, removes *the data in the database represented by the entity bean*, not the instance of the bean. Put another way, the container changes the state of an entity bean from "ready" to "pooled."

You need to ensure that the bean you are about to remove corresponds to the correct row of data in the database. Because the container pools beans, you run the risk of removing a bean that is bound to a different row of database data than the data you wish to remove from the database. You may be wondering how you tell the container which row of data to remove.

The answer is that you access the *primary key* from the container's entity context, and then use that primary key to locate the correct data from the database. With the primary key, you issue a SQL delete or whatever database dialect is used to remove database data. You learn more about the primary key later in this chapter.

You may think that the difference between the behavior of `ejbRemove` for session and entity beans is insignificant. Why should the client or you, the distributed object programmer, care about the pooling of beans and beans transitioning from a "ready" state to a "pooled" state? Most of the time you won't care. The important difference between `ejbRemove` when applied to an entity bean is that the container will remove data from the database as part of processing the `ejbRemove` method.

Using setEntityContext

The `setEntityContext` method serves a similar function for entity beans as the `setSessionContext` method does for session beans. The entity context allows your entity beans to learn about and communicate with the EJB container. Later in this chapter, you'll read about the methods available in the `EntityContext` interface and the methods in the superinterface of both `SessionContext` and `EntityContext` — the `EJBContext` interface.

Typically, your bean class contains a method implementation of `setEntityContext` that holds onto a reference of the entity context object. The implementation will closely resemble the following example.

```
//Declared outside any method
private myEntityContext ;

public void setEntityContext( EntityContext ectx )
            throws RemoteException {
    myEntityContext = ectx ;
}
```

Now, any bean method can access the entity context through `myEntityContext`, which is known throughout the bean class.

Using unsetEntityContext

Unlike the session bean interface, the entity bean interface contains a method to disassociate the bean from its context. This method, `unsetEntityContext()`, serves the same purpose for entity beans as `ejbRemove()` does for session beans. The container invokes `unsetEntityContext()` when the container decides to move an entity bean from the "pooled" state to the "does not exist" state.

Using ejbPassivate and ejbActivate

The `ejbPassivate` and `ejbActivate` methods perform much the same function for entity beans as for session beans. When the EJB container decides to pull out an entity bean from the pool and draft the bean for use, the container invokes the bean's `ejbActivate` method; immediately before the EJB container decides it's time for the bean to return to the pool, the container executes the `ejbPassivate` method.

Using ejbStore and ejbLoad

The `ejbStore` method is used with entity beans, not with session beans. Recall that entity beans represent data resident in some underlying data store. When the EJB container passivates an entity bean, the container needs a mechanism to save the data to the data store so the persistent data reflects the state of the entity bean.

Before the container passivates an entity bean, the container invokes the `ejbStore` method. The purpose of `ejbStore` is to save the entity bean's instance variables to the database.

It's not difficult to deduce the purpose of the `ejbLoad` method. Immediately following bean activation, or the invocation of `ejbActivate`, the container invokes the bean's `ejbLoad` method. The `ejbLoad` method refreshes the newly activated bean with the latest version of its underlying data.

Loading the database data into the bean's instance variables requires querying the data from persistent storage (usually a relational database) and using the appropriate mechanism (usually SQL). However, for BMP beans, you should ensure that the data you are loading belongs to the bean instance bound to the current EJB object. You need to be sure because the container pools beans and may use one bean instance to represent several EJB objects.

How do you guarantee that you are loading the right data, specifically the data belonging to the instance bound to the current EJB object? You fetch the bean instance that corresponds to the EJB object by using a *primary key* and a *finder method*. You can read more on coding and using finder methods and primary keys later in this chapter.

Why Does EJB Require load and store Methods?

Perhaps you are wondering why the container requires separate methods to activate and passivate beans and to load and store beans. Why doesn't the container load the entity bean as part of its `ejbActivate` method and save the bean as part of its `ejbPassivate` method? After all, the container doesn't require `load` and `store` methods for session beans; the container is able to save and load the state of a session bean through its `activate` and `passivate` methods.

Recall that an entity bean represents data whereas a session bean does not. The `activate` and `passivate` methods for a session bean save its state through serialization. However, more than serialization is required for entity beans. As a result, the `activate` and `passivate` methods for entity beans deserialize and serialize the bean; the `load` and `store` methods tend to fetch and save entity bean data stored in a database.

Other Methods You Would Typically Implement When Coding Entity Beans

A quick look at the `javax.ejb.EntityBean` interface reveals an absence of methods used to create new entity beans and to access existing entity beans. Recall that a client application interacts with the EJB container to create, access, and manipulate enterprise beans. Methods used to create and locate entity beans are defined in the bean's `home` interface. The code that implements the actions required to create and access the bean is located in the bean class, or the class that implements `javax.ejb.EntityBean`.

You can also implement methods that enable a client to instruct the container to access an instance variable of an entity bean — a column of a row of underlying relational database data, perhaps. You can code `get` and `set` methods to access and change column values. Later in this chapter you will see that your CMP beans will not contain SQL statements, while your BMP beans will.

In , "The Elements of an EJB," I discussed the `home` interface and briefly described the `home` interface methods. In the following sections, I describe the `home` interface methods for entity beans in more detail.

Top

Prev

Next

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

You made it! You've now looked at session beans in their stateful and stateless varieties and you've been able to read
about how you can use them in your enterprise applications. After examining the sample session beans that you've
seen implemented in this chapter, you should have a strong understanding of the different situations in which you may
want to use stateful and stateless session beans. In the next chapter, we'll complement this knowledge with a
discussion of entity beans and an example implementing them.

does not exist

1. newInstance()
2. setMessageDrivenContext(mdc)
3. ejbCreate()

ejbRemove()

onMessage(msg)

method-ready
pool

onMessage(msg)	action resulting from client message arrival
ejbCreate()	action initiated by container

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

In the previous chapters in Part III of this book we've concentrated on learning to develop entity and session enterprise
beans. Now we've learned how to create EJB clients to access these EJBs. Coding EJB clients is a matter of coding
some statements up front — namely, creating the initial context and locating the home object via JNDI. Once you take
care of the housekeeping, you can invoke bean methods as if they were located on the same JVM as the client.
Whether your EJB client is an application, servlet, applet, or another enterprise bean, you can now access and use
EJBs with ease.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
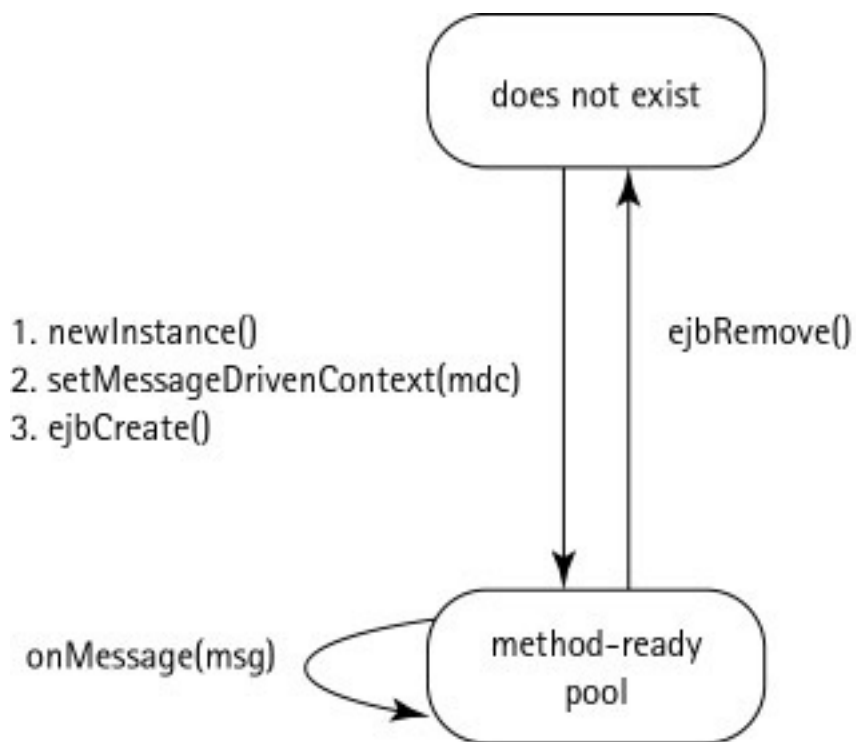Techniques.

Summary

Sun Microsystems has introduced dramatic and meaningful changes in the EJB 2.0 release. Message-driven beans, persistence managers, and EJB QL are all aspects that give the developer the ability to build better enterprise applications that are more flexible, portable, and robust. Although vendor support for the new release is scarce, if nonexistent, these changes are welcomed by developers of distributed applications. Developer support of the specification and the fact that Java is the language of Internet programming will ensure that vendor support grows as this specification takes hold in industry.

Now, let's continue to the last chapter in this book and tie it all together, building a Web-based JSP application powered by EJBs.

EJB Deployment Descriptors

Notice that in all of the talk about home and remote interfaces and entity and session beans, there has been no discussion of how to tell the Java environment various runtime attributes of beans. In addition, the topic of security or transaction properties has not been discussed. Although the EJB container handles these, and other, system-level services, you still must tell the container *how* to handle these services. Here's where the *deployment descriptor* comes into play.

The role of the deployment descriptor is to capture the declarative information (i.e., information that is not included directly in the enterprise bean's code) that is intended for the consumer of the `ejb-jar` file.

A deployment descriptor is a set of serialized classes that serve a similar function to property files. If you are using a Java IDE for enterprise application development, the IDE probably has a tool that may assist you in the creation of a deployment descriptor.

There are two basic kinds of information in the deployment descriptor: *structural* and *application assembly* information.

Structural information describes the structure of an enterprise bean and declares an enterprise bean's external dependencies. Providing structural information in the deployment descriptor is mandatory for the `ejb-jar` file producer. The structural information cannot, in general, be changed because doing so can break the enterprise bean's function.

Application assembly information describes how the enterprise bean (or beans) in the `ejb-jar` file is composed into a larger application deployment unit. Providing assembly information in the deployment descriptor is optional for the `ejb-jar` file producer. Assembly level information can be changed without breaking the enterprise bean's function, although doing so may alter the behavior of an assembled application.

In Chapters 14 and 15, you will create EJBs and their corresponding deployment descriptors. For now, understand that by using deployment descriptors, the bean deployer can customize the behavior of EJBs at runtime.

Coding an EJB Client

The Enterprise JavaBeans architecture makes coding client applications that access enterprise beans straightforward. The overall structure is that the client needs to obtain a reference to the bean and then invoke methods defined in the bean's remote interface. As you've read, the `remote` interface methods correspond to bean methods. When the client invokes a remote method, the EJB object delegates the invocation to the container. The container invokes the corresponding bean method and handles the return of any needed data to the client. The general mechanism is similar to using Java RMI.

Obtaining a Reference to the EJB Object

Before a client can invoke methods of the EJB object, which requests that the container invoke bean methods, the client must obtain a reference to the EJB object. As mentioned earlier, the client first obtains a reference to the home object. The home object contains methods that enable the client to locate EJB objects. The EJB objects contain methods that map to enterprise bean business methods.

Obtaining a Reference to the Home Object

A client uses JNDI to get a reference to the home object, which must then be cast to the `home` interface. Listing 18-1 demonstrates code that gets a reference to the home object. The code is shown with fully qualified class names to show where the JNDI and RMI classes derive from.

Listing 18-1: Obtaining a reference to the home object

```
//Name of the bean as coded in the deployment descriptor
private static final String HOME_IFACE_REF =
                          "java:comp/env/ejb/MyBean" ;
try {
   //Use JNDI to obtain a default naming context
   javax.naming.Context jndiCtx =
                       javax.naming.InitialContext() ;

   //Use lookup() to get the home object
   Object Obj = jndiCtx.lookup( HOME_IFACE_REF ) ;

   //Use the 'special cast' to coerce to home interface type
   HomeIfaceClass homeObj = ( HomeIfaceClass )
               javax.rmi.PortableRemoteObject.narrow(obj,
                       HomeIfaceClass.class);
   //Reference EJB Objects, create, destroy, etc.

}
catch (javax.naming.NamingException ne) {
   //Mention somehow that the JNDI lookup method failed
```

```
     //to locate the home object
}
```

The JNDI Name for the Home Object

The static string `HOME_IFACE_REF` is the JNDI name for the home object. Listing 18-2 shows how the JNDI name is coded in the deployment descriptor.

Listing 18-2: Deployment descriptor showing home interface JNDI reference

```
<ejb-ref>
  <description>
    Example bean used in Chapter 18
  </description>
  <ejb-ref-name>ejb/MyBean</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>chapter18.MyBeanHome</home>
  <remote>chapter18.MyBeanRemote</remote>
</ejb-ref>
```

As you can see, the JNDI name is coded as content for the `ejb-ref-name` tag. The choice of an `entity` bean type is completely arbitrary in this example.

Obtaining the Initial JNDI Naming Context

The call to the method `InitialContext()` of the `javax.naming` package generates an initial JNDI naming context. You need the initial context to perform a lookup in the JNDI namespace for the home object.

Looking Up the Home Object

The call to the method `lookup()` with the JNDI name coded in the deployment descriptor as the method argument yields an object reference or throws a `NamingException` if not found.

Performing the "Special Cast"

The `lookup()` method returns an object of class `Object`. For this object to be of any use, it must be cast into the class that implements the `home` interface. Because a stated goal of Enterprise JavaBeans is CORBA compatibility, EJB does not permit a Java-style cast because CORBA objects may not be Java-based and, if not, will fail to yield to a Java-style cast. To allow EJB to comply with CORBA as far as object casting is concerned, EJB permits the use of the `narrow()` method of class `javax.rmi.PortableRemoteObject`. The effect of using `narrow()` is tantamount to using a Java-style cast.

Notice that you can't tell by looking at the code in <u>Listing 18-1</u> what sort of Java object the client is. The client can be an application, applet, servlet, or another enterprise bean.

Now that you have a reference to the home object, you can prepare to access the EJB object and the accompanying bean operations.

Creating an EJB Object

Before you look at code that "creates" EJB objects, know that methods invoked by the client may not create any objects. In the case of stateless session beans, an EJB object is likely pulled from a bean pool, activated, and bound to the client for the duration of the method execution. In the case of a stateful session bean, an EJB object is likely created and bound to the client for the duration of the session. In the case of an entity bean, an EJB object is likely pulled from the pool, activated, and associated with data that will eventually be persisted if a transaction occurs and is committed. However, from the client's point of view, the code that "creates" the object provides the client an object by which the client can invoke bean methods.

The following code "creates" an EJB object that the client may use to communicate requests to the container.

```
//Stateless session bean – no arguments to create() method
RemoteStatelessSessionClass myBeanObject = homeObj.create() ;
//Entity Bean – may or may not have arguments
RemoteEntityClass          myBeanObject = homeObj.create( aParm );
```

Of course, the `create()` method of the `home` interface may or may not take arguments depending on the type of bean and the particulars of the bean being "created."

EJB `create()` methods may throw a `javax.ejb.CreateException`. The client should be prepared to catch the exception.

Invoking a Bean Method

Now that the client has a reference to a bean or, more accurately, a reference that allows the client to communicate a method invocation request to the container, invoking the method is straightforward:

```
//Parameters vary with method, of course
AClass   objAClass = myBeanObject.doSomething( aParm1, aParm2 ) ;
```

In other words, the client invokes the method as if the method were available locally, in the same JVM as the client.

Finding One or More Beans

Locating an entity bean by primary key is also straightforward. The code snippet that follows shows the creation of a primary key and then the location of the bean referenced by that key:

```
//aKeyValue is declared and initialized to the correct type
RemoteEntityClassPK = new RemoteEntityClassPK( aKeyValue ) ;
//Look for the bean (reference to a remote object, really)
//accessed by this key value
RemoteEntityBean   myOtherBean =
              homeObj.findByPrimaryKey(RemoteEntityClassPK ) ;
```

Notice that the `find()` method is invoked from the *home object reference.*

To locate multiple beans with a `find()` method, you need to set the search parameters as arguments to the `find()` method and save the return value of the `find()` method as an object from a class that implements the `Collection` interface. The code that follows returns a collection of "big beans."

```
Collection         myBigBeans =
              HomeObj.findBigBeans( sizeForBigBean ) ;
```

EJB `findBy()` methods may throw a `javax.ejb.FinderException`. The client should be prepared to catch the exception.

Removing a Bean

Removing a bean does not delete the bean or purge the bean from memory. However, from the client's point of view, removing the bean makes any references to the bean invalid. Thus, to the client, the net effect of bean removal is the same as if the bean were deleted or purged.

The following line of code "removes" the bean, making any references to the bean invalid:

```
myOtherBean.remove() ;
```

If only all things in life were so simple!

EJB `remove()` methods may throw a `javax.ejb.RemoveException`. The client should be prepared to catch the exception.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Chapter 20: Integrating JSPs and EJBs

Overview

In Chapter 10 you read about how to use JSPs to implement a straightforward brokerage application. The application
used JSP pages to generate a front end for clients and to communicate with a database in response to client
requests.

Since then you've also read about the different types of enterprise beans — session, entity, and message-driven
beans — and have studied code samples that implement enterprise beans. You've read about coding EJB clients in
Chapter 18. The EJB clients presented in Chapter 18 were Java classes containing methods that communicated with
enterprise beans.

In this chapter, you will learn how to code EJB clients as JSP pages. This chapter presents code that implements the
"View Transaction History" feature of a version of the "Make Money" Brokerage Application presented in Chapter 10.
The major difference between the version in Chapter 10 and that in this chapter is that the Chapter 10 version relies
solely on JSP pages for client interface and data access tasks whereas the version in this chapter relies on a JSP
page for client interface and an enterprise bean for data access.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

The Bean Class

Objects of the bean class are the enterprise beans. As previously mentioned, beans come in three flavors: entity, session, and message driven. The bean class contains implementations for methods that parallel those found in the home interface and remote interface. The method names and the signatures listed in the remote interface must exactly match the method names and signatures of the business methods defined by the enterprise bean. This differs from the home interface, whose method signatures match, but whose names are different.

The interface `EnterpriseBean` is a marker interface, serving as a superinterface for three interfaces corresponding to the three bean types previously mentioned. You can't implement the `EnterpriseBean` interface directly. Instead, you implement the `EntityBean`, `SessionBean`, or `MessageDrivenBean` interfaces.

The next sections describe the three bean types previously mentioned, starting with entity beans.

Using Entity Beans

An entity bean represents persistent data, methods that act on that data, and state management callback methods. In relational terms, an entity bean might represent an underlying database row, or a single result row returned by a SQL query. In an object-oriented database (OODB), an entity bean may represent a single object, with its associated attributes and relationships. An example of an entity bean might be a room object in a particular hotel, or a specific customer in a customer database.

Entity beans are associated with database transactions and may provide data access to multiple users. Because the data that an entity bean represents is persistent, entity beans survive server crashes because as soon as the server comes back online, the server may reconstruct the bean from existing data. In addition, references to an entity bean survive server crashes. A client can later connect to the same entity bean using its object reference because it encapsulates a unique *primary key,* enabling the enterprise bean, or its container, to reload its state.

The management callback methods notify the entity bean when a significant change of state is about to occur. For example, an entity bean can be notified when the bean is about to be removed from the JVM, causing the underlying data to be removed from the database. The entity bean may want to do some additional cleanup before removal.

Entity beans support two types of persistence: *container-managed* and *bean-managed.*

In container-managed persistence, the EJB container is responsible for saving the state of the entity bean. Because it is container-managed, the implementation is independent of the data source. All container-managed fields need to be specified in the deployment descriptor for the persistence to be automatically handled by the container. Later in this chapter, you can read a bit about EJB deployment descriptors.

In bean-managed persistence, the entity bean is directly responsible for saving its own state, and the container does not need to generate any database calls. Consequently, bean-managed persistence is less adaptable than container-managed persistence because the persistence needs to be hard-coded into the bean.

An EJB feature, new with release 2.0, available to entity beans with container-managed persistence is the *EJB Query Language* (EJB QL). EJB QL is a specification language that can be compiled to a target language (SQL is the most likely candidate) of a database used by the persistence manager. (Recall that the persistence manager is the newest role, discussed in Chapter 11, "A First Look at EJB.") With EJB QL, the responsibility for the execution of queries is shifted to the native language facilities available in the underlying database instead of requiring queries to be executed directly on the entity beans. You can read more about EJB QL in Chapter 15, "EJB Entity Beans."

To summarize, every entity bean has the following characteristics:

- Entity beans represent data in a database.

- Entity beans can participate in transactions.

- Entity beans are persistent, living as long as the bean's underlying data lives in the database.

- Entity beans may be accessed by multiple users.

- Entity beans can survive EJB server crashes. Any EJB server crash is always transparent to the client.

- Entity beans have a persistent object reference called the primary key.

- Entity beans may implement persistence in two ways: container-managed and bean-managed.

- Container-managed beans may use EJB QL, a new feature with EJB 2.0 that enables execution of queries on the underlying database.

You can read more about entity beans in Chapter 15, "EJB Entity Beans," and in the remaining chapters.

If you're wondering what the code for an entity bean looks like, the next section discusses a possible template for an entity bean.

Entity Bean Template

Your implementation of an entity bean that works with the home interface template (Listing 12-2) and the remote interface template (Listing 12-4) can resemble the template shown in Listing 12-5.

Listing 12-5: Template for defining an entity bean

```
import java.rmi.* ;
import javax.ejb.* ;
import java.util.* ;

public class  MyBeanClass implements EntityBean {

    /** Bean properties Follow **/
    public Type1  beanProp1 ;
    public Type2  beanProp2 ;
            …
    public Typen  beanpropn ;
    public Typei  beanPropi ;
    public Typej  beanPropj ;
            …
    public Typez  beanpropz ;

    public void ejbCreate ( Type1 varType1,
                            Type2 varType2,
                                  …
                            Typen varTypen ) {
        beanProp1 = varType1 ;
```

```java
    beanProp2 = varType2 ;
        …
    beanPropn = varTypen ;
    /**
        Initialize other properties
    **/
    initializePropsiThroughz() ;

}

public void ejbAnotherCreate ( Typei varTypei,
                               Typej varTypej,
                                      …
                               Typez varTypez ) {
    beanPropi = varTypei ;
    beanPropj = varTypej ;
        …
    beanPropz = varTypez ;
    /**
        Initialize other properties
    **/
    initializeProps1Throughn() ;

}

public MyBeanClassKey ejbFindByPrimaryKey( myBeanClassKey aKey ) {
    /** Code to access a relational store with JDBC,
        perhaps…
    **/
    return myBeanClassObject ;
}


public Collection ejbFindByDiffTypes
                      ( DiffType1 varDiffType,
                        DiffType2 varDiffType2) {
    /** Code to access a relational store with JDBC…
     **/
    return myBeanClassObject ;
}

public Type1 getType1Property()
        throws RemoteException {
        return beanProp1 ;
}

public void setType1Property( Type1 varType1 )
            throws RemoteException {
    beanProp1 = varType1 ;
}
//
//Other accessor methods would follow
//
public boolean processWasSuccessful ( Type1 varType1,
                                              …
                                      Typen varTypen )
                throws RemoteException {
    /** Some code to do this process **/
    return successful ;

}
```

```
      public void doAProcess (Typea varTypea,
                                     …
                            Typez varTypez )
               throws RemoteException {
         /** Some code to do this process **/
      }
      //
      //Other business methods would follow
      //
      public void ejbPostCreate ( Type1 varType1,
                                  Type2 varType2,
                                        …
                                  Typen varTypen )
               throws RemoteException {
         /** Take action if the business logic dictates
             activity after the bean is created
         **/
      }
      public void ejbPostAnotherCreate ( Typei varTypei,
                                         Typej varTypej,
                                               …
                                         Typez varTypez )
               throws RemoteException {
         /** Take action if the business logic dictates
             activity after the bean is created
         **/
      }
      //State management methods follow. No implementations
      //provided for this example
      public void setEntityContext( EntityContext eCTX )
               throws RemoteException {
         //
      }
      public void unsetEntityContext ( )
               throws RemoteException {
         //
      }
      public void ejbActivate ( )
               throws RemoteException {
         //
      }
      public void ejbPassivate ( )
               throws RemoteException {
         //
      }
      public void ejbStore ( )
               throws RemoteException {
         //
      }
      public void ejbLoad ( )
               throws RemoteException {
         //
      }
      public void ejbRemove ( )
            throws RemoteException {
         //
      }
}
```

About the preceding code listing a few points are worthy of mention:

- All methods defined in the home interface are referenced in the bean class. However, the methods in the bean class have the prefix `ejb`. Hence, in the `home` method, the method `create` is known as `ejbCreate` in the bean class. Notice that the prefacing of the `home` method name with `ejb` also holds for `finder` methods, too.

- Speaking of `finder` methods, notice that the method `findByPrimaryKey` coded in the home interface template returns an object of the bean class, whereas the method `ejbFindByPrimaryKey` returns an object of the bean's class *primary key*. A moment's reflection reveals the wisdom of this construct: The bean class method, invoked remotely through the home interface, should return a *reference* to the object. Because a primary key uniquely knows the entity bean, returning the primary key is the logical choice. However, remember that your client writes code to invoke the method on the home interface, not the bean interface. Part of the magic of EJB is the remote invocation of the bean method on the server when the client invokes the corresponding method from the home (or remote) interface.

- You can code `finder` methods for those entity beans with bean-managed persistence. Beans with container-managed persistence implement `finder` methods for you.

- Methods described in the remote interface have the same name as the corresponding methods in the bean interface. Do not prefix remote interface methods with `ejb`.

- Entity beans require that you code an `ejbPostCreate` method for every `create` method described in your home interface. Notice the presence of two `ejbPost create` methods: `ejbPostCreate` and `ejbPostAnotherCreate`. The `ejbPost` methods execute after the `create` methods.

- The last seven methods are the entity bean's *state management callback* methods, which are explained in detail in the next chapter. Here, you can say that these methods notify the entity bean when a bean is created, loaded from permanent storage, or destroyed. Because `javax.ebj.EntityBean` is an interface, you must supply a method body for these required state management methods, but you need not do anything meaningful. Many developers use *adapter classes* to provide empty method bodies in those rare cases in which beans are not already extending a superclass. You can read about the state management callback methods in Chapters 13, 14, and 15.

Let's put aside our discussion on entity beans for a moment and take a look at session beans.

Using Session Beans

A session bean is created by a client and, as a rule, exists only for the duration of a single session. A session bean performs tasks or processes on behalf of (usually) a single client, such as database access, number crunching, or some other relevant business process.

Although session beans can be transactional, they are not recoverable following a system crash. They can be stateless or they can maintain conversational state across methods and transactions. The container manages the conversational state of a session bean if it needs to be removed from memory. A session bean must manage its own persistent data.

Two types of session beans exist: *stateless* and *stateful* session beans.

Stateless session beans have no conversational state. In other words, stateless session beans are ignorant of the results of any prior method invocation. This ignorance may be put to good use. As a consequence, stateless session beans can be pooled to service multiple clients.

Stateful session beans possess conversational states. In other words, the activities (method invocations) taking place within a session bean are affected by prior method invocations.

Only one EJB client can exist per stateful session bean. However, stateful session beans can be saved and restored across client sessions. You use the getHandle method to return a bean object's instance handle, which can be used to

save the bean's state. Later, you may use the getEJBObject method to restore a bean from persistent storage.

Session beans do not have a primary key. However, you may assign your own identifier, such as a host name/port number pair of a remote connection, or even just a random number that the client may use to uniquely identify a given bean.

The characteristics of a session bean can be summarized as follows:

- Session beans model tasks or processes.

- Session beans are used by a single client.

- Session beans may be transaction-aware and can update data in an underlying database.

- Session beans are relatively short-lived. The lifetime of stateless session beans is limited to that of their client. However, stateful session beans may be saved and later restored across sessions.

- Session beans may be destroyed when the EJB server crashes. The client has to establish a connection with a new session bean object to resume its work.

What does the code for a session bean look like? The discusses a possible template for a session bean.

Session Bean Template

A sample session bean resembles the sample template entity bean in Listing 12-5. Listing 12-6 shows such a sample.

Listing 12-6: Template for defining a session bean

```
import java.rmi.* ;
import javax.ejb.* ;
import java.util.* ;

public class  MyBeanClass implements SessionBean {

   /** Bean properties Follow **/
   public Type1  beanProp1 ;
   public Type2  beanProp2 ;
           …
   public Typen  beanpropn ;
   public Typei  beanPropi ;
   public Typej  beanPropj ;
           …
   public Typez  beanpropz ;

   public void ejbCreate ( Type1 varType1,
                           Type2 varType2,
                                 …
                           Typen varTypen ) {
      beanProp1 = varType1 ;
      beanProp2 = varType2 ;
           …
      beanPropn = varTypen ;
      /**
         Initialize other properties
      **/
      initializePropsiThroughz() ;

   }

   public void ejbAnotherCreate ( Typei varTypei,
```

```
                                    Typej varTypej,
                                          …
                                    Typez varTypez ) {
    beanPropi = varTypei ;
    beanPropj = varTypej ;
         …
    beanPropz = varTypez ;
    /**
        Initialize other properties
    **/
    initializeProps1Throughn() ;

}

public Type1 getType1Property()
        throws RemoteException {
        return beanProp1 ;
}

public void setType1Property( Type1 varType1 )
            throws RemoteException {
    beanProp1 = varType1 ;
}
//
//Other accessor methods would follow
//
public boolean processWasSuccessful ( Type1 varType1,
                                          …
                                      Typen varTypen )
                throws RemoteException {
     /** Some code to do this process **/
     return successful ;

}

public void doAProcess (Typea varTypea,
                              …
                        Typez varTypez )
            throws RemoteException {
    /** Some code to do this process **/
}
//
//Other business methods would follow
//
//State management methods follow. No implementations
//provided for this example
public void setSessionContext( SessionContext sCTX ) {
    //
}
public void ejbActivate ( )
            throws RemoteException {
    //
}
public void ejbPassivate ( )
            throws RemoteException {
    //
}
public void ejbRemove ( )
            throws RemoteException {
    //
}
```

```
}
```

In this section, the naming rules, which describe the entity bean template, apply to session beans as well. The names of the session bean methods that correspond to the method descriptions in the home interface must be prefixed with `ejb`. Also, the names of the session bean methods corresponding to the remote method interface descriptions must match exactly.

Because session beans do not represent persistent data, you won't need the state management routines `ejbStore` and `ejbLoad`. You may set a session context with `setSessionContext`, but you need not remove the context.

It is unlikely that a session bean and an entity bean will use the same business methods or the same properties. The templates shown in Listings 12-5 and 12-6 are meant to illustrate the overall structure of an entity bean and a session bean, including showing you how to name your methods to correspond to those in the home and remote interfaces, as well as showing you which state management methods to include. The next two chapters will show concrete examples of both session and entity beans.

Comparing Entity Beans and Session Beans

Of course, the most dramatic difference between session and entity beans is that session beans manage information relating to a conversation between the client and the server, whereas entity beans represent and manipulate persistent application domain data. One way to conceptualize this is that entity beans replace the various sorts of queries used in a traditional two- or three-tier system, and session beans do everything else. Methods dealing with processing (tasks) found in entity beans have more to do with logic required in assigning persistent fields values than application workflow.

Session beans are supposed to be private resources, used only by the client that creates them. For this reason, a session bean hides its identity and is anonymous, in sharp contrast to an entity bean that exposes its identity through its primary key.

Table 12-1 summarizes some of the major differences between a session bean and an entity bean.

Table 12-1: Entity Beans and Session Beans Compared and Contrasted

Entity Bean	Session Bean
The data members of the entity bean represent actual data in the database.	The data members of the session bean contain conversational state information.
Entity beans may be used for database access by many clients.	A session bean may perform database access for a single client.
There is a one-to-one relationship between entity beans and a row in a relational table.	There is a one-to-one relationship between session beans and the bean's client.
An entity bean is persistent; it lives as long as its underlying data is stored in a database.	A session bean lives as long as the client session.
Entity beans survive system crashes.	Session beans do not survive system crashes.
Entity beans may not store client state information.	Stateful session beans store client state information.

In the next section is a discussion of some considerations you may ponder when using session beans and entity beans.

Using Session and Entity Beans

You may include methods to perform tasks in both entity beans and session beans. A natural question is what processes should be coded in each bean type.

Use entity beans for a persistent object model (to act as a JDBC wrapper, for instance). By doing so, you provide the rest of your application an object-oriented interface to your data model. Use session beans for application logic. Enable code in your entity beans to interact directly with the underlying database and enable code in your session beans to interact with the object-oriented layer presented by the entity beans.

Use session beans as the only interface to the client, providing a high-level interface to the underlying model. You should use entity beans to enforce the accuracy and integrity of your data in addition to an object representation of your data. The client invokes methods in your session beans, which runs processes that operate on the databases. This split reduces the pain of introducing new and changed processes because such changes are localized to the session beans.

Insist on reuse of entity beans. Although they may initially be hard to develop, over time they prove a valuable asset for your company. In contrast, expect little reuse of session beans. Often, session beans that model a set of processes are unique to a specific application domain. Of course, some thoughtful planning during design may result in increased reusability.

To summarize:

- Use session beans for application logic.

- Use session beans as the only interface to the client.

- Expect little reuse of session beans.

- Use session beans to control the workflow of a group of entity beans.

- Use entity beans to wrap all your JDBC code.

- Use entity beans to enforce accuracy and integrity of your database.

- Insist on reuse of entity beans.

The Primary Key Class

Every enterprise bean has a unique identifier. For entity beans, this unique identifier forms the identity of the information. For example, a `productIDnumber` might uniquely identify a product object in an inventory system. This is analogous to the concept of a *primary key* in a relational database system.

The *primary key class* is an abstraction that enables you to model entity beans as database entities. The primary key is a reference that you use to locate an entity bean that corresponds to a unique database entity. The only requirement is that objects of the primary key class be serializable.

Because session beans are not persistent, they do not use a primary key.

New with EJB 2.0: Message-Driven Beans

A message-driven bean is an asynchronous message consumer. The EJB container invokes methods in the message-driven bean. Message-driven beans do not have a home or a remote interface.

To a client, a message-driven bean is a Java Messaging Service (JMS) message consumer that implements some

business logic running on the server. Message-driven beans are not visible to the client and have no identity. A client accesses a message-driven bean through JMS by sending messages to the JMS destination (Queue or Topic) for which the message-driven bean class is the `MessageListener`.

Message-driven bean instances have no conversational state. This means that all bean instances are equivalent when they are not involved in servicing a client message. However, the instance variables of the message-driven bean instance can contain state across the handling of client messages. Examples of such state include an open database connection and an object reference to an EJB object.

The EJB container is responsible for the life-cycle of a message-driven bean instance; your client may have no knowledge of, or be capable of, controlling the birth and death of a message bean instance.

The topic of message-driven beans, along with other new features of EJB 2.0, is discussed in greater detail in Chapter 19, "The Proposed EJB 2.0 Specification."

Examining the EJB Interfaces

When you access an enterprise bean's properties or invoke its methods, you should not know where the bean lives, or what server or JVM the bean resides on. Of course, you need a mechanism to access the bean, regardless of where the bean lives, without knowing the bean's location. The mechanism requires the EJB developer to implement two interfaces and two classes. The interfaces define how a client interacts with the bean, or they define the client's view of the bean.

The client EJB object and the server EJB component implement the same interface. Hence, the client object and the server component look the same, even though they are separate classes. Think of using your garage door opener: you click a button on your (client-side) locally held remote control; the clicking activates a device in a remote (server) location, your garage, which opens your garage door. The button (client interface) on your remote looks the same as the button (server interface) on your garage wall; both buttons perform the same function.

Note   Java practitioners of RMI can see much in common with the concept of a client and server implementing the same interface. However, with RMI, you implement only one interface; with EJB, you must implement two.

The two interfaces you must implement to create an enterprise bean are called the *home* interface and the *remote* interface. Let's examine the home interface first.

Understanding the Home Interface

Each EJB component class has what is called a *home interface,* which defines the methods for creating, initializing, destroying, and finding EJB instances on the server. When an EJB client needs to use the services of an enterprise bean, the client creates the bean by invoking one of several `create` methods defined in its home interface. You may code several `create` methods that create objects in different states. When you no longer need a particular bean, you may invoke a `remove` method, also defined in its home interface.

Take a look at the methods that require implementation from the home interface below.

Home Interface Methods

The requirements for the home interface are not many. For starters, the home interface extends the interface `javax.ejb.EJBHome`, which extends `java.rmi.Remote`. The `EJBHome` interface defines a minimal set of requirements, meaning your beans extend `javax.ejb.EJBHome` and implement more methods than required by this interface. For example, you are likely to define one or more `create` methods to create instances of your enterprise bean, or `find` methods to locate bean instances.

Listing 12-1 below shows the interface `javax.ejb.EJBHome`.

Listing 12-1: The javax.ejb.EJBHome interface

```
public interface javax.ejb.EJBHome extends java.rmi.Remote {

    public EJBMetaData getEJBMetaData( )
           throws java.rmi.RemoteException ;

    public void remove ( Handle handle )
           throws java.rmi.RemoteException,
                  javax.ejb.RemoveException ;

    public void remove( Object primaryKey )
           throws java.rmi.RemoteException,
                  javax.ejb.RemoveException ;

    public javax.ejb.HomeHandle getHomeHandle( )
           throws java.rmi.RemoteException ;

}
```

The method `getEJBMetaData` returns an object of some class that implements the `EJBMetaData` interface. Objects of this class contain information associated with the bean's classes. For example, an `EJBMetaData` object enables a client to get the Java classes that create objects by using the home interface and the remote interface. EJB application developers may never invoke `getEJBMetaData`; the method is used primarily by tool and EJB container developers.

The `remove` methods remove an EJB object identified by the method argument. The `remove( Handle )` method requires an object from a class that implements the `Handle` interface. Objects of this class provide a reference to a networked EJB object. The `remove( Object )` method, used for entity beans only, requires an object that represents a bean's *primary key*. Calling this flavor of the `remove` method on a bean type other than entity bean results in `RemoteException` being thrown. Later, you can read about primary keys and entity beans.

As you might have guessed, the `javax.ejb.RemoveException` is thrown when the container or the object does not permit the object's removal.

The method `getHomeHandle` returns an object of a class that implements the `HomeHandle` interface. An object of this class is a handle, or reference, to the home object. The short story is that the home object is an object of a class that implements the home interface.

Note   Before you become overwhelmed with the preceding discussion on interfaces and classes, keep in mind that you do not create most of the previously mentioned classes. The EJB container creates most of these classes implementing these interfaces *for you*. You remember the EJB container — that nebulous entity that provides essential system services to your enterprise beans — don't you? Well, creating all these support classes is part of the services that the EJB container provides.

Methods defined in the home interface must follow the standard rules for RMI-IIOP remote interfaces. One rule is that any method that may be invoked across a network must throw `java.rmi.RemoteException`. Notice that all the methods described in Listing 12-1 throw this exception. You include a `throws java.rmi.RemoteException` clause in each method header in your home interface. You are free to, and probably will, include other exceptions in your `throws` clause. Another rule is that arguments passed to remotely invoked methods and returned types must be serializable. This makes sense, because such arguments and returned values are passed over a network, and Java RMI-IIOP uses object serialization to pass such data.

Note   Because EJB relies on dynamic class loading across the network, enterprise beans must follow certain security precautions. The issue of EJB security is covered in Chapter 16, "EJB Security."

You've read that the home interface is used to invoke life-cycle methods on your enterprise beans. The next section provides a template for a possible home interface implementation.

Home Interface Template

Your implementation of the home interface can resemble the template shown in Listing 12-2.

Listing 12-2: Template for defining a home interface

```
import java.rmi.* ;
import javax.ejb.* ;

public interface MyHome extends EJBHome {

    public MyBeanClass create( Type1 varType1,
                               Type2 varType2,
                                    …
                               Typen varTypen )
        throws CreateException, RemoteException ;

    public MyBeanClass anotherCreate( Typei varTypei,
                                      Typej varTypej,
                                          …
                                      Typez varTypez )
        throws CreateException, RemoteException ;

    public MyBeanClass findByPrimaryKey( myBeanClassKey aKey )
        throws FinderException, RemoteException ;

    public Collection findByDiffTypes( DiffType1 varDiffType,
                                       DiffType2 varDiffType2)
        throws FinderException, RemoteException ;

}
```

Listing 12-2 shows a couple of `create` methods and a couple of `finder` methods. As previously mentioned, you may code multiple `create` methods to create enterprise beans having a certain initial state. If you may need to access your beans by primary, secondary, and non-key fields, you may code multiple `finder` methods, as Listing 12-2 shows.

`CreateException` and `FinderException` are part of the EJB API. For now, keep in mind that every `create` method you implement requires that you code a `throws CreateException`, and that every `finder` method you implement requires that you code a `throws FinderException`.

The template in Listing 12-2 defines two `create` methods taking different parameters. All `create` method names must start with the word `create`.

The `findByPrimaryKey` method returns the unique instance of the bean accessed by the primary key argument. However, notice the returned type of method `findByDiffTypes: Collection`. The idea is that when searching on nonprimary key values, you may return 0 to many objects.

You implement the home interface by creating an object called, not surprisingly, the *home object*. An instance of the home object is created on a remote server and is made available to the clients for creating the enterprise bean. Let's take a look at the home object next.

The Home Object

The home object is an instance of a class that implements an interface that extends the `javax.ejb.EJBHome` interface. You use the home object to invoke methods that create enterprise beans on a remote server.

The home object has the needed `create`, `find`, and `remove` methods. These life-cycle methods invoke a corresponding `ejbCreate`, `ejbFind`, and `ejbRemove` method of the same signature in the actual enterprise bean from the bean class being created.

When a client wants to create an enterprise bean, it uses the *Java Naming and Directory Interface* (JNDI) to locate the home interface for the class of bean it wants. JNDI provides a service to any Java environment that enables Java programs to locate, use, and find information about resources by name. As of EJB 1.1, you must use JNDI to look up just about everything in your Java environment pertinent to EJBs. In the next chapter, you can see coding examples of using JNDI from a client to locate a home interface.

Once you have access to the home object, you can invoke methods that request the EJB container to create objects of your bean class to do work for the client. Barring any thrown exceptions, the server responds by creating an instance of the bean class remotely and returning a reference to an object. This object is an instance of a class that implements the same interface as the one implemented by the EJB component. That interface is called the *remote* interface, which you can read about in the following.

Understanding the Remote Interface

The remote interface makes the enterprise bean's business methods accessible to client objects. Methods defined in the remote interface do the work of the bean. The object of the class that implements the remote interface is called the EJB object. Once your client has a reference to an EJB object, your object can invoke that object's methods, which are implementations of the EJB component class's remote interface.

A short description of the methods required to implement the remote interface follows.

Remote Interface Methods

As with the home interface, the requirements for remote interface methods are few. Methods of a class that implement the remote interface typically extend `javax.ejb.EJBObject`, which extends `java.rmi.Remote`. Hence, remote interface methods must conform to the rules for RMI-IIOP methods described for the home interface methods in the previous section.

Listing 12-3 provides the code for the `javax.ejb.EJBObject` interface.

Listing 12-3: The javax.ejb.EJBObject interface

```
public interface javax.ejb.EJBObject extends java.rmi.Remote {

    public javax.ejb.EJBHome getEJBHome( )
            throws java.rmi.RemoteException ;

    public java.lang.Object getPrimaryKey( )
             throws java.rmi.RemoteException;

    public void remove( )
             throws java.rmi.RemoteException,
                    javax.ejb.RemoveException ;

    public javax.ejb.Handle getHandle()
            throws java.rmi.RemoteException ;

    public boolean isIdentical( javax.ejb.EJBObject )
            throws java.rmi.RemoteException ;
}
```

The method `getEJBHome` returns a reference to the home object for an enterprise bean.

The `getPrimaryKey` method returns a primary key for a bean. As you've read previously, primary keys are used only with entity beans.

Use the `remove` method to destroy the bean, thereby making available any resources previously held by this object.

The `getHandle` method returns a reference to this bean. You use `getHandle` as you would use the home interface method `getHomeHandle` — to get a reference to an EJB object that you can save and later use to reference the object.

The `isIdentical` method tests whether two enterprise beans are identical. As with any Java objects, you do not use the Java operator `= =` unless you are testing to see if two variables point to the same object handle in memory.

Recall that the remote interface exposes business logic methods to the client. What would a possible implementation of the remote interface look like? The <u>next section</u> shows a likely template.

Remote Interface Template

Your implementation of the home interface can resemble the template shown in <u>Listing 12-4</u>.

Listing 12-4: Template for defining a remote interface

```
import java.rmi.* ;
import javax.ejb.* ;

public interface MyRemote extends EJBObject {

   public Type1 getType1Property()
         throws RemoteException ;

   public void setType1Property( Type1 varType1 )
         throws RemoteException ;

   //Other accessor methods for other properties would follow

   public boolean processWasSuccessful ( Type1 varType1,
                                          …
                                          Typen varTypen )
         throws java.rmi.RemoteException ;

   public void doAProcess (Typea varTypea,
                             …
                           Typez varTypez )
         throws java.rmi.RemoteException ;
   //Other business method definitions would follow
}
```

No surprises here, right? The remote interface defines business methods, which include `accessor` methods for various bean properties as well as garden variety processing methods.

Now, let's talk some more about the previously mentioned EJB object.

The EJB Object

The EJB object is a server-side distributed object that implements the remote interface. Once your client has access to the EJB object, the client invokes its exposed methods. The methods get invoked remotely via RMI-IIOP and are

executed on the server. Then the client uses the EJB object as if it were a local object with the remote object doing all of the work.

You may be wondering how the EJB object and the home object are created, and how the server knows what methods to invoke when a client invokes a method implemented from a bean's remote and home interfaces. The following section addresses these, and other, questions.

About the Home and EJB Object

You may have figured out that you do not write code that implements the methods in the EJB and home objects. The EJB container generates the code that implements these methods. The container provider includes tools that automatically generate the required code.

Actually, you may never see the EJB and home objects, nor should you. The EJB specification describes the end-product of the code generated by the container, not the process or details. All you need to do is invoke the methods guaranteed by the interfaces to remotely invoke methods in your beans.

How does a client program create objects on a server? Operations involving the *life cycle* of server-side beans are the responsibility of the EJB container. The client program actually contacts the container and requests that a particular type of object be created, and the container responds with an EJB object that can be used to manipulate the new EJB component.

How does the container know what method of the enterprise bean to invoke when a client invokes a method from the home or remote interface? Before this question can be answered, a discussion about the creation of the bean class is in order.

Using the Enterprise Bean

An enterprise bean is a server-side software component that lives somewhere in a distributed environment. The component is the implementation of a Java class coded to follow the convention laid out by the EJB specification. The specification describes how to code the enterprise bean such that a client, or bean user, may use the bean to create EJB components on a remote server (or different JVM) and manipulate these components as if they were local objects. In the EJB distributed environment we have simplified writing clients, or applications, that access remote objects, thereby achieving the valued property of *location transparency*. While the developer writes client code that simply creates, uses, and destroys objects, these objects have counterparts executing on a server that do the actual work.

You must draw a distinction between an EJB *object* and an EJB *component*. The EJB object is a client-side object that accesses properties and invokes methods on an EJB server-side component.

Enterprise bean clients can be just about anything; applets, servlets, and other enterprise beans are possible, and likely, candidates. Using other enterprise beans as clients makes possible a "divide-and-conquer" approach to problem solving. The developer can divide the problem into bite-sized tasks, in which each task is implemented as a separate bean.

Describing the components of an EJB is difficult because these components refer to each other. In other words, it's hard to describe an EJB component without some knowledge of other EJB components. To alleviate potential confusion that may arise when describing the EJB components, a short description of these components follows.

EJB Components: The Short Story

To create and use an enterprise bean, you need to implement two interfaces and create two classes. The two interfaces are called the home interface and the remote interface. The two classes are called the bean class and the primary key class.

The class implementing the home interface provides methods that govern the enterprise bean's life-cycle. Here is where you invoke methods that find, create, and remove enterprise beans.

The class implementing the remote interface provides methods that describe the bean's business methods, which are the methods that do the work of the bean.

The bean class contains methods that do the work of the bean. Beans of the bean class come in three flavors: *entity* beans represent persistent data, *session* beans represent processes, and *message-driven bean*s represent asynchronous messaging with Java Messaging Services (JMS). In this chapter, the word *bean* without qualification refers to an instance of the bean class.

The primary key class is used to get references to entity beans. Imagine that you have created multiple instances of entity beans corresponding to rows in a relational database. The primary key would be used to access a particular bean instance.

As you read about each of the interfaces and classes discussed in the preceding, you learn how they interrelate to create bean instances and perform work.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

The "Make Money" Brokerage Application Revisited

This chapter starts with a brief discussion of how you can use a JSP page as a client to an enterprise bean. You'll take
a look at the JSP page that contains the custom tag to locate an EJB home object as well as enterprise bean business
method references in JSP scriptlets and expressions. You'll also see that the JSP page shown in this chapter is
strikingly similar to the JSP page shown in Listing 10-8 in Chapter 10.

The JSP pages that handle the logon and checks for a valid user/password combination is the same as the JSP
pages in Chapter 10. For our purposes here, you will concentrate on the user option to display the client's transaction
history, where you will see one approach in using beans together with JSPs to implement the transaction history
function. Next, you'll see the code that constitutes the custom tag used in the JSP page. You'll also see the code for
the tag library descriptor used to describe the tag.

The code for the enterprise bean comes next. Here's where the code that implements the business methods resides.
The business methods use very similar code to the methods of the `AcctHistory` bean shown in Listing 10-9 in
Chapter 10. Included is a deployment descriptor that describes the properties of the enterprise bean to the EJB
container.

You may want to quickly glance at Chapter 10, "The 'Make Money' Brokerage Application," if you need a quick
refresher on the application's functionality.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Using JSP Pages as EJB Clients

For a JSP page to be an EJB client, the page has to perform the tasks described in Chapter 18, "Creating EJB
Clients." These tasks are:

- Locate the EJB home object in order to obtain a reference to the EJB object.

- Direct the EJB container (by invoking methods on the EJB object) to create or remove enterprise beans.

- Direct the EJB container (by invoking methods on the EJB object) to invoke enterprise bean business methods.

We can create JSP pages to communicate with enterprise beans in two ways, by implementing either a JavaBean or a
custom JSP tag. Either way entails writing a Java class and coding methods that perform the EJB client tasks.

Coding a JavaBean results in the JSP pages using the bean containing `<jsp:usebean` tags and references to the
bean methods in the JSP pages in JSP scriptlets or expressions. Coding a custom tag means that the JSP pages will
contain references to the bean methods as JSP tags and, possibly, as code in JSP scriptlets or expressions.

The end result is the same whether you use JavaBeans or custom tags. However, as mentioned in Chapter 7, "JSP
Tag Extensions," using custom tags gives your JSP pages more of a natural look than using JavaBeans. The natural
look comes from having the page's functionality encapsulated in code invoked by tags as opposed to having the
page's functionality encapsulated in bean method invocations.

The example JSP page shown in this chapter uses a custom tag to locate the EJB home object.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Coding the JSP Page That Implements an EJB Client

The JSP page shown in Listing 20-1 is pretty close to the page used in Chapter 10. The major difference is that the
page shown in this chapter is a client to an enterprise bean.

The page uses a custom tag to locate an EJB object. Once the JSP page locates the EJB object through the `home`
interface and creates a reference to a session bean, the JSP page invokes the bean's business methods through the
EJB object.

Listing 20-1 lists the JSP page that acts as an EJB client. The page contains a custom tag. However, custom tag
usage is not a requirement for creating JSP pages that are clients to enterprise beans. The code in the page dealing
with the custom tag is shown in italics.

Listing 20-1: JSP Page containing a custom tag to locate an EJB object

```
<%@ page contentType="text/html"
         import="chapter20.*"
         errorPage="errorpageex1.jsp" %>

<%-- The taglib directive identifies the tag descriptor library --%>
<%-- and names a prefix used to denote the tags described in    --%>
<%-- the tag decscriptor library that are used in the page.     --%>

<%@ taglib uri="ch20taglib.tld" prefix="ch20" %>

<%-- Display the account activity for this customer --%>

<%
  CustomerBean    customer        = (CustomerBean)session.getAttribute("customer") ;
  String   customerName    = customer.getCustomerName() ;
  String   acctNum         = customer.getAcctNumber() ;
%>
<html>
<title>Transaction History for <%=customerName %> <%= acctNum %> </title>
<body bgcolor="#dddddd" >
<center>

<%-- Put in their pictures for the page top --%>

<jsp:include page="imagetable.html" flush="true" />

<%-- Here's the tag that looks up the EJB --%>

<ch20:lookupbean
     homeObject="ch20Home"
     JNDIlocation="java:comp/env/AcctHistory"
     homeObjectClass="AcctHistoryHome"
```

```
      remoteObjectClass="AcctHistory"
      remoteObjectName="anAcctHistory" />

<%-- Let's create an instance of the EJB using the home object --%>
<%
  anAcctHistory = ch20Home.create( acctNum ) ;
%>

<%-- The rest of the JSP is the same as the page in Chapter 10 --%>

<%

  if ( anAcctHistory.getHistoryThisAccount( acctNum ) ) {

%>
<p><%= customerName %>, here is a list of your transactions
<br>
<hr width="50%">
<form name="historyform" action="showcustoptions.jsp" method="POST">

<table>

<%
      while ( anAcctHistory.getNextHistRecord() ) {
%>
<tr><td>
 On <%= anAcctHistory.getColumn("transactiondate") %>, you traded
    <%= anAcctHistory.getColumn("numbershares") %> shares of
    <%= anAcctHistory.getColumn("security") %> on a
    <%= anAcctHistory.getColumn("transactiontype") %> order.
    </td>
 </tr>
 <tr bgcolor="red"><td> </td> </tr>

<%
      }

  }
%>
<tr>
  <td><p>
  <input type="submit"
        name="Return" value="Return to Customer Options">
  </td>

</tr>
</table>
</form>
<hr width="50%">
</center>

</body>
</html>
```

Recall from Chapter 7 that JSPs require a `taglib` directive when using custom tags. The following line is the `taglib` directive used in the JSP shown in Listing 20-1.

```
<%@ taglib uri="ch20taglib.tld" prefix="ch20" %>
```

The `taglib` directive provides a reference to the tag library descriptor file, `ch20taglib.tld`, and a prefix used to

reference the library throughout the page, `ch20`. Every custom tag described by the tag library descriptor file used in this page must be prefixed with the value of the `prefix` attribute, or `ch20`.

Here, we're using a custom tag to locate an EJB object that the EJB container will associate with an instance of your bean. The tag contains information, encoded in attributes, that enable the tag library code to locate a suitable EJB object. The tag listed here is the tag used in [Listing 20-1](#).

```
<ch20:lookupbean
    homeObject="ch20Home"
    JNDIlocation="java:comp/env/AcctHistory"
    homeObjectClass="AcctHistoryHome"
    remoteObjectClass="AcctHistory"
    remoteObjectName="anAcctHistory" />
```

Recall that the first task of an EJB client is to locate the home object, which the client uses to obtain a reference to the EJB object. The `homeObject` attribute provides a reference that the JSP page uses later to create references to enterprise beans — in our case, just a single session bean.

The `JNDILocation` attribute provides the location of the bean class as the default naming context. Once we have the naming context, we can get a reference to the home object by invoking the `lookup` method of class `javax.Naming.Context`.

The `lookup` method returns the reference to the home object as an object of class `Object`. We need the name of the `home` interface to cast the home object reference to a usable form. Therefore, we include an attribute in our custom tag, named `homeObjectClass`, that provides the `home` interface name.

An EJB client uses the methods in the `home` interface to create EJB objects. The EJB object is an object of the `remote` interface. The `remote` interface contains descriptions of the enterprise bean's business methods. Therefore, for a client to invoke bean business methods defined in the `remote` interface, the client needs to know the name of the `remote` interface. Hence, we provide the `remote` interface as an attribute value aptly named `remoteObjectClass` in our custom tag. While we're at it, we need a name for our EJB object, which we'll supply as a value of the attribute `remoteObjectName`. The tag library code requires these parameters to locate and narrow the EJB object to the correct class.

With an appropriate EJB object reference handy, the JSP creates a reference to the bean by using the home object's `create` method, as shown in the following line of code:

```
anAcctHistory = ch20Home.create( acctNum ) ;
```

Use the account number from the customer object as a unique client identifier.

Once we create the EJB object, we're ready to request the EJB container execute business methods. These methods will access the transaction history database and return data to the JSP page. The page will format and display the returned data just like the page shown in [Listing 10-8](#) of [Chapter 10](#).

Now let's take a look at the code that implements the custom tag.

The Tag Library Code

The tag library code implements the EJB client methods that locate and narrow a home object. To create our custom tag, we'll need to implement a class that provides the tag's functionality, or the tag class. We'll also need a tag library descriptor file (`tld`) that describes the tag class and allows you to use the custom tags in your JSP pages.

The Tag Class

The tag class is a class that extends the convenience class `TagSupport`. Recall from Chapter 7 that class `TagSupport` is an implementation of the tag interface for custom tags that do not contain a tag body. The name of the tag class is not terribly important. What is important is that elements in the `tld` associate the name of the tag class with the name of the custom tag. Listing 20-2 shows the code for the tag class.

Listing 20-2: Code for the tag class

```
package chapter20 ;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class LookupCh20Bean extends TagSupport
{
      private String JNDIlocation;
      private String homeObjectClass;
      private String remoteObjectClass;
      private String remoteObjectName;

      //Set the class of the home interface.

      public void setType(String type)
      {
        this.homeObjectClass = type;
      }

      //Set the JNDI location of the home.

      public void setLocation(String location)
      {
        this.JNDIlocation = location;
      }

      //Set the class of the remote interface.

      public void setRemoteType(String remoteType)
      {
        this.remoteObjectClass = remoteType;
      }


      // Set the name of the remote interface variable.

      public void setRemoteName(String remoteName)
      {
        this.remoteObjectName = remoteName;
      }

      public int doStartTag() throws JspException
      {
        try
        {
          InitialContext context = new InitialContext();

          // Get the home interface class using the
          // proper (the page's) ClassLoader.
          Class homeClass =
                Class.forName(this.homeObjectClass, true,
                  pageContext.getPage().getClass().getClassLoader());
```

```
      // Lookup the home object
      Object homeObject = context.lookup(this.JNDIlocation);

      // Narrow the home object
      EJBHome home =
      (EJBHome)javax.rmi.PortableRemoteObject.narrow(homeObject,
                                                homeClass);

      // Put the home object into the PageContext
      pageContext.setAttribute(this.getId(), home);
    }
    catch(NamingException e)
    {
      throw new JspException("Error looking up home at " +
                               this.JNDIlocation);
    }
    catch(ClassNotFoundException e)
    {
      throw new JspException("Home class not found: " +
                               this.homeObjectClass);
    }

    return SKIP_BODY;
  }
}
```

The tag library code is straightforward. The class contains methods to set the values of the tag attributes to object properties and to invoke the `lookup` and `narrow` methods. Notice how the code wraps up the exceptions as `JSPExceptions`.

The tag class uses the `pageContext` variable to hold a reference to the home object. The JSP page needs the reference to the home object to create an EJB object by invoking the home object's `create` method.

The tag class contains code to locate a home object. There's no code in our tag to access the session bean (or access the EJB object that, in turn, requests the EJB container to execute a remote method). There's no code in our tag to create EJB objects. The code to request bean business method execution and to create EJB objects is included in the JSP page.

The Tag Descriptor File

The `tld` describes the tag's properties, notably, the tag's attributes. Listing 20-3 shows the `tld` for our enterprise bean locator tag.

Listing 20-3: tld for the bean locator tag

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC
      "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
      "web-jsptaglib_1_1.dtd">

<taglib>
  <jspversion>1.1</jspversion>
  <tlibversion>1.0</tlibversion>

  <shortname>ch20lib</shortname>
  <urn></urn>

  <info>
```

```
        Tag Library That Provides Info on Tags That Display Trade
        History For Customers
    </info>
    <tag>

        <name>lookupbean</name>

        <tagclass>LookupCh20Bean</tagclass>

        <teiclass></teiclass>

        <bodycontent>empty</bodycontent>

        <info>This tag locates a home object for a session bean</info>

        <attribute>
            <name>homeObject</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>

        <attribute>
            <name>JNDIlocation</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>

        <attribute>
            <name>homeObjectClass</name>
            <required>true</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>

        <attribute>
            <name>remoteObjectClass</name>
            <required>true</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>

        <attribute>
            <name>remoteObjectName</name>
            <required>true</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>

    </tag>
</taglib>
```

The `tld` describes the tag as an *empty* tag, as shown by the `bodycontent` element (`<bodycontent>empty</bodycontent>`). Also, all attributes are required, as shown by the `required` element, a child element of the `attribute` element (`<required>true</required>`). Finally, no attribute contains JSP expressions or JSP tags as shown by the `rtexprvalue` element (`<rtexprvalue>false</rtexprvalue>`).

Now let's look at the components of the enterprise bean.

The Enterprise Bean

The enterprise bean requires essentially the same methods as the JavaBean used in Chapter 10. The EJB requires

methods to determine if the client has history records and to access a transaction history record. In addition, to satisfy the architectural requirements of an enterprise bean, the bean must provide implementations of all the methods of a bean. Some of the required method implementations are "dummy" implementations, or a method signature with a pair of empty braces. Nonetheless, the EJB architecture requires that you implement the laundry list of required methods, even if you code dummy implementations.

The Home Interface

The `home` interface contains a reference to the `create` method used by the JSP page to request that the EJB container create a reference to the home object. Listing 20-4 shows the small but vitally important code that constitutes the `home` interface.

Listing 20-4: The home interface

```
import javax.ejb.*;
import java.rmi.*;


public interface AcctHistoryHome extends EJBHome
{
  public AcctHistory create( String acctID)
                   throws RemoteException, CreateException;
}
```

Notice that the `create` method requires an argument and that your bean is bound to a particular customer by the account ID.

The Remote Interface

Recall that the `remote` interface contains references to the business methods contained in the bean. You request that the EJB container invoke a bean method by invoking a like-named method described in the `remote` interface. The reference to the EJB object, obtained through the `home` object, allows your EJB client to issue requests for remote enterprise bean method invocation. Listing 20-5 shows the code for the `remote` interface.

Listing 20-5: The remote interface

```
import javax.ejb.*;
import java.rmi.*;

//Remote Interface

public interface AcctHistory extends EJBObject
{
  public  boolean getHistoryThisAccount( String accountID )
        throws RemoteException ;
  //Disconnect from database — close connection, stmt, resultset
  public void takeDown( ) throws RemoteException ;
  public boolean getNextHistRecord() throws RemoteException ;
  //Get a column for display in the HTML table
  public String getColumn( String inCol ) throws RemoteException ;

}
```

Notice that the four methods described in the `remote` interface are those methods used in the JSP page to list the customer's transaction history.

The Enterprise Bean Code

The enterprise bean is a stateful session bean that looks very similar to the code shown in . shows the EJB code that retrieves the customer's transaction history.

Listing 20-6: Session bean code that displays transaction history

```
import java.sql.* ;
import javax.ejb.*;
import javax.naming.* ;
import javax.sql.* ;

public class AcctHistoryBean  implements SessionBean {

    private ResultSet myResultSet ;

    private SessionContext mySessionContext;
    private Connection        myConn = null ;
    private Statement         stmt   = null ;

    //Use the account number to access the history table.
    public  boolean getHistoryThisAccount( String accountID )
                    throws Exception {
      String histQuery  = "select accountid, transactiondate," + ,
             "transactiontype, security, numbershares " +
             "from transactionhistory where accountid = " ;

      String query = histQuery + "'" + accountID + "'";
      try {
        myConn       = getConnection() ;
        stmt         = myConn.createStatement();
        myResultSet = stmt.executeQuery( query );

        return myResultSet != null ;
      }
      catch ( SQLException sqlEx ) {
          throw new EJBException( sqlEx ) ;
      }

  }

  public boolean getNextHistRecord() throws Exception
    {
     try {
         return myResultSet.next();
      }
      catch (SQLException sqlEx ) {
          throw new EJBException( sqlEx ) ; }

    }

  public String getColumn( String inCol) throws Exception
    {
    try {
      return myResultSet.getString(inCol);
    }
    catch (SQLException sqlEx ) { throw new EJBException( sqlEx ) ;
    }

  }

   public void takeDown() throws Exception
```

```
    {
    try {
        if ( stmt != null )    stmt.close() ;

            if ( myConn != null ) myConn.close() ;
    }
    catch ( SQLException sqlEx ) {sqlEx.printStackTrace();}
    }

    //Utility method from Chapter 14 that establishes a connection

    private Connection getConnection() {
        String jndiDBName = "java:comp/env/jdbc/stock" ;
        try {
            InitialContext jndiCtx = new InitialContext() ;
            DataSource      stockDS =
                        (DataSource) jndiCtx.lookup(jndiDBName);
            return stockDS.getConnection() ;

        }
        catch (NamingException ne) {throw new EJBException( ne );}
        catch (SQLException sqlEx) {throw new EJBException( sqlEx );}
    }
    // EJB methods. Most of these methods are dummy implementations.
    // However, the SessionBean interface requires the implementing
    // class provide implementations of ALL the methods.

    public void ejbActivate()
    {
    }

    public void ejbPassivate()
    {
    }

    public void ejbRemove()
    {
    }

    public void ejbCreate() throws CreateException
    {
    }

    public void setSessionContext(SessionContext sessionContext)
    {
        mySessionContext = sessionContext;
    }

}
```

Aside from the extra code required for bean implementation and the method used to locate the database, the code shown in is the same as that used in .

The method `getConnection` used to locate and establish a database connection differs from the methods used in to establish a connection. With enterprise beans, we can code database location information in the bean's deployment descriptor. Enterprise beans may access the location information with the same `lookup` method used to locate a home object.

The Deployment Descriptor

The deployment descriptor shows the location of the database as well as providing some details on the EJB's components. Listing 20-7 shows the deployment descriptor.

Listing 20-7: The EJB deployment descriptor

```xml
<?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>AcctHistoryBean</ejb-name>
      <home>chapter20.AcctHistoryHome</home>
      <remote>chapter20.AcctHistory</remote>
      <ejb-class>chapter20.AcctHistoryBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <resource-ref>
        <description>Brokerage Database Data Store</description>
        <res-ref-name>jdbc/stock</res-ref-name>
        <res-type>javax.sql.DataSource </res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>AcctHistoryBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>

</ejb-jar>
```

The deployment descriptor contains three main groups of information: bean particulars, resources used by the bean, and transaction information.

The bean particulars include the bean type (stateful session, stateless session, entity, or message-driven), the `home` and `remote` interface names, the bean class name, and how transactions are managed.

The resources used by the bean are described as child elements of the `resource-ref` element. The content of the `res-type` element, `javax.sql.DataSource`, tells the container this resource is a reference to a database.

The transaction information is described by child elements of the `container-transaction` element.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Using Stateless Session Beans

The EJB developer can use stateless session beans to implement a business process that does not require extended
contact or conversation with a client. In other words, your stateless session beans may perform a task that requires no
knowledge about a particular client.

Stateless session beans are ignorant of the results of any prior method invocation. Some examples of tasks that can
be performed by a stateless session bean are:

- Calculating the price for an item

- Evaluating the current value of a portfolio

- Converting an RGB color space value to an HSV color space value

Characteristics of Stateless Session Beans

As mentioned previously in this chapter, stateless session beans are blissfully unaware of client particulars. This
ignorance leads the EJB container and server developers to devise strategies for pooling session bean instances.

Pooling Stateless Session Beans

Because any stateless session bean can service any client, the container may create instances of a stateless session
bean ahead of time and keep them "on ice." When a client invokes a stateless session bean method through the
bean's remote interface, the container may have a bean instance available for use. The advantage of having the
container pull a bean from a ready-made pool of instances, as opposed to creating an instance on demand, is
performance. The pooling strategy is faster than the create-on-demand strategy because a small number of stateless
session beans may service a large number of clients.

Using the create and ejbCreate Methods with Stateless Session Beans

In the previous chapter, you read about creating instances of beans by invoking `ejbCreate` methods by invoking a
`create` method in the home interface. Because stateless session beans have no concept of state, they do not retain
any knowledge of parameters passed to the bean via `ejbCreate`. Therefore, you should not code stateless session
bean `create` methods in your home interface and any corresponding `ejbCreate` methods in your bean class that
require parameters.

Caution You may be thinking that a stateless session bean shouldn't use instance variables since it doesn't track state. You are free to include instance variables in your session bean. However, each method in a stateless session bean exposed to clients must be provided with *all* information needed for your bean to do its work. You cannot pass data by way of arguments to `create` methods, so you are left with passing arguments when you invoke bean methods.

Instance variables for stateless session beans do not hold information that can be used for subsequent method invocations of the bean. That is, the stateless bean has no knowledge from one method invocation to the next, or any knowledge of the previous method invocation. If a stateless bean contains three business methods exposed to a client through the bean's remote interface, a client typically invokes only one of those methods. Once a client invokes a bean method, the data contained in any of the bean's instance variables cannot be used by a subsequent method invocation.

Life Cycle of Stateless Session Beans

The simplicity of stateless session beans is mirrored in its life cycle. Stateless session beans exist in only two states: the *nonexistent* state and the *method-ready* state.

You don't have to be a rocket scientist to glean what the nonexistent state means. This state is used to describe when a bean is no longer needed and its existence is ended with a call to `ejbRemove`.

When a bean is in the method-ready state, the bean instance is in a pool or has been created to service a client request. Such beans are ready to perform work on behalf of their clients.

Another way of looking at the bean instance pool is that a stateless session bean is bound to a particular EJB object, which is the entity managed by the container that works on behalf of the client. The creation and destruction of beans may or may not correspond to the creation and destruction of EJB objects. The client interacts with interfaces that interact with the EJB container that manipulates EJB objects. The particular EJB container implementation may involve bean instance creation for every client or (more likely) creating a pool of bean instances and binding these instances to EJB objects.

As previously mentioned, EJB containers usually create multiple instances of stateless session beans for performance reasons. The container manages assigning bean instances to clients based on client requests. If the container pools beans, the container issues the `Class.newInstance` method to create an instance of the bean and put that instance in the bean pool. Next, the bean receives a reference to its `EJBContext` when the container invokes the `SessionBean.setSessionContext` method. When a client issues a call to `create` in the bean's `home` interface, the container invokes the `ejbCreate` method (with no arguments, remember?). At a time of the container's choosing, the container purges the bean by issuing an `ejbRemove` method.

In truth, the client does not really operate on the bean instance. The client, by making calls to methods in the bean's home (and remote) interface, operates on the *underlying* EJB object. The client may view the container activity as creating and purging beans. That's part of the EJB magic — hiding the implementation details from the client and, of course, the client and developer's code.

Using Stateful Session Beans

Stateful session beans model business processes where the concept of a conversational state takes center stage. In other words, the activities (method invocations) taking place within a session bean are affected by prior method invocations.

Characteristics of Stateful Session Beans

A stateful session bean needs a way of identifying one particular client among many. The simplest (and surprisingly effective) way for EJB containers to cause stateful session beans to maintain client identity is to bind the underlying EJB object to a particular client for the lifetime of the bean.

Pooling (or not) Stateful Session Beans

EJB containers typically do not create pools of stateful session beans and swap the bean instances among multiple EJB objects. Once a client requests the creation of an EJB object by invoking the home interface's `create` method, the EJB object is bound to that client for life.

Although the EJB object corresponding to the stateful session bean is dedicated to a single client, the EJB container may swap bean instances to and from storage or perform whatever optimizations the container creators deem necessary.

So, do EJB containers pool stateful session beans or not? The practical result of the container swapping stateful session beans is the same as pooling bean instances. However, there are differences between the swapping of stateful session beans and the pooling of stateless session beans. Later in this chapter, the section titled "Using the `javax.ejb.SessionBean` Interface" will discuss these differences.

The create and ejbCreate Methods with Stateful Session Beans

Unlike their stateless cousins, stateful session beans may accept data via a `create` method in the bean's home interface (and the corresponding `ejbCreate` method in the bean class). Stateful session beans remember data from one method invocation to the next for the same client. It makes sense to initialize EJB objects corresponding to session beans with data peculiar to a particular client.

In addition, you may code multiple `create` methods, each with a different set of parameters (or no parameters) when instructing the container to create an EJB object to serve as an agent for a stateful session bean.

Instance Variables and Stateful Session Beans

Instance variables of stateful session beans may serve different uses than instance variables of stateless session beans. Because stateful session beans are used to maintain a sense of session with a single client, instance variables

may hold information relevant to successive method invocations. If a stateful session bean exposes three methods to a client through the bean's remote interface, any of these method invocations may use instance variables to store data for use by subsequent method invocations.

Life Cycle of Stateful Session Beans

One important difference between the life cycle of a stateful and a stateless bean is that stateful beans are not pooled whereas stateless beans are pooled. From the client's perspective, the client has access to a stateful bean (do you recall that the client has access to the EJB object?) for the life of the client session. If the server or EJB container decides to swap the stateful session bean, the server or container *must* preserve the state of the conversation between the EJB object and the client. The container may decide to perform the swap to conserve server resources or to implement a "last-used" strategy to decide when to remove beans from memory.

Activating and Passivating Beans

The process of swapping out a stateful session bean from memory to storage is called passivation. It's important to understand that passivation is the process that an EJB container uses to preserve the state of a stateful session bean (or an entity bean) when the container swaps the bean out to storage.

Do not confuse passivation with storing a piece of persistent data. When you save data to persistent storage, you hold on to that data for an indefinite period of time. The persistent storage is usually a database. Although passivation usually includes writing out the bean to storage, the intent is not to save the stateful session bean for an indefinite period of time. Rather, the intent is to make room for other beans that, at the present time, are more used than the passivated bean. The server reloads the passivated bean when some client needs it. The passivated bean is not data per se; the passivated bean represents the state of a client session with the enterprise application.

The opposite of passivation is the process called *activation*. A stateful session bean is activated when the client issues a method invocation through the bean's remote interface that requires the services of the EJB object. The server is responsible for reloading the previously passivated EJB object from storage to memory so that the methods of the EJB object may be invoked and work may be done.

Because stateless session beans do not maintain any state, the concepts of passivation and activation are not relevant with stateless session beans.

The activation and passivation process affects the makeup of instance variables. You may have figured out that the server uses Java's object serialization to passivate and activate a bean. Hence, any instance variable critical to describing the conversational state of the bean must be *serializable* or be declared as a Java primitive type.

Using the javax.ejb.SessionBean Interface

To write session beans, you must create a class that implements the `SessionBean` interface. The `SessionBean` interface contains callbacks that govern the container's treatment of your bean. Please note that the client never calls these methods. In other words, the methods in the `SessionBean` interface are not made available to the client through the remote interface.

As with other Java interfaces, you may not need implementations of all of the methods defined in the `SessionBean` interface. If this is the case, you may code an empty implementation for unneeded methods. For your convenience, some EJB servers contain adapter classes such as those used in GUI event handlers. Of course, you may code your own adapters as well.

Listing 14-1 shows the `SessionBean` interface.

Listing 14-1: The SessionBean interface

```
public interface javax.ejb.SessionBean
        extends javax.ejb.EnterpriseBean {

  public abstract void setSessionContext( SessionContext ctx )
          throws java.rmi.RemoteException ;

  public abstract void ejbPassivate( )
          throws java.rmi.RemoteException ;

  public abstract void ejbActivate( )
          throws java.rmi.RemoteException ;

  public abstract void ejbRemove( )
          throws java.rmi.RemoteException ;


}
```

Your bean class implements the methods in the `SessionBean` interface *and* the business methods in the bean's `remote` interface. You can take a look at the methods in the `SessionBean` interface now. Remember that *all* of the methods that follow are invoked by the EJB container.

ejbActivate

The `ejbActivate` method is a callback invoked by the container *immediately after* activating a bean. Of course, you remember that the container activates a bean by loading a previously saved (passivated) bean from storage into

memory in response to a client request through the EJB object's remote interface. Once the container resurrects your bean, the container invokes the bean class's `ejbActivate` method.

Because activation and passivation do not apply to stateless session beans, you must code empty implementations for `ejbActivate` and `ejbPassivate` for stateless session beans.

Use `ejbActivate` to acquire the resources your bean needs to accomplish its appointed duty. For example,

```
import javax.ejb.* ;
//Other imports as required for the bean
public class SampleStatefulSessionBean implements SessionBean {

    public void ejbActivate() (
       //Open Database connections
       //Open files, load properties, etc.
    }
    //Other required SessionBean interface methods and
    //business methods defined in the bean's home
    //interface follow
}
```

ejbPassivate

The `ejbPassivate` method is invoked by the EJB container *immediately prior to* the passivation of the stateful session bean. As we read earlier in the section titled "Activating and Passivating Beans," passivation is the process, involving Java serialization, in which the container writes a bean to storage.

For stateless beans, you code dummy implementations as follows:

```
import javax.ejb.* ;
//Other imports as required for the bean
public class SampleStatelessSessionBean implements SessionBean {

    public void ejbActivate() (
       //Do not code any statements here
    }
    public void ejbPassivate() (
       //Do not code any statements here
    }
    //Other required SessionBean interface methods and
    //business methods defined in the bean's home
    //interface follow
}
```

For a stateful session bean, you would release resources held by the bean in the `ejbPassivate` implementation. These resources would then be reacquired in your `ejbActivate` method implementation.

ejbRemove

The EJB container calls the `ejbRemove` method immediately prior to removing a session bean instance. Do not confuse removing the instance with passivating the instance. When the container passivates the bean, the container may bring the bean back to satisfy a client request. When the container removes the bean, that bean is dead, gone, finished, whacked.

setSessionContext

The bean's `SessionContext` object provides methods for the bean to interact with the EJB container. Later in this chapter you can learn more about the session context. For now, it's enough to know that the session context is set by a call to the `setSessionContext` method.

A commonly used technique is to declare a session context outside any method and assign the session context to this object. For example, the following code does the trick:

```
import javax.ejb.* ;
//Other imports, etc.
public class SampleSessionBean implements SessionBean {
   private SessionContext sCtx ;

   public void setSessionContext( SessionContext ctx ) {
      sCtx = ctx ;
   }
   //Other SessionBean and business method implementations
   //follow
}
```

ejbCreate

Strictly speaking, the `ejbCreate` method is not included in the `SessionBean` interface. For session beans, however, you *must* code at least one `create` method. Thus, a discussion of `create` and `ejbCreate` is included here.

To direct the container to create beans for your use, issue a `create` method implemented on the bean's `home` interface. When the client invokes the `create` method, the container invokes a corresponding `ejbCreate` method, which makes an EJB object available to the client.

From the client's perspective, the call to `create` creates a session bean. But that may or may not be true; the truth depends on the EJB container implementation. If the client requires access to an enterprise bean, the client issues a call to `create`, which, in turn, causes the container to issue a call to an `ejbCreate` method.

As previously mentioned, for stateless session beans, you may code only the garden-variety, parameterless `create` and `ejbCreate` methods. For stateful session beans, you are free to code several `create` and corresponding `ejbCreate` methods. The only requirement is that the `create` and `ejbCreate` methods must have matching argument lists and all return void.

For example, if you coded the following `create` method signature in your `home` interface:

```
public void create( Customer aCust, double payAmt )
      throws RemoteException, CreateException {
```

You would code the following method signature in your bean class:

```
import javax.ejb.* ;
//Other imports as required for the bean
public class SampleStatefulSessionBean implements SessionBean {

   public void ejbCreate( Customer aCust, double payAmt) (
      //Initialize bean with arg variables or use as you see fit
   }
   //Other required SessionBean interface methods and
   //business methods defined in the bean's home
   //interface follow
```

```
}
```

As an aside, you must code `create` methods in your `home` interface that throw `CreateException`. Recall that all methods coded in the `remote` and `home` interfaces must throw a remote exception. In Chapter 15, "EJB Entity Beans," when you read about entity beans, you'll learn that `finder` methods coded in the `home` interface must throw a `FinderException`.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

The javax.ejb.SessionContext Interface

No bean is an island. Beans work and play within containers. You might imagine that beans and their containers
engage in a fair amount of communication. The `SessionContext` interface extends the more general `EJBContext`
interface. You can read about the `EJBContext` interface in . The following section covers the
`SessionContext` interface.

getEJBObject

As it turns out, your beans may need to communicate with the container when your beans need to invoke methods in
other EJBs. Session beans have access to a `SessionContext` object to facilitate interbean communication. This
object enables your bean to invoke one method, `getEJBObject`, which enables your bean to pass a reference to
itself to other EJBs.

You might be thinking, "Why can't the session bean use the `this` keyword to pass a reference to itself to another
bean?" Recall that clients do not invoke bean methods directly on the bean; clients invoke bean methods through a
reference to the `remote` interface, or its EJB object. If the bean returned `this`, it would be returning a reference to
the bean itself and not to its EJB object. The bean must return a reference to the EJB object, hence the purpose of the
`getEJBObject` method.

Top

← Prev   Next →

Creating a Stateless Session Bean

The code in this section implements a system in which a customer can buy or sell stocks. The application executes one of two methods from a stateless session bean: buy and sell. What follows is the code for the home and remote interfaces, support classes, and a sample client that invokes the EJB object through the remote interface.

The buy and sell methods interact with a relational table called stocktable. Here are the relevant columns for table stocktable used in the stateless session bean example:

```
Create table stocktable(

    customerID          NUMBER          Primary Key,
    sharesymbol         CHARACTER( 4 ) Primary Key,
    numSharesOnAcct     NUMBER
)
```

Although simple, the table suffices for our sample stateless bean application.

The payment bean uses two application classes: StockCustomer and StockTradeResult. Listings 14-2 and 14-3 show the code for these classes.

Listing 14-2: Code for the StockCustomer class

```java
package chapter14.stocktrader ;
import java.io.* ;

public class StockCustomer implements Serializable {
    private String custName ;
    private int    custID   ;
    private String custAcctNum ;

    public StockCustomer( String cName, int cID, String cAcctNum ) {
        custName    = cName ;
        custID      = cID ;
        custAcctNum = cAcctNum ;
    }

    public String getCustName () {
        return custName ;
    }
    public int getCustID () {
        return custID ;
    }
    public String getCustAcctNum () {
        return custAcctNum ;
    }
```

```
}
```

Notice that this class implements `java.io.Serializable`, because the code passes objects of class `StockCustomer` to remote methods.

Listing 14-3 shows the class `StockTradeResult`. An instance of this class is returned to the client after each trade. As you'd expect, `StockTradeResult` implements `java.io.Serializable`.

Listing 14-3: Code for the StockTradeResult class

```
package chapter14.stocktrader;

import java.io.Serializable;


public final class StockTradeResult implements Serializable {

  private String statusMsg ;     //Successful or not?.
  private int    numberTraded;  //# of shares bought or sold.
  private String custName    ;  //Like it says.
  private String stockSymbol;   //Ditto.

  public StockTradeResult(String status,
                          String cName,
                          int nt,
                          String ss) {
    statusMsg    = status ;
    custName     = cName ;
    numberTraded = nt;
    stockSymbol  = ss;
  }
  public String getStatusMsg()     { return statusMsg; }
  public String getCustName()      { return custName; }
  public int    getNumberTraded()  { return numberTraded; }
  public String getStockSymbol()   { return stockSymbol; }
}
```

Coding the Remote Interface

Listing 14-4 provides the code for the remote interface `StockTrader`. Notice that both methods return an object of class `StockTradeResult`.

Listing 14-4: Code for the remote interface

```
package chapter14.stocktrader;

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

/*
  Remote interface for the Stock Trader. These methods
  get implemented in the bean class StockTraderBean
 */
public interface StockTrader extends EJBObject {
```

```
  public  StockTradeResult buy (StockCustomer aCust,
                                String   stockSymbol,
                                int      numShares)
     throws RemoteException;


  public StockTradeResult sell (StockCustomer aCust,
                                String   stockSymbol,
                                int      numShares)
     throws RemoteException;
}
```

Coding the home Interface

Listing 14-5 shows the code for the `home` interface. All you need is a `create` method as required by the EJB specification for session beans. Remember that you cannot pass arguments to the `create` method for stateless session beans.

Listing 14-5: Code for the home interface

```
package chapter14.stocktrader;

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

public interface StockTraderHome extends EJBHome {

   StockTrader create() throws CreateException, RemoteException;
}
```

Coding the Bean Class

The code for bean class `StockTraderBean` is shown in Listing 14-6. Implement the required methods from the `SessionBean` interface plus business methods defined in the `remote` interface.

Listing 14-6: Code for the bean class

```
package chapter14.stocktrader;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.EJBException ;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.io.* ;
import java.sql.* ;
import java.rmi.RemoteException ;
import javax.sql.DataSource ;
import java.util.* ;

public class StockTraderBean implements SessionBean {
```

```java
/** Declare the connection and preparedStatement object so we
    can use them in multiple methods. Also, we want to
    reuse the connection because we'll issue two SQL statements
    here.
    For a buy order, we need to determine if the customer has
    this stock already. If so, we issue an UPDATE sql. If not,
    we issue an INSERT SQL.
    For a sell order, we need to see how many shares the customer
    has to sell. If not enough, we issue a StockTraderException.
    If enough, we issue an UPDATE to change the shares on
    account.
*/
Connection         payConn = null ;
PreparedStatement payPS   = null ;

private SessionContext sCtx;
/** These are the required methods of the SessionBean interface.
    Although we don't need them we must supply a dummy
    implementation.
 */
public void ejbActivate() { } ;
public void ejbRemove() { } ;
public void ejbPassivate() { } ;
/*
 */
public void setSessionContext(SessionContext ctx) {
  sCtx = ctx;
  /* When the bean becomes available, the container makes a
     session object available. Here's a good place to perform
     startup tasks. Recall that stateless session beans live
     and die for a single method invocation.
     Let's acquire the DB connection here... */
  payConn = getConnection() ;
}

public void ejbCreate () throws CreateException { } ;
/* Finally! A business method implementation.
 */
public StockTradeResult buy(StockCustomer aCust,
                            String stockSymbol,
                            int shares)
      throws RemoteException {
 /* To buy stock, we merely update the record if this
customer has some stock or add a new record if not.
 */
 boolean success   = false ;
 String statusMsg = "Stock Purchase Failed" ;
 int numShares = getNumShares( aCust.getCustID(), stockSymbol ) ;
 if ( numShares > 0 )
    success = updateCustomerRecord( aCust.getCustID(),
                        stockSymbol,
                        numShares + shares ) ;
 else
    success = addCustomerRecord( aCust.getCustID(),
                     stockSymbol,
                     numShares ) ;

 if ( success )
    statusMsg = "Stock Purchase Successful" ;
 /*
     Let the client know the status of the trade.
```

```
     */
     return new StockTradeResult(statusMsg,
                                 aCust.getCustName(),
                                 shares,
                                 stockSymbol);
}

/*
   The other business method…
 */
public StockTradeResult sell(StockCustomer aCust,
                             String  stockSymbol,
                             int     shares)
       throws RemoteException {
  /* To sell stock, we check if the customer has enough stock
     to cover the sell order or update the stock on hand after
     a successful sale.
   */
   boolean success   = false ;
   String  statusMsg = "Stock Sale Failed" ;
   int numShares = getNumShares( aCust.getCustID(),
                                 stockSymbol ) ;
   if ( numShares >= shares ) {
       success = updateCustomerRecord( aCust.getCustID(),
                            stockSymbol,
                            numShares - shares ) ;
      statusMsg = "Stock Sale Successful." ;
   }
   else {
     statusMsg = statusMsg + "Not enough shares on account" ;
     shares     = 0 ;
   }

   return new StockTradeResult(statusMsg,
                               aCust.getCustName(),
                               shares,
                               stockSymbol);
}
/*This method adds customer share table returning success/
failure flag. When a customer buys stock and has none
on hand, this method adds a record for the
customerID/sharesymbol combination.
*/
private boolean addCustomerRecord( int    aCustomerID,
                                   String  stockSymbol,
                                   int     numShares) {
     try {
       payPS = payConn.prepareStatement(" insert " +
                       " into stocktable " +
                       "( customerid, sharesymbol, " +
                       "  numSharesOnAcct) " +
                       "  values (?,?,?) " ) ;
       /* Well, let's set the parametters.... */
       payPS.setInt   ( 1, aCustomerID ) ;
       payPS.setString( 2, stockSymbol ) ;
       payPS.setInt   ( 3, numShares ) ;
       /* Issue the SQL...
        Table primary key is compound key CustID and ShareSymbol
        1 row impacted or error. */
       int numRowsAdded = payPS.executeUpdate() ;
       /* Let's get the result. */
```

```java
         if ( numRowsAdded != 1 )
             throw new EJBException ("Table Addition Failed!!!" ) ;

      }
     catch ( SQLException sqlEx ) {
           throw new EJBException ( sqlEx ) ;
      }
     finally {
       try {
               if ( payPS != null )   payPS.close() ;
               /* We shut down the connection here..no more SQL
                   today!! */
               if ( payConn != null ) payConn.close() ;
           }
       catch ( SQLException sqlEx ) {
               sqlEx.printStackTrace() ;
           }
       }
     /* If we get here, we're successful */
     return true ;
}
/*This method updates customer share table returning success
  failure flag. This method gets called for buys and sells. */
  private boolean updateCustomerRecord( int     aCustomerID,
                                        String  stockSymbol,
                                        int     numShares) {
      try {
        payPS = payConn.prepareStatement(" Update stocktable " +
                                 " Set numSharesOnAcct = ?   " +
                                 " where customerid = ?    " +
                                 " and sharesymbol = ? " ) ;
        /* Well, let's set the parametters.... */
        payPS.setInt   ( 1, numShares ) ;
        payPS.setInt   ( 2, aCustomerID ) ;
        payPS.setString( 3, stockSymbol ) ;
        /* Issue the SQL...
         Table primary key is compound key CustID and ShareSymbol
         1 row impacted or error. */
        int numRowsUpdated = payPS.executeUpdate() ;
        /* Let's get the result. */
        if ( numRowsUpdated != 1 )
            throw new EJBException ("Table Update Failed!!!" ) ;

      }
     catch ( SQLException sqlEx ) {
             throw new EJBException ( sqlEx ) ;
      }
      finally {
        try {
               if ( payPS != null )   payPS.close() ;
               /* We shut down the connection here..no more SQL
                   today!! */
               if ( payConn != null ) payConn.close() ;
               }
               catch ( SQLException sqlEx ) {
                  sqlEx.printStackTrace() ;
               }

      }
   /* If we get here, we're successful */
   return true ;
```

```java
}
/*This method returns the number of stock the customer has.
Both buy and sell invokes this method — if the customer
puts in a sell order, the method checks if the customer has
the requisite stock on hand. For buy orders, we use the
number of existing shares added to the new purchase to
update the table. */
private int getNumShares( int     aCustomerID,
                          String  stockSymbol ) {
  int         numShares = 0 ;
  ResultSet numSharesOnTable = null ;
  try {
     payPS = payConn.prepareStatement(
                              " Select numSharesOnAcct " +
                              " from stocktable "        +
                              " where customerid = ?    " +
                              " and sharesymbol = ? " ) ;
     /* Well, let's set the parametters.... */
     payPS.setInt   ( 1, aCustomerID ) ;
     payPS.setString( 2, stockSymbol ) ;
     /* Issue the SQL...
        Table primary key is compound key CustID and ShareSymbol
        Either 0 or 1 row returned. */
     numSharesOnTable = payPS.executeQuery() ;
     /* Let's get the result. */
     while ( numSharesOnTable.next() )
         numShares = numSharesOnTable.getInt( 1 ) ;

  }
  catch ( SQLException sqlEx ) {
         throw new EJBException ( sqlEx ) ;
  }
  finally {
       try {
            if ( payPS != null )
                 payPS.close() ;
            if ( numSharesOnTable != null )
                 numSharesOnTable.close() ;
          }
       catch ( SQLException sqlEx ) {
            sqlEx.printStackTrace() ;
       }

       }
  /* If no shares on record, zero is returned, right? */
  return numShares ;
}
//Utility method to acquire a database connection
private Connection getConnection() {
   // The string below is coded in the Deployment Descriptor
   String jndiDBName = "java:comp/env/jdbc/paymentDB" ;
   try {
      InitialContext jndiCtx = new InitialContext() ;
      DataSource payDS =
              (DataSource) jndiCtx.lookup( jndiDBName ) ;
      return payDS.getConnection() ;

   }
   catch (NamingException ne) {
        throw new EJBException( ne ) ;
   }
```

```
        catch (SQLException sqlEx) {
            throw new EJBException( sqlEx ) ;
        }
    }
}
```

Notice that every business method requires information identifying the particular client. Also, the "business" of the bean can be completed in a single method invocation from the client.

You acquire a connection to the database by using an object of class `DataSource`, which is found in the J2EE version of JDBC that works with other J2EE APIs such as JNDI. By following the code examples in this chapter, you can learn more about JNDI.

Coding the Deployment Descriptor

Listing 14-7 shows the deployment descriptor for the `StockTrader` bean.

Listing 14-7: Code for the 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>exampleStatelessSessionBean</ejb-name>
      <home>chapter14.stocktrader.StockTraderHome</home>
      <remote>chapter14.stocktrader.StockTrader</remote>
      <ejb-class>chapter14.stocktrader.StockTraderBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <resource-ref>
        <description>
         Customer shares datasource reference
        </description>
        <res-ref-name>jdbc/custstockDB</res-ref-name>
        <res-type>javax.sql.DataSource </res-type>
        <res-auth>Container</res-auth>
      </resource-ref>

    </session>
  </enterprise-beans>

    <assembly-descriptor>
      <container-transaction>
     <method>
        <ejb-name>exampleStatelessSessionBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
      </container-transaction>
    </assembly-descriptor>
```

```
      </ejb-jar>
```

For the most part, your EJB deployment tool generates deployment descriptors for you. But, a few words on the content of this deployment descriptor are in order.

- The root element of a deployment descriptor is `<ejb-jar>`.

- You may code information pertaining to more than one EJB in a single deployment descriptor.

- The deployment descriptor contains sections that describe session and entity beans.

- The deployment descriptor contains information that JNDI needs to locate resources. Notice the reference to the database used in the above bean in the `<resource-ref>` tag.

- The `<assembly-descriptor>` tag may contain transaction and security-related information. The topic of transactions or security hasn't been discussed yet, so you haven't missed anything.

Code for a Sample Client

 shows some code for a sample client. The meat of the client is the invocation of the `buy` and `sell` methods defined in the `remote` interface.

Listing 14-8: Code for a sample client

```java
import chapter14.stocktrader;

import java.rmi.RemoteException;
import java.util.Properties;

import javax.ejb.CreateException;
import javax.ejb.RemoveException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import java.util.* ;

public class StockTraderClient {

  private static final String JNDI_NAME = "StockTraderHome";

  private StockTraderHome home;

  private StockCustomer   me = new StockCustomer( "Lou Marco",
                                                  12345,
                                                  "123ABC" ) ;
  //Represents the result returned from the buy or sell order
  private StockTradeResult resultLastTrans = null ;

  public StockTraderClient()
    throws NamingException
  {
    home = lookupHome();
  }

  /*
   * Runs this example from the command line.
   */
```

```java
public static void main(String[] args) throws Exception {

   StockTraderClient client = null;
   try {
      client = new StockTraderClient( );
   } catch (NamingException ne) {
      System.exit(1);
   }

}

/**
 * Runs this example.
 */
public void example()
   throws CreateException, RemoteException, RemoveException
{

   //Create a StockTrader
   StockTrader trader = (StockTrader)
             PortableRemoteObject.narrow(home.create(),
             StockTrader.class);

   String [] stocks = {"LMAA", "IOU2", "DOAP", "RIPP" };

      // execute some buys
   for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      System.out.println("Buying " + shares + " shares of" +
                          stocks[i] + "." );
      resultLastTrans = trader.buy(me, stocks[i], shares);
      //List out result of this trade
      showTradeResult( resultLastTrans ) ;
   }

   // execute some sells
   for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      System.out.println("Selling " + shares + " shares of" +
                          stocks[i]+".");
      trader.sell(me, stocks[i], shares);
      //List out result of this trade
      showTradeResult( resultLastTrans ) ;
   }
   //We're done…remove the instance of our trader bean.
    trader.remove();

}
private void showTradeResult( StockTradeResult str ) {
      Date now = new Date() ;
      System.out.println("On " + now + str.getStatusMsg() ) ;
      System.out.print("You traded " + str.getNumberTraded() ) ;
      System.out.println(" shares of " + str.getStockSymbol() +
                         "\n" ) ;
}

/*
 * Lookup the EJB home in the JNDI tree
 */
private StockTraderHome lookupHome()
   throws NamingException
{
```

```
     /* Lookup the beans home using JNDI.
        RMI/IIOP clients should use
        PortableRemoteObject.narrow function */
     Context ctx = new InitialContext();

     try {
       Object home = ctx.lookup(JNDI_NAME);
       return (StockTraderHome)
              PortableRemoteObject.narrow(home,
              StockTraderHome.class);
     } catch (NamingException ne) {
         System.out.println("Cannot locate Home object" ) ;
       throw ne;
     }
   }

}
```

The client acquires a reference to the `home` interface and then invokes business methods defined in the bean's `remote` interface.

Using JNDI

The Java Naming and Directory Interface, or JNDI, is pervasive in creating and using EJBs. Resources are stored in a *JNDI tree,* which resembles a directory tree. JNDI enables Java code to locate needed resources by using a structure called the *JNDI Context.* The context contains a JNDI method called `lookup` used, not surprisingly, to locate objects in the JNDI tree.

The process is to first acquire an `InitialContext` using a constructor. Then, you invoke the `lookup` method of class `InitialContext` to locate a resource on the JNDI tree.

The JNDI `lookup` method takes a name registered with JNDI through some other Java facility. For EJBs, JNDI is used in both the client and the enterprise bean.

The client uses JNDI to locate a reference to the `home` interface as shown in the `lookup` method. The name for the `home` method used in the client must match the name coded in the deployment descriptor. Fortunately, EJB servers and containers have tools that enable you to use a GUI to enter deployment descriptor information.

To maintain compatibility with CORBA, Java clients use the `PortableRemoteObject.narrow()` method. This method is not a JNDI method. Think of using this method as a way of coding a CORBA-compliant cast. The `narrow` method takes two arguments: an object of class `Object` to be cast, and an object of class `Class`, representing the class to cast the object to. The returned value of the `narrow` method is cast, Java-style, to the desired class. In the case of using the `narrow` method on `home` interfaces, the cast is to the class of the `home` interface of the bean.

The bean uses JNDI to locate a reference to the database. The process is the same as that used in the client program: acquire an `InitialContext` object followed by invoking the `lookup` method.

If one bean needs to call another bean, the calling bean may need access to the called bean's home object. In this scenario, the calling bean acts as a client. The calling bean could use JNDI to locate the `home` interface of the called bean.

Creating a Stateful Session Bean

The code presented in this section implements the same bean as shown in the preceding section but as a stateful session bean. The main difference is that the client passes a customer ID as an argument to the `create` method, which enables the stateful bean to maintain state.

Coding the remote Interface

The difference in the remote interface shown in Listing 14-9 for the stateful bean is the absence of a customer object passed to business methods. When the stateful bean is created, the container remembers the client who invoked the `create` method. The client does not have to pass a client identifier to the bean.

Listing 14-9: Code for the remote interface

```
package chapter14.stocktraderstateful;

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

/*
  Remote interface for the Stock Trader. These methods
  get implemented in the bean class StockTraderBean
 */
public interface StockTraderStateful extends EJBObject {

  public  StockTradeResult buy (String    stockSymbol,
                                int        numShares)
    throws RemoteException;

  public StockTradeResult sell (String    stockSymbol,
                                int        numShares)
    throws RemoteException;
}
```

Coding the home Interface

Listing 14-10 contains the code for the `home` interface. The difference between this and the stateless bean version is that here, you pass a client identifier as an argument to the `create` method.

Listing 14-10: Code for the home interface

```
package chapter14.stocktraderstateful;

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

public interface StockTraderStatefulHome extends EJBHome {

    StockTrader create(int custID ) throws CreateException, RemoteException;
}
```

Coding the Bean Class

Listing 14-11 shows the code for the stateful bean version. The bean class is very similar to the stateless version, but the stateful version uses a `customerID` to maintain client identity. This version does not access objects of class `Customer`, and the business methods — `buyStateful` and `sellStateful` — do not accept an argument that identifies the client. The relational database access routines are identical to those in the stateless bean version.

Listing 14-11: Code for the stateful bean

```
package chapter14.stocktraderstateful;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.EJBException ;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.io.* ;
import java.sql.* ;
import java.rmi.RemoteException ;
import javax.sql.DataSource ;
import java.util.* ;

public class StockTraderBeanStateful implements SessionBean {

    /** Declare the connection and preparedStatement object so we
        can use them in multiple methods. Also, we want to
        reuse the connection because we'll issue two SQL statements
        here.
        For a buy order, we need to determine if the customer has
        this stock already. If so, we issue an UPDATE sql. If not,
        we issue an INSERT SQL.
        For a sell order, we need to see how many shares the customer
        has to sell. If not enough, we issue a StockTraderException.
        If enough, we issue an UPDATE to change the shares on
        account.
    */
    Connection        payConn = null ;
    PreparedStatement payPS   = null ;
    int               customerID ;

    private SessionContext sCtx;
    /**

     */
```

```java
public void ejbActivate() { } ;
public void ejbRemove() { } ;
public void ejbPassivate() { } ;
/*
 */
public void setSessionContext(SessionContext ctx) {
  sCtx = ctx;
  /* Let's acquire the DB connection here... */
  payConn = getConnection() ;
}
/*
  Here, we pass a customer ID which will define
  the conversational state of the bean.
 */
public void ejbCreate (int custID ) throws CreateException {
    customerID   = custID ;
} ;
/*
  Note the absence of a client identifier passed as an argument
  The bean instance variable customerID serves to identify the
  Client.
 */
public StockTradeResult buyStateful(String stockSymbol,
                                    int shares)
       throws RemoteException {
 /* To buy stock, we merely update the record if this customer has
    some stock or add a new record if not.
 */
 boolean success   = false ;
 String  statusMsg = "Stock Purchase Failed" ;
 int     numShares = getNumShares( customerID, stockSymbol ) ;
 if ( numShares > 0 )
    success = updateCustomerRecord( customerID,
                         stockSymbol,
                         numShares + shares ) ;
 else
    success = addCustomerRecord( customerID,
                    stockSymbol,
                    numShares ) ;

 if ( success )
    statusMsg = "Stock Purchase Successful" ;

  return new StockTradeResult(statusMsg,
                              "Customer With ID " + customerID,
                              shares,
                              stockSymbol);
}

/**

 */
public StockTradeResult sellStateful( String   stockSymbol,
                                      int      shares)
        throws RemoteException {
  /* To buy stock, we merely update the record
     if this customer has
     some stock or add a new record if not.
   */
   boolean success   = false ;
  String  statusMsg = "Stock Sale Failed" ;
```

```java
        int numShares = getNumShares( customerID, stockSymbol ) ;
        if ( numShares >= shares ) {
            success = updateCustomerRecord( customerID,
                                            stockSymbol,
                                            numShares - shares ) ;
            statusMsg = "Stock Sale Successful." ;
        }
        else {
          statusMsg = statusMsg + "Not enough shares on account" ;
          shares     = 0 ;
        }

        return new StockTradeResult(statusMsg,
                                    "Customer With ID " + customerID,
                                    shares,
                                    stockSymbol);
  }
  //This method adds customer share table returning success/
  //failure flag
  private boolean addCustomerRecord( int     aCustomerID,
                                     String  stockSymbol,
                                     int     numShares) {
      try {
        payPS = payConn.prepareStatement("insert into stocktable" +
                              "( customerid, sharesymbol, " +
                              "  numSharesOnAcct) "          +
                              "  values (?,?,?) " ) ;
        /* Well, let's set the parametters.... */
        payPS.setInt   ( 1, aCustomerID ) ;
        payPS.setString( 2, stockSymbol ) ;
        payPS.setInt   ( 3, numShares ) ;
        /* Issue the SQL...
             Table primary key is compound key CustID and
             ShareSymbol 1 row impacted or error. */
        int numRowsAdded = payPS.executeUpdate() ;
        /* Let's get the result. */
        if ( numRowsAdded != 1 )
            throw new EJBException ("Table Addition Failed!!!" ) ;

    }
    catch ( SQLException sqlEx ) {
          throw new EJBException ( sqlEx ) ;
    }
    finally {
      try {
              if ( payPS != null )   payPS.close() ;
              /* We shut down the connection here..no more SQL
                 today!! */
              if ( payConn != null ) payConn.close() ;
          }
      catch ( SQLException sqlEx ) {
              sqlEx.printStackTrace() ;
          }
      }
     /* If we get here, we're successful */
    return true ;
  }
  //This method updates customer share table returning success/
  //failure flag
   private boolean updateCustomerRecord( int     aCustomerID,
                                         String  stockSymbol,
```

```java
                                            int    numShares) {
    try {
      payPS = payConn.prepareStatement("Update stocktable " +
                              " Set numSharesOnAcct = ?  " +
                              " where customerid = ?    "   +
                              " and sharesymbol = ? " ) ;
      /* Well, let's set the parametters.... */
      payPS.setInt   ( 1, numShares ) ;
      payPS.setInt   ( 2, aCustomerID ) ;
      payPS.setString( 3, stockSymbol ) ;
      /* Issue the SQL...
          Table primary key is compound key
          CustID and ShareSymbol
          1 row impacted or error. */
      int numRowsUpdated = payPS.executeUpdate() ;
      /* Let's get the result. */
      if ( numRowsUpdated != 1 )
          throw new EJBException ("Table Update Failed!!!" ) ;

    }
    catch ( SQLException sqlEx ) {
          throw new EJBException ( sqlEx ) ;
    }
    finally {
      try {
            if ( payPS != null )   payPS.close() ;
            /* We shut down the connection here..no more SQL
               today!! */
            if ( payConn != null ) payConn.close() ;
            }
            catch ( SQLException sqlEx ) {
               sqlEx.printStackTrace() ;
            }

    }
  /* If we get here, we're successful */
  return true ;
}
//This method returns the number of stock the customer has
private int getNumShares( int     aCustomerID,
                          String  stockSymbol ) {
  int        numShares = 0 ;
  ResultSet numSharesOnTable = null ;
  try {
     payPS = payConn.prepareStatement(" Select numSharesOnAcct " +
                                " from stocktable "        +
                                " where customerid = ?   " +
                                " and sharesymbol = ? " ) ;
     /* Well, let's set the parametters.... */
     payPS.setInt   ( 1, aCustomerID ) ;
     payPS.setString( 2, stockSymbol ) ;
     /* Issue the SQL...
         Table primary key is compound key CustID and ShareSymbol
         Either 0 or 1 row returned. */
     numSharesOnTable = payPS.executeQuery() ;
     /* Let's get the result. */
     while ( numSharesOnTable.next() )
         numShares = numSharesOnTable.getInt( 1 ) ;

  }
  catch ( SQLException sqlEx ) {
```

```
            throw new EJBException ( sqlEx ) ;
        }
      finally {
        try {
                if ( payPS != null )  payPS.close() ;
                if ( numSharesOnTable != null )
                                numSharesOnTable.close() ;
                }
                catch ( SQLException sqlEx ) {
                    sqlEx.printStackTrace() ;
                }

            }
        /* If no shares on record, zero is returned, right? */
        return numShares ;
    }
    //Utility method to acquire a database connection
    private Connection getConnection() {
        String jndiDBName = "java:comp/env/jdbc/paymentDB" ;
         try {
                InitialContext jndiCtx = new InitialContext() ;
                DataSource payDS = (DataSource)
                                jndiCtx.lookup( jndiDBName ) ;
                return payDS.getConnection() ;
          }
           catch (NamingException ne) {
                throw new EJBException( ne ) ;
           }
           catch (SQLException sqlEx) {
               throw new EJBException( sqlEx ) ;
           }

      }

}
```

Coding the Deployment Descriptor

The deployment descriptor for the stateful version is the same except for the `<session-type>Stateless</session-type>` tag, which, of course, would be coded as `<session-type>Stateful</session-type>`.

Code for a Sample Client

The client code uses a customer ID as an argument to create and remove the object of class Customer as the first argument to the `buyStateful` and `sellStateful` methods. Listing 14-12 shows the code.

Listing 14-12: Client implementation using the stateful bean

```
import chapter14.stocktraderstateful;

import java.rmi.RemoteException;
import java.util.Properties;

import javax.ejb.CreateException;
import javax.ejb.RemoveException;
```

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

public class StockTraderClientStateful {

  private static final String JNDI_NAME = "StockTraderHome";

  private StockTraderHome home;
  //This gets passed to the bean to identify the client
  private StockCustomerID   me =  12345;
  //Represents the returned from the buy or sell order
  private StockTradeResult resultLastTrans = null ;

  public StockTraderClient()
    throws NamingException
  {
    home = lookupHome();
  }

  /**
   * Runs this example from the command line.
   */
  public static void main(String[] args) throws Exception {

    StockTraderClient client = null;
    try {
      client = new StockTraderClient( );
    } catch (NamingException ne) {
      System.exit(1);
    }

  }

  /**
   * Runs this example.
   */
  public void example()
    throws CreateException, RemoteException, RemoveException
  {

    //Create a StockTraderStateful object. Note that the
    //customer ID is passed to the home.create()
    //method as the first argument to
    // PortableRemoteObject.narrow()
    StockTraderStateful trader = (StockTraderStateful)
              PortableRemoteObject.narrow(home.create( me ),
              StockTraderStateful.class);

    String [] stocks = {"LMAA", "IOU2", "DOAP", "RIPP" };

    // execute some buys. Note the absence of a client identifier
    //in the call to the business methods
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      System.out.println("Buying "+shares+" shares of" +
                          stocks[i]+".");
      resultLastTrans = trader.buyStateful( stocks[i], shares);
      //List out result of this trade
      showTradeResult( resultLastTrans ) ;
```

```
    }

    // execute some sells
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      System.out.println("Selling "+shares+" shares of" +
          stocks[i]+".");
      trader.sellStateful( stocks[i], shares);
      //List out result of this trade
      showTradeResult( resultLastTrans ) ;
    }

  }
  private void showTradeResult( StockTradeResult str ) {
      Date now = new Date() ;
      System.out.println("On " + now + str.getStatusMsg() ) ;
      System.out.print("You traded " + str.getNumberTraded() ) ;
      System.out.println(" shares of " + str.getStockSymbol() +
          "\n" ) ;
  }

  /**
   * Lookup the EJBs home in the JNDI tree
   */
  private StockTraderStatefulHome lookupHome()
    throws NamingException
  {
    /* Lookup the beans home using JNDI.
       RMI/IIOP clients should use PortableRemoteObject.narrow
       function */
    Context ctx = new InitialContext();

    try {
      Object home = ctx.lookup(JNDI_NAME);
      return (StockTraderStatefulHome)
             PortableRemoteObject.narrow(home,
                      StockTraderStatefulHome.class);
    } catch (NamingException ne) {
        System.out.println("Cannot locate Home object" ) ;
      throw ne;
    }
  }
}
```

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

Error handling and debugging JSPs have much in common with the same tasks in regular Java development.
However, the nature of JSPs can make this task much more difficult. The techniques and tools that you have learned
about in this chapter, especially the use of the `errorPage` and `isErrorPage` directives, should help you overcome
many of the problems that you will face when handling errors and debugging your JSP pages.

JSP Debugging Techniques

A syntax error in a JSP causes a compile error, with references to code in the servlet, not the JSP. Because JSP errors rarely reference the true cause, debugging JSPs can be quite different from other debugging you may be used to. This section provides some often-used techniques to debug JSPs.

Remembering Good Coding Practices

To get a job done within the deadline, even the most meticulous programmer may not follow all good coding practices. However, diligence following good practices will save you time as you develop JSP pages.

Comment, Comment, Comment!

If JSP authors take the time and trouble to comment their code, they may not make the coding mistakes that cause many runtime errors. The act of commenting the code often makes you stop and think about what you are coding. By expressing the intent of the code in human language, it's possible to uncover flaws in our reasoning — flaws that can result in runtime errors.

The JSP author can write JSP comments (`<%-- … --%>`), write Java comments in scriptlets and declarations, or write HTML comments. Java and HTML comments get passed to the JSP translator and eventually end up in the generated servlet.

All of us have used comments to block out pieces of code in an attempt to isolate the code causing an error, but be mindful of commenting out code in JSP scriptlets. Sometimes, a scriptlet intermingles static text with Java code. Combining the Java scriptlet code and static text results in syntactically correct Java, with the static text represented as `out.print` statements in the generated servlet. Comment out the wrong piece of scriptlet code and you will generate all sorts of translation (compile) errors.

The following code is an example JSP construct composed of several scriptlets. It shows a piece of the JSP page `example1.jsp` (from Listing 9-1) modified to display square roots in excess of 50 in blue.

```
<tr>
    <td><%= number %></td>
    <td>
 <% if ( squareRootNumber > 50 ) { %>
    <font color="blue">
 <% } %>

    <%= squareRootNumber %>
 <% if ( squareRootNumber > 50 ) { %>
       </font>
 <% } %>
    </td>
    <td><%= cubeRootNumber %></td>
```

```
        </tr>
```

Notice the proliferation of JSP scriptlet identifiers required to generate even a simple change. Commenting out pieces of scriptlet code while debugging or coding is an invitation to meet a translation time error.

Although Java and HTML comments get passed to the JSP translator and end up in the generated servlet, you can perform some JSP processing within HTML comments. You can embed JSP expressions within HTML comments without affecting the layout of the finished HTML page. For example:

```
<!-- Value of lobound parameter =
                  <%=request.getParameter( "lobound" ) %> -->
```

You would not see this comment line in the displayed page, but if you viewed the resultant HTML source you would see:

```
<!-- Value of lobound parameter = 2000.25 -->
```

You can embed time information in HTML comments throughout your pages if you want to know the order of the JSP processing — if page A is processed before page B, for example. Once you have fixed your errors and your JSP page is put into a production environment, such comments should not be visible to the end user. There are two methods of hiding debug comments from the end user that I will discuss briefly. First, you can remove the code that created the comments from the JSP pages. This method, while surefire, forces you to add them to the pages at a future time when debugging. A second method that is more robust is to use a `session` variable to flag whether debugging comments should be output. This flag could be turned on or off through a password-protected page so that you could switch to and from debugging mode as necessary.

Does the Resultant HTML (or XML) Work?

Although your JSPs may generate the HTML (or XML) you desire, the result of the generation may be in error. In other words, you may be generating HTML that is incorrect. The JSP is generating the *incorrect* HTML correctly.

Some JSP authors create several HTML pages that model the desired result of the JSP execution, *without* using their JSPs. If you have a working end product — one or more HTML pages that you want your JSPs to produce — you may be in a better position to catch logic errors in your JSPs.

Reloading JSPs and Classes

As you debug your application and find errors, you will see these errors in your JSP pages or Java classes. As you make changes and test them, you need to be sure that the changes are reflected in the server. By default, your server may not recompile JSP pages or reload classes that have been changed without being restarted. If you think you've fixed an error but your changes don't appear to be reflected, try restarting the server. Also, examine your server documentation to see how you can set it up to reload classes that have been changed and recompile JSP pages that have been altered.

Using println and log Methods

Often used by practitioners and frowned upon by theorists, intermittent attribute and variable values dumped to output by using various println and log methods can be very illuminating.

Using `out.println` is quick, requires no real setup or changes to your environment, and is unlikely to introduce any additional errors in your JSPs. You see the results in your JSP page after execution. However, adding output to the final, rendered page by using `out.println` will likely alter the presentation. It is not entirely a bad thing that `out.println` alters the presentation because it makes it obvious for subsequent removal where you coded the `out.println` statements.

If you want most of the convenience of using `out.println` without the possibility of altering your presentation, you can direct debugging output to the server. The destination of output depends on your server and your server configuration. You'll likely need a console window open to catch the output.

In Tomcat, you get a console window by using the standard startup batch files. For Windows environments, remember to set the command window properties to allow for scrolling or else your output may scroll into nowhere. Some servers have a startup switch that opens a console window, which will receive output from `System.out.println`.

The `log` method of class `GenericServlet` writes to a server-dependent log file (in Tomcat, the log file is in `TOMCAT_HOME/docs/servlet.log`). All you do to invoke the method is include the invocation in a declaration or a scriptlet.

The following is an example of the `log` function in a JSP declaration:

```
<%!
   private double getBound( String bound ) {
       log("In method getBound") ;
       return Double.parseDouble( bound ) ;
   }
%>
```

Here's an example in a scriptlet:

```
<% log("End of Table Generation") ; %>
```

Because log files typically append output, open the log file *after* JSP execution and view the results. It's a bit tedious to open the log, scroll to the bottom, and close the log, but just think of using the `log` method as another wrench in your JSP toolbox.

If you decide to use logging or `System.out.println`, you can toggle this debugging information on or off using a `session` debug flag. This method, as mentioned in the [previous section](#) on comments, will allow you to quickly transfer from a testing to a production environment with minimal time and changes on your part.

Debugging Concurrency Issues

Developing code that works with concurrent users is tough enough when your application consists of components that use visible code. Developing working code with components from nonvisible components just makes this more difficult. Code that is not developed to be thread-safe usually won't be. In addition, problems arising from timing issues are very difficult, if not impossible, to re-create.

That said, there are a few tips that may prove useful in debugging problems that arise from concurrent access of your JSPs.

You cannot solve a coding problem unless you can re-create it. The marketplace offers several tools to stress-test Java servlets, which may be used to create several concurrent requests to JSP pages.

You can include a generous sprinkling of `println` calls that write information identifying the client (session) and the threads. You can use the following line of code to track the client session:

```
<% System.out.println( "Client Session ID = " + session.getId() ) ; %>
```

You may use the following line to list the currently executing thread:

```
<% System.out.println( "Thread Name = " +
                       Thread.currentThread.getName() ) ; %>
```

There's always the possibility that the activity of writing output via `System.out.println` can introduce additional timing problems or make the problems that caused the original error more difficult to reproduce.

You can always try to take advantage of a debugger. Since you already use some sort of IDE, and virtually all IDEs come with a debugger, you probably have a debugger at your fingertips. However, I'm talking about JSP and servlet debugging here, and not all Java IDEs have support for debugging JSPs. If you set breakpoints and step through the

code and timing is the root of the problem, you'll rarely find the error, because code-stepping freezes the program and allows you to look at a statement in isolation. The very nature of a multithreaded application is that not all statements can be viewed in isolation. In short, using a code-stepping debugger is a less-than-perfect solution to a complex problem. As JSP development becomes more popular, hopefully we will see more robust and useful JSP debugging tools.

Tracking JSP Errors

JSP processing involves several steps, using several different software tools. An error encountered in using any of these tools may be difficult to track down. In the following sections, I will discuss using the JSP page directive attributes `errorPage` and `isErrorPage` to handle errors. I also cover the differences between JSP translation errors, which are coding mistakes, and runtime errors, which may or may not be coding mistakes.

Recall that JSPs use the page directive to set properties of the JSP by way of assigning values to attributes. Two attributes of the page directive especially applicable to JSP debugging are `errorPage` and `isErrorPage`. Also, the JSP implicit object called `exception` is useful in handling JSP errors. Let's go ahead and look at the use and importance of `errorPage`, `isErrorPage`, and `exception` next.

The errorPage Page Directive Attribute

The `errorPage` attribute names a JSP page that handles exceptions not handled in the current page. In the following coding example, the `errorPage` attribute is a relative URL.

```
<%@ page errorPage="mydir/myErrorPage.jsp" %>
```

The virtue of using custom error pages is that your JSP code is not cluttered with handling errors. (Remember that one of the design goals of JSPs is to separate business logic from presentation details.) Code that handles errors, a business logic activity, should not be intermingled with code to handle the presentation.

The actual reporting of the error is not done within the page containing the page directive with the `errorPage` attribute set. Instead, the reporting is done in the JSP page named as the `errorPage`, or the value of the `errorPage` attribute. The page mentioned should itself use the `isErrorPage` attribute. Before showing an example of the `errorPage` attribute in action, let's first take a look at the `isErrorPage` attribute.

The isErrorPage Page Directive Attribute

The `isErrorPage` attribute identifies a JSP page to handle errors. The coding is straightforward:

```
<%@ page isErrorPage="true" %>
```

The `isErrorPage` attribute defaults to false.

The JSP `example1.jsp`, for example, generates a table of square and cube roots based on the request parameters `lobound` and `hibound`. The JSP page directs errors not caught in the page to an error page. Listing 9-1 shows the code for `example1.jsp`.

Listing 9-1: JSP example showing use of the errorPage attribute

```
<%-- Tell JSP to redirect uncaught exceptions to errorpageex1.jsp --%>
<%@ page errorPage="errorpageex1.jsp" %>

<html>
<head>
<title>errorPage and isErrorPage Demonstration</title>
</head>

<body bgcolor="#dddddd">

<%!
    private double getBound( String bound ) {
        return Double.parseDouble( bound ) ;
    }
%>

<%
  double loBound   = getBound( request.getParameter( "lobound" ) ) ;
  double hiBound   = getBound( request.getParameter( "hibound" ) ) ;
  double increment = (hiBound - loBound) / 5;

%>

<center>
<font size=+2>Table of Roots from <%= loBound %> to <%= hiBound %></font>
<table border=2>
<tr>
   <td>Number</td>
   <td>Square Root</td>
   <td>Cube Root</td>
</tr>
<%
  double number, squareRootNumber, cubeRootNumber;

  for (int numIDX = 0; numIDX < 6; numIDX++ ) {
    number            = loBound + increment * numIDX ;
    squareRootNumber = Math.sqrt( number ) ;
    cubeRootNumber    = Math.pow( number, 0.3333 ) ;
%>
  <tr>
     <td><%= number %></td>
     <td><%= squareRootNumber %></td>
     <td><%= cubeRootNumber %></td>
  </tr>
<%
  }
%>
</table>
</body>
</html>
```

The page directive at the beginning of <u>Listing 9-1</u> directs exceptions to the JSP page `errorpageex1.jsp`. Notice that the routine `getBound` does not catch any exceptions, notably the `NumberFormatException`. Any exceptions will be directed to the JSP page `errorpageex1.jsp`.

<u>Figure 9-2</u> shows what a faultless execution of the JSP page `example1.jsp` looks like.
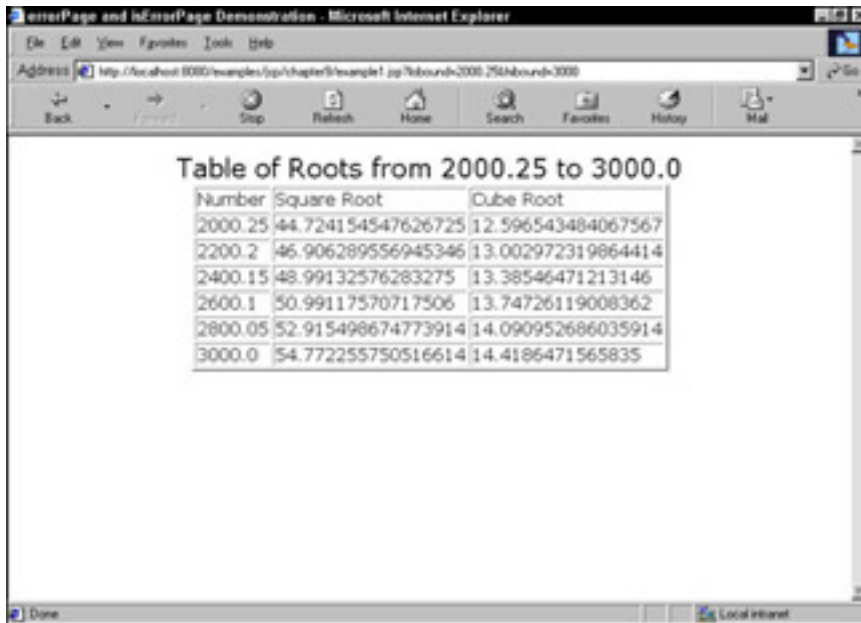
Figure 9-2: Display of example1.jsp without errors

Notice the presence of the request parameters `lobound` and `hibound` on the browser location bar.

Change the value of `hibound` from 3000 to 3000Z. The new value for `hibound` will generate a `NumberFormatException`. Because there is no code to catch the exception, you expect the processing to be directed to the error page. Figure 9-3 shows what you will see.
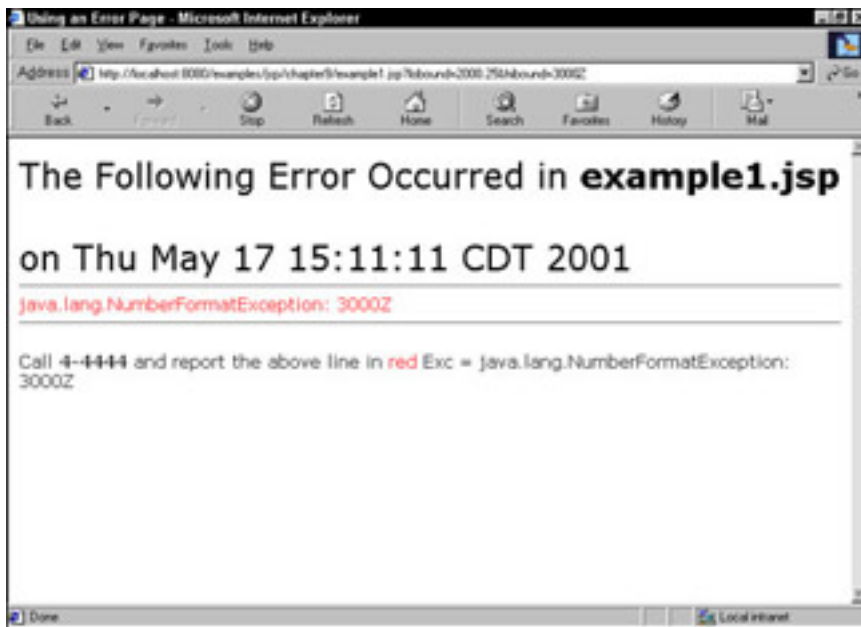
Figure 9-3: Display of example1.jsp with errors

Figure 9-3 is output of the JSP page `errorpageex1.jsp`. Recall that `errorpageex1.jsp` is the value of the attribute `errorPage` of the page directive of `example1.jsp`. Listing 9-2 provides the code for `errorpageex1.jsp`.

Listing 9-2: Code for error page errorpageex1.jsp

```
<%-- Tell JSP that this is an error page --%>
<%@ page isErrorPage="true" %>
```

```
<html>
<head>
<title>Using an Error Page</title>
</head>

<body bgcolor="#dddddd">

<P><font size=+3
The Following Error Occurred in <b>example1.jsp</b>
<br>on <%= new java.util.Date() %>
</font>
<br>
<hr>
<font color="red"><%=exception %></font>
<hr>
<br>
<p>Call <b>4-4444</b> and report the above line in <font color="red">red</font>.

</body>
</html>
```

The error page can contain a list of probable causes, a list of contacts, e-mail links, or any other useful information.

An error page identified by the `isErrorPage` attribute can handle errors from multiple pages. For generic error reporting, you might have a small set of core error pages with a few others to handle specific cases.

Caution   The server output buffer is flushed prior to display of an error page. If the `autoFlush` attribute of the page directive is set to true, the display of the error page may cause an error.

The exception Implicit Variable

You may wonder if you can catch exceptions yourself and forward error-processing to a JSP page using the `jsp:forward` action, instead of using the `errorPage` and `isErrorPage` directives. Well, you can. Any JSP page can be used to handle exceptions. However, only JSPs with the `isErrorPage` attribute set to true have access to the implicit variable `exception`. During the rest of this discussion, when we talk about JSP error pages, I will be referring to JSP pages with the `isErrorPage` attribute set to true.

Notice the following statement from Listing 9-2:

```
<font color="red"><%=exception %></font>
```

The expression `<%= exception %>` yields the following output (in this example):

```
java.lang.NumberFormatException: 3000Z
```

The `exception` variable represents the `Throwable` object that caused the JSP container to invoke the page named as the error page. This variable exists only for the duration, or life, of the error page. You can hold on to the variable by saving its value as an attribute as follows:

```
session.setAttribute("errorVarPageex11", exception ) ;
```

Table 9-1 shows some handy JSP expressions using the `exception` variable.

Table 9-1: JSP Expressions Using the exception Variable

Expression	Meaning
`<%=exception>`	Displays a description of the exception.
`<%=exception.getMessage() %>`	Displays the message associated with the exception.
`<%=exception.getlocalizedMessage() %>`	Displays a local version of the exception. If localization parameters are not set, `exception.getlocalizedMessage` returns the same output as `exception.getMessage`.
`<%=exception.printStackTrace() %>`	Displays a stack trace.

Remember that JSP error pages have access to the implicit variable `exception`. The implication is that non-error pages do not have access to this variable. The truth is you can reference the `exception` variable as an attribute of the implicit `request` object. You can think of the `exception` variable as a convenient reference for the value of the `request` attribute `javax.servlet.jsp.JspException`.

JSP Translation Errors

As you should know by now, JSP pages are translated into Java servlets. When you introduce syntax errors in your JSPs, you will generate compilation errors. Compilation errors *will not* be detected by error page redirection. Put differently, you need functioning, syntactically correct JSP pages to use error pages.

The default output of a JSP translation/compilation error is server-dependent. The output should strongly resemble what you would see if you compiled the generated servlet directly. Refer to Figure 9-1 to see an example of a screen displayed in response to a translation error for the Tomcat Web server; your server should display something similar.

Your Web server should also generate *log files* describing, among other things, translation and runtime activity. You should reference your server's documentation and locate these log files, as one of them may provide details on translation (compile-time) errors.

Recall that JSPs are loaded and translated when first referenced. Subsequent references do not cause a JSP page retranslation; the server kicks off another thread and loads the existing copy of the previously translated servlet. The JSP author can compile the JSP page and save the results of the compilation on the server. If the author compiles the JSP page ahead of time, the first user to access the JSP page does not have to wait for JSP page translation. In essence, the user is accessing a servlet.

Compiling your JSP pages ahead of time will not only save the end-user time when pages are first accessed, but the author is assured that the pages are syntactically correct. Many servers have tools that enable you to compile the JSP page; Tomcat has the *JspC* tool located in the bin directory.

When repairing translation errors, you should not make changes to the generated servlet directly. Even though you know what you are doing, the next person may not. More important, the next time the JSP is compiled, the servlet will be regenerated, causing any changes to be destroyed.

JSP Runtime Errors

Sadly, the execution of syntactically correct JSP pages can still generate runtime errors. Because JSP runtime errors are essentially Java runtime errors, you can use Java exception handling mechanisms to catch and handle errors. In particular, you may include `try/catch` blocks in Java scriptlet code and methods coded in Java declarations.

Within `try/catch` blocks, you cannot easily use the `jsp:forward` action to redirect activities to another JSP page. The following syntax is not correct:

```
<%
try {
  //Some Java code
}
catch (someExceptionClass sexobj) {
  <jsp:forward page="somePage.jsp">
}
%>
```

Instead, use the `sendRedirect` method of the class `java.servlet.http.HttpServletRequest` to redirect JSP processing within a `try/catch` block, as shown in the following code:

```
<%
try {
  //Some Java code
}
catch (someExceptionClass sexobj) {
  response.sendRedirect( "somePage.jsp" ) ;
}
%>
```

Of course, you can also use the JSP `errorPage` and `isErrorPage` attributes of the `page` directive, as shown earlier.

JSP Exception Classes

The two exception classes specific to JSP are `JspException` and `JspError`. You encountered these classes in Chapter 7, "JSP Tag Extensions." `JspException` is a subclass of `Exception`, and `JspError` is a subclass of `JspException` (not class `Error`). Both JSP exception classes may be found in the `javax.servlet.jsp` package.

Note   Unless you are writing custom tag libraries, you're not likely to use these exception classes.

Now that you have a feel for the nature of the errors you'll encounter as you author JSP pages, let's discuss how to track the errors down.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Examining a JSP Error

The laws of thermodynamics state that energy cannot be created or destroyed; it can only be transformed from one form into another. You can make an analogous statement about JSP creation. Yes, JSPs do lots of work for you. Think of all that effort — or energy — you can save by not having to code servlets to generate dynamic content. However, work not done by you is transferred to the JSP translator. The JSP translator performs the task of changing the JSP page into a servlet. While this process simplifies your development work, it can also, at times, make the source of an error in a JSP more difficult to find.

JSPs get translated into servlets. Often, your diagnostics will refer to activity in the generated servlet. For this reason alone, you should have a good handle on Java servlets to make advanced JSP development go more smoothly. Although JSPs are touted as a Web page designer's tool, a JSP author will have an extremely difficult time tracking down errors if he or she doesn't have a programming background or the help of somebody with one.

The following JSP syntax error generates a runtime error, which generates the error page shown in Figure 9-1.
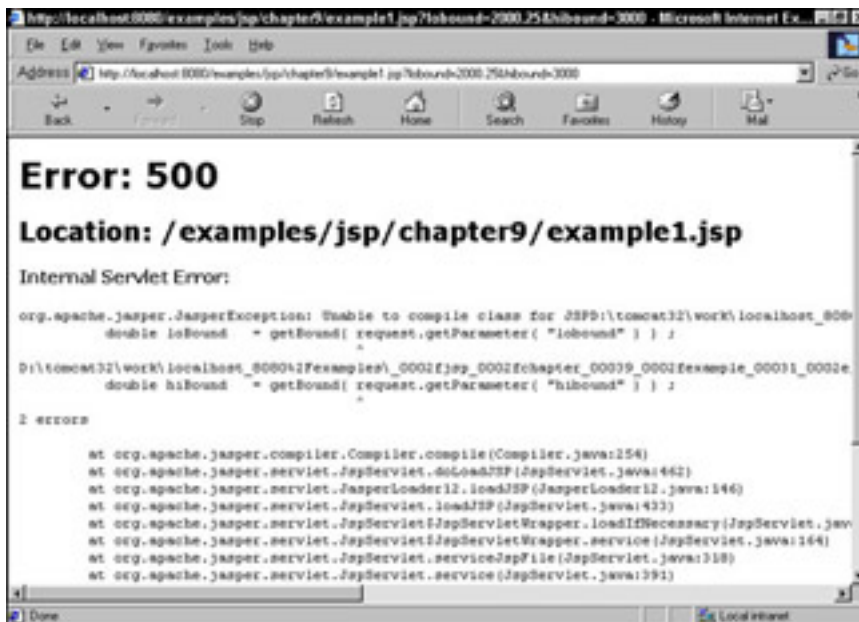
Figure 9-1: Typical JSP error screen

```
<%!
  double loBound   = getBound( request.getParameter( "lobound" ) ) ;

%>
```

The incorrectly coded JSP declaration should be coded as a JSP scriptlet. The correct syntax is as follows:

```
<%

  double loBound   = getBound( request.getParameter( "lobound" ) ) ;

%>
```

Notice that the line numbers and the file references in the stack trace in Figure 9-1 do not refer to the JSP page but to the contents of the generated servlet (although the file location points to the JSP page). In addition, nothing in the diagnostic directly discusses JSP declarations or JSP scriptlets.

Most JSP-enabled servers have an option that allows you to save the generated source code. Hunt that option down and switch it on now. You'll have a hard time finding and solving problems unless you can see the actual generated servlet source code.

Note   Tomcat uses a directory called "work" under TOMCAT_HOME where intermediate files, including compiled JSP servlets, are stored.

JSP pages usually generate HTML, possibly combined with Javascript, that is returned to the client's browser. Any page with more than one programming languages can be quite confusing to read. Finding errors in these situations can be challenging. Your JSP pages may be generating HTML or scripting code that contains errors. Given the interesting ways different browsers render HTML, you can have resulting pages that render properly on one browser but fail to render on a different browser.

While multiple clients may access the same JSP pages, concurrent access opens up a set of debugging issues. During debugging, you'll need ways of identifying different clients and different threads. I'll have more to say about debugging when there are multiple (concurrent) clients later in this chapter.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

In this chapter, you learned about using sessions in your JSP pages. Sessions are a convenient way for an application
to share data among multiple JSP pages during the same client session. However, they are resource intensive, so
remember to examine if you need to use them, depending on the needs of your application. In conclusion, sessions
are a powerful tool that overcome the stateless HTTP protocol, thus making our Web applications more robust and
user-friendly.

.

Summary of Session Methods

Table 5-2 is a list of methods you can use to manage and manipulate the session object. Most of the method names are self-explanatory. Most of these methods throw an `IllegalStateException` when invoked from an invalidated session object. Unless otherwise specified, the method throws the previously mentioned exception.

Table 5-2: Session Methods

Method	Description
`Object getAttribute( String)`	Fetches an object that was previously stored in the session object with `setAttribute`.
`void setAttribute(String, Object)`	Assigns a value to a session object. Returns null if the property does not exist in the session object. Also, you cannot store primitive types in session objects; you need to use a wrapper class to generate an object representation before saving to the session object. Any object listening for changes to the session object has its `valueBound` method executed.
`Enumeration getAttributeNames`	Returns the list of objects stored in the session object.
`long getCreationTime`	Returns the number of milliseconds since January 1st, 1970, since the session was created.
`String = getId`	Returns the session ID.
`long = getLastAccessedTime`	Returns the number of milliseconds since January 1st, 1970, since the session was last accessed.
`int getMaxInactiveInterval`	Returns the number of seconds of inactivity that must occur before the session times out. When this method returns a negative number, the session never timeouts.
`void setMaxInactiveInterval(int)`	Sets the number of seconds of inactivity before a session timeout.
`void invalidate`	Removes all objects stored in the session object and invalidates the session. Any object listening to changes in the session object has its `valueUnbound` method invoked.

`boolean = isNew`	Returns true when a session object is created and known to the server but not known to the client; false otherwise. If your application relies on cookies to track sessions and a client has disabled cookies, `isNew` always returns true.
`void removeAttribute( String )`	Removes the object referenced by the property name argument from the session object. All objects listening to changes in the session object are notified, and their respective `valueUnbound` methods are executed on the server. If the property named in the argument does not exist in the session object, nothing happens (not even the invocation of the `valueUnbound` methods).

You may see session methods `putValue`, `getValue`, and `getValueNames`. These three methods have been deprecated since the 2.2 release of the Java Servlet API.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

.

Understanding Sessions

Simply put, a *session* is a series of client requests and server responses such that the requests and responses are somewhat dependent on one another. Put differently, a session is a group of requests and responses coming from and going to the *same* client over a continuous period of time. A session is the difference between a simple exchange of information and an extended conversation. Sessions make possible complex Web transactions, such as electronic shopping cart implementations and interactive data analysis (fetch data, fine-tune selection based on data returned, and so on).

The JSP/servlet *implementation* of a session is an object called, appropriately enough, a *session object.* The session object is available to JSPs and servlets. The JSPs and servlets use the session object to store and retrieve data relevant to the ongoing session.

HTTP, the major Internet protocol, is *stateless.* HTTP sends a client request to a server. The server processes the client's request and uses HTTP to send a response back to the client. HTTP works at the single request/response level, meaning that each request is handled independently of any other. Your application consists of independently sent HTTP requests and responses to these requests.

Note   You may have heard about HTTP 1.1, which can use a *persistent connection.* The basic idea is to group requests and send them as a single transmission, thereby avoiding some of the overhead incurred by initiating several transmissions.

No one would fault you for thinking that a persistent connection somehow implies a session as discussed previously. However, the HTTP 1.1 persistent connection involves network transmissions and has nothing to do with the server maintaining a connection with the same client.

As previously mentioned, HTTP is a stateless protocol. You have to graft JSP and servlet features and capabilities atop HTTP to create, maintain, and track sessions. In short, these features are methods available to your JSPs by accessing the aforementioned session object.

JSPs use the implicit session object to track the requests of a single client and to distinguish one client from another. Consider an application used concurrently by multiple clients. Each client progresses through a series of steps to complete a transaction. A good, if not overused, example of such a multi-step transaction is an electronic shopping cart. A client must complete steps A, B, and C before proceeding to step D. Meanwhile, other clients are progressing through the same steps. By using sessions, the application is able to identify what clients are doing and what steps the clients are executing. Imagine the chaos if an application mixes up clients' shopping carts!

The Session Life Cycle

This section discusses how a session is created, how the client and server use the session to maintain state and communicate, and how a session dies.

Creating a Session

The client requests the display of a Web page from a server. The request causes the server to issue a call to the `getSession` method. The `getSession` method requests that a session be established for this client-server pair. The successful execution of `getSession` results in a returned object, the *session* object, mentioned in the previous section, as an instance of class `HTTPSession`.

Recall that JSPs use a group of implicit objects, one of which is the session object. The session object is implicit because your JSP code need not explicitly create the session object. Having immediate access to the session object through a well-defined interface is one advantage of using JSPs. Another way of saying that JSPs automatically have access to the session object is saying that JSPs automatically participate in sessions.

Note   Although JSP pages by default have access to the implicit session object, you can override the default by coding the following page directive:
```
<%@ page session="false" %>
```

The session object holds data relevant to the session. However, you still need a mechanism for identifying the client among possibly thousands of clients. The solution is for the server to generate a unique *session ID*. The server returns the session ID to the client by using a cookie (if the browser has cookies enabled) or by appending the session ID to a URL (called *URL rewriting)*. However, there is no session connecting the client to the server until the client returns the session ID *back* to the server by issuing a new request for some server resource.

Note   If the browser (client) has disabled cookies or the server does not support URL rewriting, no session ID is returned from the client to the server. Now, every JSP page that requests some server resource causes the server to generate a new session object. That means information does not pass between pages.

Your JSP pages or servlets never have to look at the session ID. The format of the session ID is dependent on the Web server. Actually, the session ID is the sole piece of information identifying the session that is presented to the client.

Client-Server Interaction During a Session

Every time the client requests some resource available from the server, the client passes the session ID as part of the request. The server uses the session ID to access the requested resource. Think of the server maintaining a collection of dictionaries or hash tables — one for each client — and the session ID is the entry into the dictionary/table for that client. Figure 5-1 depicts interaction between a client and a server during a session.
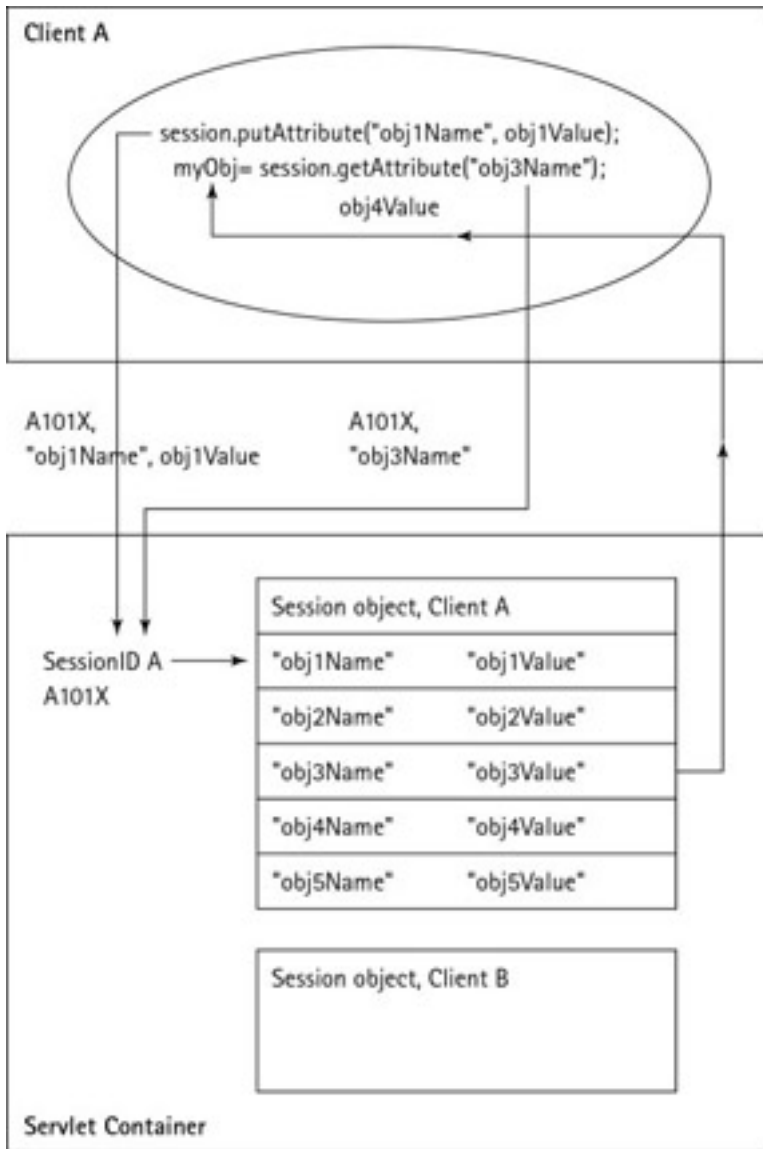
Figure 5-1: Client-server interaction during a session

The server has established a session with client A by issuing a `getSession` method invocation. (The call to `getSession` is not shown in Figure 5-1.)

Note   The session ID in our example has a value of A101X. The value used for the session ID is purely for illustrative purposes. For example, a session ID returned by the Tomcat server by the call to `getSession` in a JSP page by coding the expression `<%= request.getSession %>` is `org.apache.tomcat.session.StandardSession@6dfa45`.

The session ID for client A is different than the session ID for other clients. An important point is that the JSP page need not explicitly use the session ID when communicating with the server. You do not code the session ID as an argument to JSP or servlet methods.

Figure 5-1 depicts the session object as a table, with the left column containing names and the right column containing values. In JSP and servlet parlance, the left column contains the session object's *attributes* and the right column contains the *attribute values*.

The method invocation `putAttribute( attributeName, attributeValue)` places the name-value pair in the session object. Figure 5-1 shows the name-value pair together with the session ID being transmitted to the server. The server knows to access the session object for client A because of the session ID sent with the request.

In a JSP page, you can code the `putAttribute` method as a scriptlet:

```
<% session.setAttribute("nameLuLu", (Object)"LuLu"); %>
```

Figure 5-1 also shows a client (JSP page, for example) retrieving the value of a previously saved session attribute named *obj3Name*. Again, the session ID gets sent to the server along with the attribute name. The session ID for client A points the way to the session object stored for client A. The server reacts to the client request by accessing the session object and returns the value of the requested attribute.

In a JSP page, you can code the `getAttribute` method invocation in a scriptlet or expression:

```
<%= session.getAttribute("nameLuLu") %>
```

Your JSP pages and servlets get and set attribute values for the life of the session. The session object provides access to the attribute values and the session ID provides access to the session object.

Note   You may see calls to the methods `getValue` and `setValue` in some JSPs or servlets that purport to fetch or change session attribute values. The `getValue` and `setValue` methods are deprecated as of Java Servlet 2.2, replaced by `getAttribute` and `setAttribute`.

In short, your JSPs and servlets access and change session object data by using `setAttribute` and `getAttribute` methods.

Saving Session Data

Session data is saved for the duration of the client connection to the server. When a client logs off a site, the session data generated during the client visit would normally be destroyed once the client logged off the site. If you have a need to save client data, you should use a storage-based solution, such as using a database or using cookies, to save the data.

Using Cookies to Save Session Data

A *cookie* is a piece of data placed on a client by a server. Cookies were developed for the express purpose of providing a mechanism to save session data. Cookies can be temporary (for the life of the session) or semi-permanent (days or months).

Cookies have several limitations that make them less than perfect as a session tracking mechanism. Browsers are required to accept 20 cookies per site, 300 cookies per user and can limit the size of the cookie to 4K.

One severe limitation in using cookies to store session data is that the client has control over the storage of cookies on the client machine. If the client turns cookies off, the application that relies solely on cookies to save session data cannot do so. That's why applications that rely on sessions implement multiple session tracking mechanisms.

You can use cookies by invoking the methods in the class `javax.servlet.http.Cookie`. Table 5-1 lists some of the methods available to JSPs (and, therefore, to servlets) to use cookies.

Table 5-1: Methods Used to Manipulate Cookies

Method	Description
`Object clone`	Returns a copy of the cookie.
`String getComment`	Returns the comment set for this cookie with the `setComment` method. If no comment exists, `getComment` returns null.

`void setComment(String)`	Specifies a comment that describes the cookie's purpose.
`String getDomain`	Returns the URL set for this cookie with the `setDomain` method.
`void setDomain(String)`	Specifies the domain where this cookie should be presented.
`int getMaxAge`	Returns the maximum age for the cookie, possibly set by `setMaxAge`. The default value of –1 means the cookie lasts until the client closes the browser.
`void setMaxAge(int)`	Sets the maximum age for the cookie in seconds.
`String getName`	Returns the name of the cookie set with the cookie constructor.
`String getPath`	Returns the location of the cookie on the server set by the `setPath` method.
`void setPath( String )`	Specifies a path where the client should return the cookie.
`boolean getSecure`	Returns true when the browser sends the cookie over a secure protocol, false otherwise.
`void setSecure( boolean )`	Indicates whether the cookie should be sent using a secure (HTTPS or SSL) protocol.
`String getValue`	Returns the value of the cookie set with the `setValue` method.
`void setValue( String )`	Specifies a value for the cookie.
`int getVersion`	Returns the version of the protocol, 0 (original) or 1 (Netscape) set with the `setVersion` method.
`void setVersion( int )`	The version of the cookie protocol this cookie compiles with.

Listing 5-1 shows adding a cookie in a JSP and listing the names of all cookies.

Listing 5-1: Adding a cookie and listing all cookies in a JSP

```
<html>
<head>
<title>Cookies in JSP</title>
</head>

<body bgcolor="#dddddd">
<%
//Create a new cookie and add to the rest
//"Name", "Value"
Cookie ch5Cookie = new Cookie ("Ch5Cookie", "Chapter 5");
//This cookie will persist for 1 minute
ch5Cookie.setMaxAge( 60*2 ) ;
ch5Cookie.setComment("This is a test Cookie for Chapter 5" ) ;
response.addCookie(ch5Cookie);

//
//Loop over all cookies, printing out some cookie info
Cookie aCookie = null ;
Cookie[] allCookies = request.getCookies() ;
```

```
for (int cIDX = 0; cIDX < allCookies.length; cIDX++ ) {
    aCookie = allCookies[ cIDX ] ;
    out.print( "Cookie # " + cIDX + ": Name    = " + aCookie.getName()+ "<br>" ) ;
    out.print( "                          Value  = " + aCookie.getValue()+ "<br>" ) ;

}

%>
</body>
</html>
```

To add a cookie, you instantiate a cookie object and add attributes to the cookie followed by invoking the `addCookie` method from the `response` object.

Class `cookie` has no method to access a cookie by name. The technique is to grab all the cookies by invoking the `getCookies` method from the `request` object. Loop over the array of cookies and use cookie methods to extract the desired information.

As an aside, note the use of the implicit `out` object in the scriptlet to generate HTML output. Figure 5-2 shows the JSP page in Listing 5-1.
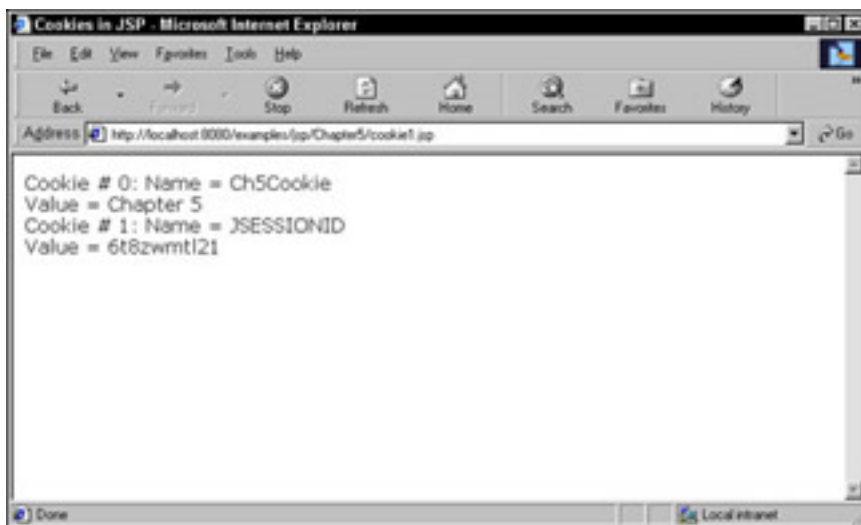
Figure 5-2: JSP page with cookie information

Terminating a Session

The session can be terminated by a client logging out (assuming the client had to log in), by a timeout (period of client inactivity), or by an explicit call by the client or server of the session method `invalidate`. The session object is tagged for garbage collection after the session terminates.

Should All Your JSPs Participate in Sessions?

You may not want to use the session object. Think about it — the session object allows a client and a server to have access to the same data, the data stored in the session object. If you do not need this requirement, and you do not need for the client and server to share some data, then you may not need to establish a session.

Later, you'll see that several JSPs can access the session object. If you don't want several JSP pages to have access to the same information, you probably do not need to establish a session.

Recall that JSP participation in sessions (JSP access to the implicit session object) is the default behavior. The pertinent question is whether or not you should, at times, squelch the default behavior of JSP session participation? In other words, is there a reason why sessions are *not good* for your Web application?

The answer is that sessions do, indeed, exact a performance penalty on your application. The main performance penalties include use of memory and the amount of time and server resources required to access and manipulate the session object. Remember that no server is an island; think of all those other clients out there hungry for server resources. If your application has no need of sessions, merely code the page directive with the session attribute set to false. If you try to access the session object in a page that isn't participating with sessions, you receive a fatal error from your JSP container.

Sharing Data with Sessions

As previously mentioned, one good use of sessions is to share data with a server and a client. The session object with the object's associated data, is known to all JSPs used by a client during a session. If JSP page A invokes JSP page B, JSP page B has access to data in the session object established by JSP page A because JSP pages A and B are of the same session.

Listing 5-1 shows a JSP page that has references to the session object and a link to another JSP page. The references to the session object are in italics.

Listing 5-2: JSP page A with references to the session object

```
<%-- Tell JSP that this page renders HTML --%>
<%@ page contentType="text/html" %>
<html>
<head>
<title>Show Session Data Across JSP Pages</title>
</head>

<body bgcolor="#dddddd">

Some Session Data: <p>

<p>Session = <b><%= request.getSession() %></b>
<p>Session ID = <b><%= session.getId() %> </b>
<P>Session Create Time = <b><%=session.getCreationTime()%></b>
<P>Session Last Access Time = <b>
<%= session.getLastAccessedTime() %> </b>
<P>Session Kept Open (alive) Without Inputs For <b>
<%= session.getMaxInactiveInterval()%> </b>Seconds
<P>Adding 100 to the time the session is Kept Open (alive) Without Inputs.
<% session.setMaxInactiveInterval(
          session.getMaxInactiveInterval() + 100 ); %>
<p>Set session attribute <b>nameLuLu </b>to string <b>"LuLu"</b>
<% session.setAttribute("nameLuLu", (Object)"LuLu"); %>
<br>
<a href=forch5b.jsp>Click Here for Next JSP Page</a>

</body>

</html>
```

Figure 5-3 is the page displayed from the JSP code in Listing 5-2.

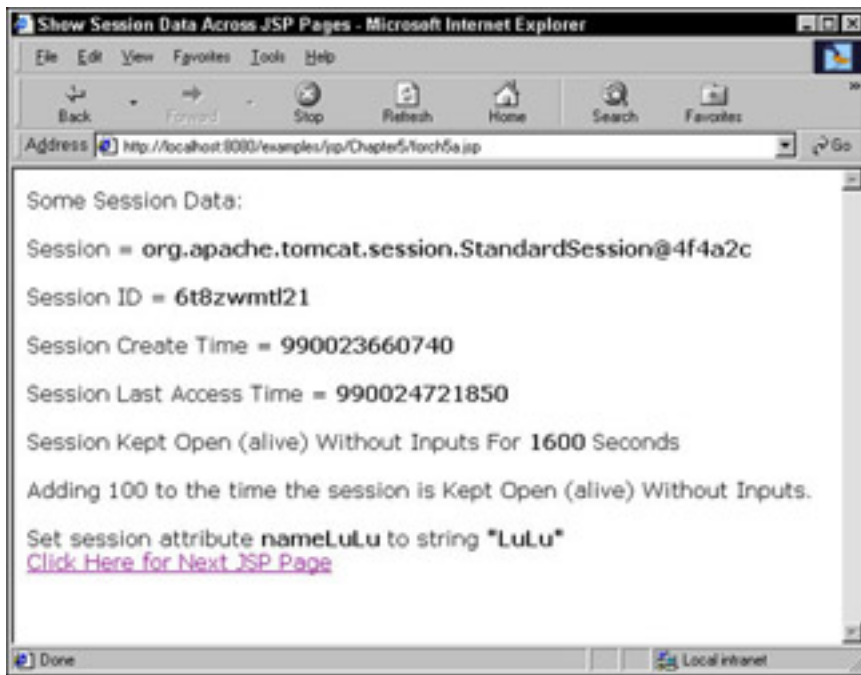**Figure 5-3:** The JSP page resulting from the code in Listing 5.1

Recall that JSP pages, by default, participate in sessions. If the JSP page that would be displayed when the user clicks on the link `Click Here for Next JSP Page` does not have a page directive that denies session participation (`<%@ page session="false" %>`), the JSP page has access to the session object.

When you have multiple JSP pages accessing the same objects within the session object, you may have to pay special heed to synchronization issues. JSP does not help you here — you must tend to such issues yourself. You also may have to tend to changes in the session object, which is covered in the next section.

Tending to Changes in the Session Object

Because the session object may be shared among all servlets and JSPs in a client's session, the objects in your application may need to know when objects are added, removed, or changed to the session object. Fortunately, the Java servlet API provides for an event-listener mechanism familiar to Java programmers such as you.

When a class implements the `javax.servlet.http.HttpSessionBindingListener` interface, the objects instantiated from the class are notified of changes to the session object. We say that an object is *bound* to a session when the object implements the interface. When the application issues a `setAttribute` or `removeAttribute` method call, the bound objects are notified and can take action. One common use is to allow objects to save their state when the user, or application, is about to terminate the session.

The `HttpSessionBindingListener` interface has two methods:

`public abstract void valueBound(HttpSessionBindingEvent hsbe)`

An object implements this method to be notified of additions to the session object by use of the `setAttribute` method.

`public abstract void valueUnbound(HttpSessionBindingEvent hsbe)`

An object implements this method to be notified of removals from the session object by use of the `removeAttribute` method.

Invoking ValueBound and ValueUnbound

Here's how `valueBound` and `valueUnbound` get invoked based on session object activity. First, you create an object from a class that implements the `HttpSessionBindingListener` interface:

```
public class ClassForMyObj implements
                          HttpSessionBindingListener

ClassForMyObj myObj = new ClassForMyObj() ;
```

Next, you bind the object to the session using `putAttribute`:

```
sessionObject.setAttribute( "AttribName", myObj ) ;
```

After execution of the `putAttribute` method, the server executes the implementation of `valueBound` for object `myObj`.

When the application takes some action that causes the removal of object `myObj`, either by session termination using `invalidate` or removing the attribute using `removeAttribute`, the server executes the implementation of `valueUnbound` for object `myObj`.

Client A

session.putAttribute("obj1Name", obj1Value);
myObj= session.getAttribute("obj3Name");
obj4Value

A101X,
"obj1Name", obj1Value

A101X,
"obj3Name"

SessionID A
A101X

Session object, Client A

"obj1Name"	"obj1Value"
"obj2Name"	"obj2Value"
"obj3Name"	"obj3Value"
"obj4Name"	"obj4Value"
"obj5Name"	"obj5Value"

Session object, Client B

Servlet Container

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Chapter 5: JSP Web Sessions

Overview

Suppose that you encounter a Web site that looks interesting and you want to explore it. Before you proceed, the site requests that you provide some information. You dutifully enter the requested information and click a link on the site, but then the site asks you to enter the same information you entered a few screens earlier. In disgust, you move on and find another interesting site. As with the previous site, the second site also requests that you enter information. You comply, click a link, and proceed to explore the site unfettered. Unlike the first site you visited, the second site "knows" who you are and recalls the previously entered data. The programmers of the second site have established a way for the site to remember who you are.

If you, the Web applications developer, want your application to remember previous visits by clients, you have to add something extra to your Web application called *Web sessions*.

This chapter describes how you implement Web sessions with the help of JSP features. You can read about Web sessions in general and JSP support for Web sessions in particular. This chapter also includes several JSP pages that implement JSP Web session support.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Summary

You've covered a lot of ground in this chapter! You've read about using static content in your JSP pages. You've also looked at the different types of JSP tags, including those that are programmable and those that are directives. We will be able to combine these elements to create working applications once we've covered Web sessions and how to use JavaBeans with your JSPs in the next two chapters.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

JSP Implicit Objects

You, the JSP programmer, have access to several predefined objects known as *implicit objects*. Some of these objects correspond to the value of the `scope` attribute for the `useBean` action described previously. Others represent the request and response objects for the generated servlet. Because the implicit objects *are* objects, they have scope like other Java objects. Table 4-4 lists the essential details of the JSP implicit objects.

Table 4-4: JSP Implicit Object Scope and Descriptions

Object Name	Scope	Description
application	Application	Represents the servlet context, or where the JSP is executing. The application object is an instance of class `javax.servlet.ServletContext`.
config	Page	Represents the servlet configuration. Object methods enable you to store and retrieve initialization data. These objects are instances of the class `javax.servlet.ServletConfig`.
exception	Page	Represents a `Throwable` object. Only JSP pages that set the `isErrorPage` attribute to true define this implicit object.
out	Page	Represents the output sent back to the client browser. The `out` object is an instance of `javax.servlet.jsp.JspWriter`.
page	Page	Represents the current JSP page. The page object is an instance of class `Object` and can be referenced by the Java keyword `this`.
pageContext	Page	Represents the current JSP page context. This object is an instance of class `javax.servlet.jsp.PageContext`.
request	Request	Represents the request coming from the client for processing by the JSP page. The request object is an instance of class `javax.servlet.ServletRequest.`
response	Page	Represents the response sent back to the client browser, generated by the servlet derived from the JSP page. The response object is an instance of class `javax.servlet.ServletResponse.`

session	Session	Represents the session created for the client. The `session` object is an instance of class `javax.servlet.http.HttpSession`.

Listing 4-7 shows a JSP page that accesses some of the methods in some of the implicit objects listed in Table 4-4.

Listing 4-7: JSP showing some implicit object methods

```
<%@ page contentType="text/html" %>
<html>
<head>
<title>Implicit Variable example</title>
</head>

<body>
<b>Server Info is:</b>
<br>
<%= application.getServerInfo() %>
<br>
<b>Host Name: </b><%= request.getRemoteHost() %>
<b>Session ID: </b><%= session.getId() %>
</body>

</html>
```

Notice that you need to reference only the object. Figure 4-2 shows the resulting page shown in Internet Explorer.

Figure 4-2: Internet Explorer screen showing page generated by Listing 4-7

For details on the supported methods attached to the implicit objects, refer to the appendixes or your Java documentation for the classes mentioned in Table 4-4. For example, to determine what methods are attached to the `request` object, reference the documentation for class `javax.servlet.ServletRequest`.

Using Beans in a JSP Page

Now that we've examined what JavaBeans are in the first half of this chapter, we are going to look at how we can use JavaBeans in our JSP pages. There are three important JSP actions required to interact with beans in your JSP pages. The jsp:useBean action is used to make a JavaBean known to your JSP page. The jsp:setProperty and jsp:getProperty actions are used to change a bean property and query a bean property respectively.

Here's a JSP action that makes the calculator bean shown in <u>Listing 6-1</u> known to a JSP page:

```
<jsp:useBean id="CalcBean" class="cbean.CalcBean" />
```

The above JSP action states that a JavaBean called `CalcBean` in package `cbean` goes by the name `CalcBean` when referenced in the JSP page. The `id` attribute does not need to be the same as the class name.

As noted in <u>Chapter 4</u>, the `jsp:useBean` action has the following format:

```
<jsp:useBean    id="beanInstanceName" class="className"
                scope="beanScope" type="classType" />
```

You can read more about the `scope` and `type` attributes later in this chapter. To use the bean shown in <u>Listing 6-1</u> now, you only need the `id` and `class` attributes.

To refer to the properties of `CalcBean`, you use the `jsp:getProperty` and the `jsp:setProperty` actions respectively. For example, to refer to the `operation` property, you code:

```
<jsp:getProperty name="CalcBean" property="operation" />
```

Here, the value of the `name` attribute must match the value of the `id` attribute in the `jsp:useBean` action.

To change the value of a property of our `CalcBean` from a JSP page, code a `jsp:setProperty` action as follows:

```
<jsp:setProperty name="CalcBean" property="operation" value="*" />
```

Variations on a Theme

You can use other JSP constructs to access bean properties, such as JSP expressions. For example, the following expressions also read and write the `operation` property of our `CalcBean`:

To read the value of the bean property:

```
<%= CalcBean.getOperation() ; %>
```

To write the value of the bean property:

```
<%= CalcBean.setOperation("+") ; %>
```

Use scriptlets to load bean properties into Java variables. For example, the following expression hangs onto the value of a bean property for possible use in your JSP pages:

```
<% int enteredOperand1 = CalcBean.getOperand1() ; %>
```

Now, you can use the Java variable `enteredOperand1` in expressions or scriptlets, as follows:

```
<% if (enteredOperand1 > 1000000) { %>
    <p> You Entered a <b>large</b> number
```

```
<% } else { %>
   <p> You Entered a <b>small</b> number
<% } %>
```

You can combine expressions within scriptlets as follows:

```
 <% if (enteredOperand1 > 1000000) { %>
   <p> You Entered <%= enteredOperand1 %>, a <b>large</b> number
<% } else { %>
   <p> You Entered <%= enteredOperand1 %>, a <b>small</b> number
<% } %>
```

You can combine JSP actions to access bean properties with static HTML text as follows:

```
<% if (enteredOperand1 > 1000000) { %>
   <p> You Entered <jsp:getProperty name="CalcBean"
                      property="operand1" />a <b>large</b> number
<% } else { %>
   <p> You Entered <jsp:getProperty name="CalcBean"
                      property="operand1" />a <b>small</b> number
<% } %>
```

Notice how the scriptlets include curly braces in order to construct valid Java statements.

The following construct will *not* translate:

```
<% int enteredOperand1  = <jsp:getProperty name="CalcBean"
                          property="operand1" /> ; %>
```

You can use the JSP actions in place of static page text, not Java code.

You've now seen some examples of JSP actions, expressions, and scriptlet code that make JavaBean properties known to a JSP page. Before you get into an example of displaying the values of `CalcBean` properties in Web pages, let's begin by exploring an example of how to collect values of `CalcBean` properties to the bean.

Collecting Values of Bean Properties from Web Pages

The following example shows a Web page that collects some inputs by using an HTML form, and then makes this data known to a JavaBean referenced within a JSP page. Figure 6-1 shows the Web page, `calcpage.html`, that collects the inputs.

Figure 6-1: Web page that collects input for the Calculator Bean

For the purpose of discussion, Figure 6-1 contains all of the essentials. You don't have to be a rocket scientist to deduce that the input field `Operand1` refers to the bean property `operand1`, and so on. Listing 6-2 shows the code for the Calculator Bean Web page shown in Figure 6-1.

Listing 6-2: Form used to get input for the CalcBean

```
<html>
<head>
<title>JSP Sample Page - Calculator Bean</title>
</head>
<body bgcolor="#dddddd">
<center>
<br>
<h1>A Simple Calculator</h1>
<!-- Here is the reference to the JSP that uses CalcBean -->
<form name="calcform" action="calculate.jsp" method="POST" >

<p>Enter Operand1 and Operand2 (Integers)
<br>and Select an operation From the Pull Down
<br>Menu, then Click <b>Calculate</b> to Continue
<br>
<hr width="50%">
<table>
<tr>
    <td><P>Enter Operand1:</td>
<!-- Entered value to be CalcBean property called operand1 -->
    <td><input type="text" name="operand1" value="" width="25"></td>
</tr>
<tr>
    <td><P>Enter Operand2:</td>
<!-- Entered value to be CalcBean property called operand2 -->
    <td><input type="text" name="operand2" value="" width="25"></td>
</tr>
<tr>
     <td><P>Select Operation:</td>
<!-- Selected value to be CalcBean property called operation -->
    <td><select NAME="operation">
            <option> +
            <option> -
            <option> *
            <option> /
        </select>
    </td>
<tr>
<!-- Click here to invoke the JSP page coded in the FORM tag -->
    <td><input type="button" name="Calculate"
value="Calculate"></td>
</tr>
</table>
<hr width="50%">
</center>

</form>
</body>
</html>
```

The first thing you may notice about `calcpage.html` is that the Web page is not a JSP. Changing the Web page into a JSP simply requires changing the name, depending on the Web server used. For Tomcat, files with an extension `.jsp` are recognized as JSP pages. If we named the preceding HTML page `calc.jsp`, Tomcat would take the Web page through the translation process, generating and storing a servlet on the server for subsequent execution. Because this page contains no JSP expressions, scriptlets, actions, or means of generating dynamic content, let's leave it as an HTML page for now.

`Calcpage.html` does have code that refers to the JSP that uses the calculator bean. The following line does just that:

```
<form name="calcform" action="calculate.jsp" method="POST" >
```

The following form input fields capture the data that corresponds to the properties of `CalcBean`:

```
<input type="text" name="operand1" value="" width="25">
<input type="text" name="operand2" value="" width="25">
<select NAME="operation">
        <option> +
        <option> -
        <option> *
```

```
         <option> /
</select>
```

When you enter numbers, select an operation, and click the "Calculate" button, the data entered in the form is sent to the server and the JSP page (`calculate.jsp`) is translated (if necessary), and then executed. The JSP `calculate.jsp` contains JSP coding constructs that access the entered data and invokes methods coded in `CalcBean.java`. You can dissect `calculate.jsp` in the following section.

Displaying and Using Values of Bean Properties in JSP Pages

Figure 6-2 shows `calculate.jsp` as displayed with the values of `CalcBean` properties derived from inputs entered in `calcpage.html`.

Figure 6-2: JSP page displaying values of CalcBean properties

Looking at the *rendered* JSP page tells little about the page's workings. After all, you're looking at generated HTML now, not JSP code. Listing 6-3 provides the code for `calculate.jsp`.

Listing 6-3: Using the CalcBean in JSP based on user input

```jsp
<%-- Tell JSP that this page renders HTML --%>
<%@ page contentType="text/html" %>
<%-- Tell JSP to use the bean CalcBean --%>
<jsp:useBean id="CalcBean"  class="cbean.CalcBean" />
<%-- Tell JSP to map CalcBean properties to like-named input parameters --%>
<jsp:setProperty name="CalcBean" property="*" />
<html>
<head>
<title>Show Calc Results bean</title>
</head>

<body bgcolor="#dddddd">
Here's what was entered from <i>calcpage.html</i>: <p>
<P>The next 4 lines show using the jsp:getProperty action to fetch bean properties
<%-- jsp:getProperty writes the value of the bean property where coded --%>
<p>Operand1 <b><jsp:getProperty name="CalcBean" property="operand1" /></b>
<p>Operand2 <b><jsp:getProperty name="CalcBean" property="operand2" /></b>
<p>Operation <b><jsp:getProperty name="CalcBean" property="operation" /></b>
<p>Result <b><jsp:getProperty name="CalcBean" property="result" /></b>
<br>
<P>The next 4 lines show using the request implicit object to fetch input properties
<%-- Access input parameters by accessing the request object. --%>
<p>Operand1 <b><%= request.getParameter("operand1")%></b>
```

```
<p>Operand2 <b><%= request.getParameter("operand2")%></b>
<p>Operation <b><%= request.getParameter("operation")%></b>
<p>result - not an input parameter!<b>
<%= request.getParameter("result")%></b>
<br>
<p>The next group of lines access and display bean properties by using scriptlet code, expressions
   and JSP actions
<%-- Invoke the 'get' method of the bean directly to get the property value --%>
<% int enteredOperand1  = CalcBean.getOperand1();
   int enteredOperand2  = CalcBean.getOperand2();
   if (enteredOperand1 > enteredOperand2  ) { %>
<%-- You may use expressions, actions and scriptlets together in your JSPs as shown below --%>
   <p> <jsp:getProperty name="CalcBean" property="operand1" /> is <b>larger</b> than <%= enteredOperand2 %>
<% } else { %>
   <p> <%= enteredOperand1 %> is <b>smaller</b> than <jsp:getProperty name="CalcBean" property="operand2" />
<% } %>

</body>

</html>
```

Here's the rundown on the JSP code included in `calculate.jsp`:

The JSP directive shown in the following tells the JSP engine that the results of the dynamic text generation is HTML:

```
<%@ page contentType="text/html" %>
```

You do not need to code this because the value of the `contentType` attribute is the default.

Here's the line in the JSP file that directs JSP to access the bean class `CalcBean`:

```
<jsp:useBean id="CalcBean"   class="cbean.CalcBean" />
```

As mentioned previously, JSP *actions* are coded as XML tags — note the ending tag `/>`. Also, in <u>Listing 6-1</u>, `CalcBean` was placed in a package called `cbean`. Note the value of the `class` attribute in the above `jsp:useBean` tag.

The value of the `id` attribute is the instance of `CalcBean` used in JSP page(s) that access the bean. You do not need to name the instance the same as the class name.

The `jsp:useBean` action does not make property values known to your JSPs. You cannot read or write bean property values without coding the `jsp:useBean` action. In the following section, you can take a closer look at JSP statements that manipulate bean property values.

JSP Code That Reads and Writes Bean Property Values

You read bean property values by coding the `jsp:getProperty` action and write property values by coding the `jsp:setProperty` action. If you accessed bean property values in your JSPs immediately after coding the `jsp:useBean` action, you would access the default, or initialized, `property` bean values. Make sure that your beans contain meaningful data before accessing the beans' `property` values. This can be done using the `jsp:setProperty` action in a variety of ways, which we look at next.

The `jsp:setProperty` action coded in `calculate.jsp` changes the values of *all* the bean properties with like-named input parameters.

```
<jsp:setProperty name="CalcBean" property="*" />
```

The input parameters come from the `<FORM>` submitted in `calcpage.html`. The single JSP action shown above matches the names of the `FORM` input elements to the names of the bean's properties and uses the values from the form as the values of the bean properties. Put differently, the above JSP action is a very handy and quick way of sticking input data into a bean occurrence.

Did you think it was a happy coincidence that the input `FORM` elements in `calcpage.html` have the same names as the properties in bean `CalcBean`? Later in this chapter, you learn that the names of bean properties do not need to match those of request-input parameters and you can learn how to change bean properties when the names differ.

You can get your hands on the values of the bean properties in a number of ways. Here is a JSP action that fetches the value of the `operand1` bean property:

```
<jsp:getProperty name="CalcBean" property="operand1" />
```

Here is a JSP expression that fetches the value of the FORM input parameter by accessing the implicit JSP `request` object:

```
<%= request.getParameter("operand1")%>
```

Caution   If you invoke the `getParameter` method for the request object for a nonexistent parameter, the method returns *null*, as shown in the following code:

```
<p>result - not an input parameter!<b>
<%= request.getParameter("result")%></b>
Refer to Figure 6-2, and you can see that JSP renders the word null on the page.
```

Here is a JSP expression that invokes the bean `accessor` method directly:

```
<%= CalcBean.getOperand1(); %>
```

Notice that `getOperand1` is the `accessor` method coded within `CalcBean` in accordance with the naming convention described earlier in this chapter; however, you can invoke any method in `CalcBean` in a JSP expression.

It's sensible to question which of the preceding methods is preferable for accessing values of bean properties. This question is tackled in the following section.

Comparing Ways to Access Bean Properties in a JSP Page

Not surprisingly, each method of accessing bean property data has certain advantages and disadvantages.

Using the `jsp:getProperty` action is more explicit inasmuch as the `jsp:getProperty` action is specifically designed to access bean properties. In addition, using the action may be more understandable by nonprogrammers than coding Java method invocations.

Using the bean `get` method invocation is shorter (requires less code) and can be used in some places where the `jsp:getProperty` action cannot. For example, the following line of JSP code causes a translation error:

```
<% int enteredOperand1  = <jsp:getProperty name="CalcBean"
                        property="operand1" /> ; %>
```

Whereas the following line of JSP code does not:

```
<% int enteredOperand1  =  CalcBean.getOperand1(); %>
```

Using the implicit `request` object is, strictly speaking, *not* accessing a bean property. The `request` object access fetches the bean property data because `CalcBean` and `calcpage.html` were craftily coded to use the same names for the bean properties and the input parameters.

Before you mutter to yourself, "What's the big deal?" take a look at the three lines of JSP code here:

```
<jsp:setProperty name="CalcBean" property="operation" value="Lou"/>
<p>From getProperty:  <jsp:getProperty name="CalcBean" property="operation" />
<p>From request object: <%= request.getParameter("operation") %>
```

The first line changes the value of the bean property to the string `Lou` by using the `jsp:setProperty` action. The second line fetches the recently changed value and generates the following line of output:

```
From getProperty:  Lou
```

The third line accesses the value of the parameter stored in the implicit `request` object. Before you read on, do you think the `jsp:setProperty` action changed the value of the parameter of the `request` object? Here's the line resulting from the expression that references the value of the parameter of the `request` object:

```
From request object: +
```

No surprise here because you've read that invoking the `getParameter` method from the `request` object does not access the bean property.

The moral of the story is that if you want to access a bean property, use a JSP technique designed to access bean properties. If you can get away with using the `jsp:getProperty` action, do so. If you want to store the value for use in a Java variable for use in a scriptlet or some other use, invoke the bean `accessor` method directly.

As previously mentioned, bean property names do not need to be the same as request input properties. If they are not the same, you cannot use the following JSP action to change the values of bean properties:

```
<jsp:setProperty name="CalcBean" property="*" />
```

Read on to see how to use other attributes of the `jsp:setProperty` action to change bean property values.

More About jsp:setProperty

Let's take a look at all the attributes of the JSP action `jsp:setProperty`. You may code three versions of `jsp:setProperty`. One version is what you've already seen, the version to change all bean properties with names matching those of the `request` object. Another version is shown in the following segment of code:

```
<jsp:setProperty name="CalcBean" property="operation" value="Lou"/>
```

This version follows the syntax for the `jsp:setProperty` action shown here:

```
<jsp:setProperty name="beanName"
                 property="propertyName"
                 value="scriptletOrStringValue" />
```

Here, you change the value of a bean property by referring to that property and assigning a value. You can code a string or dynamically generate a value. For example, you may use the value of an input parameter. Look at the code below and try to figure out what it does:

```
<jsp:setProperty name="CalcBean" property="operand1"
                 value="<%= CalcBean.getOperand2()%>"/>
```

Can you see that the JSP action above assigns the value of the bean parameter `operand1` to the bean property `operand2`?

You may think that you are limited to using simple expressions to assign bean property values from within your JSP pages. But this isn't the case. The piece of JSP code below gives a hint of the possibilities:

```
<% int enteredOperand1  = CalcBean.getOperand1();
   int enteredOperand2  = CalcBean.getOperand2();
%>
<%!  String whichIsLarger( int op1, int op2 ) {
     int theLarger =  ( op1 > op2 )? op1: op2 ;
     return "This Number is " + theLarger +
        " the larger of the two operands" ;
   } %>
<jsp:setProperty name="CalcBean" property="operation"
                 value='<%= whichIsLarger( enteredOperand1,
                                           enteredOperand2 )%>'/>

<!--Output from the below line:
    From Scriptlet. Operation = This number is 456 the larger of
    the two operands
--!>
From Scriptlet. Operation = <jsp:getProperty name="CalcBean"
                                property="operation"/>
```

Notice that the `jsp:setProperty` action enables you to code expressions that invoke *methods* coded within your JSP. Actually, you can invoke methods coded within your beans in addition to those coded within your JSP pages.

Using the form of `jsp:setProperty` with the `value` attribute does present a problem. Look at the following line of JSP for an illustration:

```
<jsp:setProperty name="CalcBean" property="operand1"
                 value='<%=request.getParameter("operand1")%>' />
```

When you pass this line of JSP to the translator, it rudely responds as follows:

```
Error: 500
Location: /examples/jsp/cbean/calculate.jsp
Internal Servlet Error:
org.apache.jasper.JasperException: argument type mismatch
```

Input parameters, or properties of the `request` object, are class `String`, whereas the bean property `operand1`, is type `int`. When you use the `value` attribute to assign data to bean properties, you need to ensure that the data type of your data agrees with the data type of the bean property. The following code shows the essential strategy:

```
<%
  int oper1 = 0 ;
  try {
     oper1 = Integer.parseInt(request.getParameter("operand1"));
  }
  catch (NumberFormatException myNFE) {}
%>
```

```
<jsp:setProperty name="CalcBean" property="operand1"
                 value="<%= oper1 %>" />
```

Generate an equivalent value of the desired class or primitive type and use a JSP expression to assign the value to the bean property.

Note that you can assign a value, as follows, without worrying about data types:

```
<jsp:setProperty name="CalcBean" property="operand2" value="25"/>
```

The other form of the `jsp:setProperty` action has the following format:

```
<jsp:setProperty name="beanName"
                 property="propertyName"
                 param="paramName" />
```

You can use the preceding format for the `jsp:setProperty` action when you want to assign the value of an *individual* input parameter from the `request` object to a bean property. In addition, you do not need to write JSP code to perform data type conversions because the result of the `jsp:setProperty` action with the `param` attribute performs most conversions automatically.

The JSP following code automatically performs the conversion from class `String` to type `int`.

```
<jsp:setProperty name="CalcBean" property="operand1"
                                 param="operand1" />
```

The form of the `jsp:setProperty` action with the `param` attribute performs type conversions from class `String` to any primitive type or object representation of a primitive type (Integer for `int`, Float for `float`, for example).

Note The form of the `jsp:setProperty` action which copies all values of input parameters to like-named bean properties also performs automatic type conversions.

You can see how useful it is to have access to JavaBeans in your JSP pages. You may be wondering if you can leverage your beans across several JSPs in your application. Well, you don't have to wonder anymore! The following section discusses how to use your beans with multiple JSPs in your application.

Top

← Prev   Next →

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

A Typical JSP Execution

Here's the rundown on a typical JSP execution. Figure 3-1 summarizes the following steps:

1. The customer enters the name of the JSP page in the browser location area or clicks some action item in an
   HTML page (button or menu selection, for example) that invokes the JSP page.

2. The browser sends the JSP page to the Java-enabled Web server as an HTTP request.

   Note   The JSP specification does not mandate use of the HTTP protocol. Some companies use JSP with their
                proprietary protocols. However, a JSP-enabled server must, at a minimum, support HTTP.

3. The Web server recognizes (by the `.jsp` extension) that the JSP page requires special handling and forwards the
   JSP to the JSP engine. The JSP engine translates the JSP page into a Java class file, implementing the special
   JSP tags found in the page.

4. The server sends the class file to the servlet engine, which generates a servlet from the class file by using the
   Java compiler and associated files.

5. The servlet engine compiles, loads, and executes the generated servlet. Successful execution of the generated
   servlet creates an HTML file, which the server pumps back to the customer's browser for display.

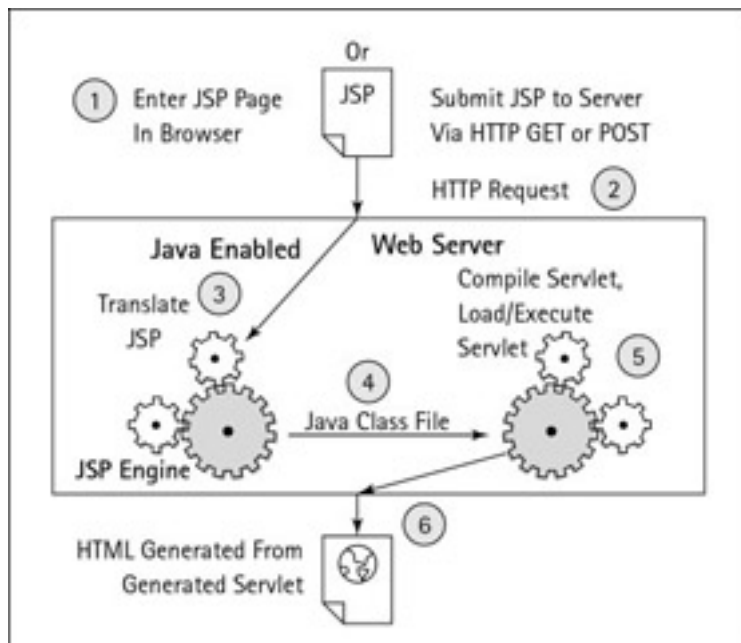6. The newly generated HTML file is displayed in the customer's browser.

Figure 3-1: Executing a JSP Page

shows the process for using, or of calling, a JSP page for the first time, or if the JSP page changed from the first invocation. The process is different when a customer calls a page that has been viewed before. The JSP engine will not retranslate the page, nor will the servlet engine recompile the class file. The servlet engine creates a new thread to handle the execution of the generated servlet.

A JSP page may use other files, including other JSP pages. The JSP engine translates other, referenced JSP page files into their own class files. These class files are passed to the servlet engine for compilation and execution.

1. Enter JSP Page In Browser

Or

JSP

Submit JSP to Server Via HTTP GET or POST

HTTP Request 2
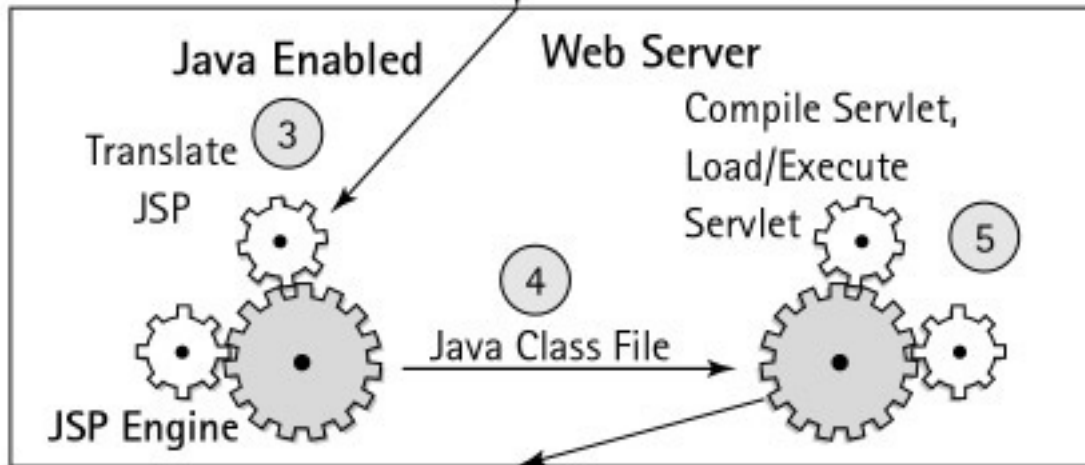
Java Enabled **Web Server**

Translate 3
JSP

Compile Servlet,
Load/Execute
Servlet 5

4

JSP Engine

Java Class File

6

HTML Generated From
Generated Servlet

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

.

Part II: JavaServer Pages

Chapter List

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

You have now read about the major classifications of J2EE APIs and have read a short description of each. In the
following chapter, we will dive into JavaServer Pages in depth.

Tag Extensions

Tag extensions or custom tags are application-defined language extensions to JavaServer Pages. You can use custom tags in your JSP pages to do the following:

- Create static text (HTML, XML) or JSP output

- Create objects that are hidden from the JSP page or that can be seen as scripting variables in the JSP page

- Process a block of JSP page text

- Determine if a section of the calling JSP should be processed or skipped

Custom-related tags are grouped together into a *tag library*. This chapter talks about creating and using tag libraries. Some custom-tag activities relate to the tag library, whereas others relate to individual tags within the library.

A tag is a bean that implements either the `javax.servlet.jsp.tagext.Tag` or `javax.servlet.jsp.tagext.BodyTag` interfaces. The two classes, `javax.servlet.jsp.tagext.TagSupport` and `java.servlet.jsp.tagext.BodyTagSupport`, are convenience classes that the developer can subclass to create tags as well. These classes implement the aforementioned interfaces respectively. You can read more about these two interfaces and classes later.

A simple description of how custom tags work is that when the JSP engine comes across a tag in a JSP page, it invokes a method of the bean with which this tag is related. Throughout the rest of this chapter we will elaborate on this process.

Custom tags are referenced in your JSP page as an XML element, such as the examples shown below:

```
<mytaglib:atagwithbody>
This tag has a body
</mytaglib:atagwithbody >
```

Custom JSP tags may contain a body. The tag body can be static text or JSP code. The particulars of the tag's behavior govern how the JSP container interprets the tag body.

Custom tags may also be empty, that is without a body, as shown in the following example:

```
<mytaglib:anemptytag />
```

Custom tags may also contain attributes, as in the following:

```
<mytaglib:anemptytagwithattrs attr1="attr1Value" attr2="attr2Value"  />
```

You can *nest* tags to any practical level, as shown in the following example:

```
<mytaglib:outertag >
    This tag can contain plain old text
```

```
    <%= session.getAttribute("userID") %>
    <mytaglib:innertag anattr="attrvalue">
        <p>This tag <I>too</I>
    </mytaglib:innertag>
</mytaglib:outertag>
```

Custom tags follow XML syntax rules. See for details on XML syntax. You may be thinking that if a custom tag causes the invocation of a bean, why use custom tags at all? Why not code the bean and use the JSP action `jsp:useBean` instead? Let's tackle this question in the following section.

.

A Simple XML Document

This section shows one of the simplest XML documents you can create and also explores some of the document's
properties. Some important rules about writing XML documents that should be noted are:

- XML is case-sensitive

- XML tags must have a corresponding closing tag, or use a special self-closing syntax

- Attributes in an XML tag must use quotes

Now, let's look at the code for the simple XML document:

```
<?xml version="1.0" standalone="yes"?>
<First>
My First XML Document
</First>
```

This document has code that tells a parser that it is an XML document. The document's content is the phrase "My
First XML Document." The document's first line tells the parser that the document is an XML document. Notice the
presence of the characters `<?` and `?>`; these characters indicate the presence of an XML *processing instruction*. The
word after the characters `<?` (`xml`, in this case) tells the parser the particular processing instruction that is an XML
declaration.

XML documents are free form, because XML parsers typically do not care about column positions or white space.
Thus, XML document authors should take some time making their documents easy to read with judicious use of tabs,
white space, and blank lines.

XML processing instructions and tags often use XML *attributes*, which are name/value pairs separated by an equal (`=`)
sign. The values must be enclosed in quotes. (Although XML requires the use of quotes, most HTML values do not
require the quotes.) The XML declaration requires the use of the `version` and `standalone` attributes. For now, it's
important to note that the XML declaration shown here states that the document conforms to XML version 1.0 and
does not require any other documents for parsing its content.

It is worth mentioning that this document does not contain any display or formatting information. Recall that a big part
of XML is the separation of content from display. Therefore, you would need a style sheet and a way of telling the XML
document to use that style sheet to display the XML document. Let's use the following simple style sheet, saved as
`forfirst.css` in the same directory as the XML document. Notice that the style sheet refers to the tag `<First>`
used in the document.

```
First  {display: block; font-size: 36pt; font-weight: bold; color="00FF00";}
```

The second, italicized processing instruction associates the style sheet with the XML document:

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/css" href="forfirst.css"?>
<First>
```

```
My First XML Document
</First>
```

shows this simple XML document displayed in Internet Explorer 5.0.
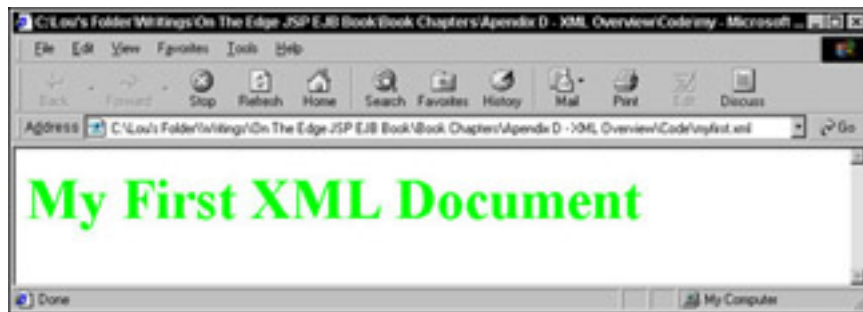
Figure D-1: A simple XML document displayed in Internet Explorer 5.0

XML Document Components

XML documents consist of data and markup texts. The data is what the author encodes and the markups tell the XML parser how this data is organized and structured. Markup includes processing instructions, comments, elements, entity references, CDATA delimiters, and Document Type Definitions (DTDs). All of these markups are case sensitive and are described later in this appendix.

The preceding simple XML document contains an XML declaration, a processing instruction that associates a style sheet with the document, and a tag (or element). The XML declaration, a processing instruction, is an optional statement that identifies the XML version in use; currently, version 1.0 is the only version. If present, the XML declaration must be the first statement in the document. If an XML document must be displayed, a processing instruction that associates one or more display documents (such as a CSS) must be coded as well.

XML comments begin with `<!--` and end with `-- >`. However, an XML author cannot place comments with reckless abandon. Comments cannot be coded before the XML declaration. Comments may not be coded inside element tags, and they may not be nested. Comments may not include two successive dashes other than those that start and end the comment.

XML document content is enclosed in tags, or *elements*. Our simple example explored previously contains one element coded as `<First>`. XML documents require that one element enclose all others. In XML lingo, that special element is called the *root element.* Stated differently, every element coded in an XML document must be sandwiched between the opening and closing tags of the root element. Elements that are coded within (sandwiched between) other elements are known as *child elements.* Listing D-1 shows an XML document with child elements.

Listing D-1: XML document with child elements

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/css" href="mystylesheet.css"?>
<Root>
     <Employee_Data>
          <Name>
               John Q. Public
          </Name>
          <Department>
               Human Resources
          </Department>
          <Hire_Date>
               May 1 2000
          </Hire_Date>
          <Review_Date>
               December 1 2000
          </Review_Date>
     </Employee_Data>
</Root>
```

All elements other than `<Root>` (which is not a keyword) are child elements. Style sheet display attributes may be attached to each element or a child element may inherit display attributes from its parent element. For example, if the file `mystylesheet.css`, associated with the above XML document, contained only the following line, then all the content in the XML document would be 36-point blue text.

```
Root      {display: block; font-size: 36pt; font-weight: bold; color="blue";}
```

However, if the following line were also present, the content for element `Hire_Date` would be 18-point red text positioned slightly to the left.

```
Hire_Date  {position: absolute; top:90; left:190;display: block; font-size: 18pt;
            font-weight: bold;color="red";}
```

Element names must begin with either the underscore character or a letter; following characters may be just about anything except spaces. Also, element names are case sensitive. For instance, `</aTag>` is not the closing tag for `<Atag>`.

XML enables tags to contain no data. For example, HTML tags that contain no data have no closing tags (for example, no `</IMG>` tag is required). HTML may or may not ignore unknown tags. However, XML must be able to recognize and process every tag present in a document. To deal with tags that contain no data, XML allows for *empty tags*. An empty tag is closed with `/>` (for example, `<IMG/>`).

XML enables an author to categorize data by using meaningful tag names and organize data by developing a hierarchy between parent and child elements. An author may also attach *attributes* to an element. As in HTML, XML enables an author to code name/value pairs with elements. For example, the following element contains one attribute:

```
<Department   Location="Home Office">
```

Notice that an author could have coded a separate location element.

Then the question arises of when an author should code attributes instead of elements. Rather than supply a laundry list of dos and don'ts, a simple rule works best here. If an author needs to access and independently display some data or the data has structure, it may work best to encode that data in an element. If an author needs to encode some data *about* the data (sometimes called metadata) or a piece of data does not need to be independently accessed, the best approach may be to encode that data in an attribute. A good example of using attributes instead of elements is coding the `Height` and `Width` attributes of the `<IMG>` tag in an HTML document.

Entity references are markup that the XML parser replaces with a single character. For example, authors who need to encode data that includes a less-than sign (<)would use the entity reference `<`. If the author didn't use this entity reference, the XML parser would interpret < as the start of a tag.

Table D-1 contains examples of entity references. Note the presence of the semicolon after each entity reference.

Table D-1: Predefined XML Entity Reference

Entity Reference	Character
&	&
'	`
>	>
<	<
"	"

An XML author may want to include a block of text as is, without having the XML parser perform any translations. The data may contain numerous entity characters (such as the preceding example), and coding the entity references may become tiresome. XML has markup that enables an author to include text as is, including comments, called the *CDATA* section. Just enclose text between the CDATA delimiters `<![CDATA[` and `]]>`.

Rules for Well-Formed XML Documents

XML requires that an author follow certain rules when creating an XML document. These rules were designed to enable XML parsers to understand properly constructed XML documents. In XML lingo, an XML document that obeys the rules, shown in the following, for proper construction is said to be *well-formed*.

- The XML declaration, if present, must be the first statement in the document.

- Every XML tag that contains data must have a closing tag. Close empty tags with `/>`.

- Include a root element.

- Put attribute values in quotes.

- Use `<` only to start element tags; use `&` to start entity references.

- Enclose every element tag (except the root) inside another. For example, the following line is not well-formed XML:

`<Outer> This is NOT <Inner> Well-Formed </Outer> XML </Inner>`

- Use the five entity references shown in Table D-1.

Top

EJB & JSP: Java On The Edge, Unlimited Edition

 by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

.

Document Type Definitions

XML enables an author to create entirely new markup languages with tags that contain industry-specific language.
The markup language creator can define the language with a *Document Type Definition*, or DTD. In short, DTDs
define a set of rules that govern the relationships among the tags contained in a document. For example, a DTD
describing a company's personnel may specify that every `Employee` element have at least one `Hire_Date` child
element and one and only one `Name` child element.

When the tags in an XML document conform to the specifications in an associated DTD, the XML document is said to
be *valid*. An XML document can be well-formed without being valid. Also, DTD keywords are case sensitive.

A Simple DTD

Let's look at a simple XML document with a DTD:

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE First [
     <!ELEMENT First (#PCDATA)>
]>
<First>
My First XML Document
</First>
```

The three italicized lines are the *Document Type Declaration,* not a DTD. The declaration is delimited by `<!DOCTYPE`
`rootname and ]>;` the DTD is what is sandwiched between these delimiters. The `rootname` of the XML document
must follow the starting delimiter.

The DTD author may opt to store the DTD as opposed to coding the DTD in the XML document. If so, the Document
Type Declaration changes to:

```
     <!DOCTYPE     First      SYSTEM "myDTD.dtd">
```

The entire DTD, `<!ELEMENT First (#PCDATA)>`, states that the element `First` must contain parsed character
data only, or text that has no markup, such as child elements. The following section covers additional element
declarations that a DTD author can code.

Element DTD Declarations

DTD element declarations begin with `<!ELEMENT` and end with `>`. The name of the element must follow the starting
delimiter. Element attributes specified in the DTD for that element follow, usually in parentheses. Each element should
have one, and only one, declaration. The order of declarations is not important; XML enables forward and backward
references.

[Table D-2](#) shows some DTD element declarations and their meanings.

Table D-2: DTD Element Declarations and Their Meanings

DTD Element Declaration	Meaning
`<!ELEMENT usuallyARootElement ANY>`	No restrictions on element use
`<!ELEMENT Employee_Data (Name)>`	One occurrence of `Name` for `Employee_Data`
`<!ELEMENT Parent (Child1, Child2)>`	One occurrence of `Child1` and `Child2` for `Parent`
`<!ELEMENT Parent (Child, Child, Child)>`	Exactly three occurrences of `Child`
`<!ELEMENT Parent (Child+)>`	One or more occurrences of `Child`
`<!ELEMENT Parent (Child*)>`	Zero or more occurrences of `Child`
`<ELEMENT Parent (Child?)>`	Zero or one occurrence of `Child`
`<!ELEMENT Parent (Choice1 | Choice 2 | Choice 3) >`	Exactly one of `Choice` 1, 2, or 3
`<!ELEMENT Parent (#PCDATA | Child)>`	Both character data and occurrence of `Child`
`<!ELEMENT Parent (Child1 | Child2 | Child3)*>`	Zero or more occurrences of `Child1`, 2, or 3
`<!ELEMENT Parent (Child1 ,Child2, Child3)+>`	One or more occurrences of `Child1` and 2 and 3
`<!ELEMENT Parent (Child1, (Child2 | Child3 ) ) >`	One occurrence of `Child1` followed by one occurrence of `Child` 2 or 3
`<!ELEMENT Parent (Child1?,( Child2,Child3)+ | Child4*)) >`	Zero or one occurrence of `Child1` followed by one or more occurrences of `Child2` and `Child3` or zero or more of `Child4`

Note how a DTD author can build complex relationships between elements by combining different requirements inside nested parentheses.

DTD Entities

DTDs support the inclusion of text from internal and external sources. In essence, the DTD author codes a *general entity reference* that, when processed, substitutes text in the XML document for the entity reference. The mechanism is identical to that described for the XML entity references. First, the DTD author codes the `<ENTITY>` tag:

```
<!ENTITY    entityName      "Replacement Text">
```

The XML document may contain the element:

```
<SomeTag> Where is that &entityName; going? </SomeTag>
```

During processing, the XML processor substitutes `Replacement Text` for `&entityName;`. The semicolon must be the last character in the entity reference.

External general entities enable the author to include text from other locations into a document. For example, the author can code the following line, in which `someFile` can be a URL or a file on the network or on the local machine.

```
<!ENTITY  entityName   SYSTEM  "someFile">
```

The internal and external general entities become part of the XML document, not the DTD. XML provides a mechanism, called a *parameter entity reference*, that enables authors to substitute text in the DTD. The coding of a parameter entity reference is similar to that of a general entity reference, but with two differences: Parameter entity references start with a percent sign (not an ampersand), and parameter entity references cannot appear in the document content.

Here is a DTD entry for a parameter entity reference and some references to the entity:

```
<!ENTITY  %   entityNameList    "Replacement Text">
<!ELEMENT   anElement1         (Child1, (%entityNameList;)) >
<!--anElement 2 through 999 follow with the same parameter entity reference-- >
<!ELEMENT   anElement1000  (ChildA, (%entityNameList;)) >
```

If the author needs to add or remove an element from the list, the author can change the parameter entity reference instead of each `<!ELEMENT` tag.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming Techniques.

Related XML Technologies

As previously mentioned, XML stresses the separation of content from presentation. This appendix uses CSS for XML document display. However, a strong up-and-coming contender for XML display is *XSL*, the Extensible Style Language. XSL, an XML application, consists of a transformation language and a formatting language. The transformation language specifies how one XML document may be transformed into another — for example, how the custom tags used in an XML document may be changed to tags of another. The formatting language, similar to CSS, describes how an XML document is rendered.

XML developers can mix and match tags from multiple applications. However, a mechanism needs to be in place to enable XML applications to distinguish between elements and attributes of the same name. A related XML technology called *Namespaces* enables an XML developer to prefix custom tags by directing the XML parser to reference a unique resource (dataset, URL).

The XML community is crying out for more robust technologies than DTDs to perform XML document structure and element/attribute validations. *XML Schemas* is an attempt to provide database-like validations, such as data types, to XML elements.

With the proliferation of Web pages on the Internet, technology that facilitates quick and accurate searches is in increasing demand. The XML application *RDF*, the Resource Description Framework, encodes data about data, or metadata. In other words, RDF provides a consistent way to describe metadata. The basic idea is that RDF standardizes vocabularies used to describe metadata and should provide a strong foundation for building applications to search XML documents.

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Objects and Components

The basic idea behind object technology dates back to the 1960s (yes, the 1960s) and the work done by Xerox and others with the Smalltalk and Simula programming languages. Object technology treats software as a set of cooperating objects. Each object is derived from a *class,* or a template describing similar objects, or objects with the same behavior. Objects have a well-defined interface that allows for complete and concise communication. The objects have no knowledge of other objects' internals; all they know is what is exposed in the interface to other objects. Object software development environments allow for the creation of objects that share some, or most, of the behaviors of other objects.

Object technology makes possible the development of software *components.* A software component is simply a piece of code with a well-defined interface. In other words, a software component is an object that can be reused. The goal is that an object development team can create libraries of interrelated software components that model fundamental, low-level behaviors such as database access (finely grained components) or application tasks such as printing (coarsely grained components), and that object applications can be constructed by pulling together existing components. This plug-and-play property has always been the Holy Grail of application development, and object technology brings the grail closer than before. As discussed later in the chapter, developing and deploying software components are integral parts of the modern enterprise applications environment.

Of course, one milestone in the development and refinement of object technology is the Java programming language. Scarcely six years old, Java is dominant on the Internet, is taught by any school that teaches computer science (even some high schools), and is gaining a frothy head of steam as it charges into the business data processing community. Java is well suited to serve as the backbone of a host of technologies that help the enterprise application developer accomplish his or her difficult task.

How does the application development team leverage the power of objects and components when developing applications? The answer, described in the next section, is that the team combines the use of objects and components with application servers.

Server Component Architecture

Developing n-tier applications involves writing much more than the code that implements the business logic. Many ancillary but important functions — such as transaction support, security, resource allocation, and dealing with hardware failures — have to be implemented. In addition, you can't forget numerous procedural and analysis matters, such as performance monitoring and tuning, database administration, and training.

Thank goodness for application servers! The *raison d'être* for application servers is to supply the services mentioned, plus many more, to enterprise application development teams. Freed from the drudgery of providing these services from scratch, application developers can concentrate on the job at hand — solving the problems of their customers.

Still, the application programmer's code has to communicate with these application servers in order to use the various system services these servers offer. Given the numerous services required in a substantial enterprise application, the developer still faces a monumental task. The developer needs a consistent, well-defined method of having his or her code communicate with application server code that provides much-needed services. In other words, the developer

needs a server-side architecture to provide and help enforce a consistent method for interacting with application servers.

The server architecture should provide a set of well-defined interfaces to application server services. The developer need not know or care how these services are implemented on the application server; all he or she needs to know is the interface required to access the required services.

Ideally, the architecture should allow for plug-and-play components. For example, the application should allow a database connection pooling component X from company ABC to be compatible with a transaction support component from company XYZ on the same application server, as long as both components follow the same server-side architecture.

All this is starting to sound a bit like using object technology and software components, isn't it? For this reason, the major players in the server-side architecture business have embraced some flavor of object technology. The application server vendors believe in the strength and value of components and realize that components combined with application servers offer customers a powerful combination: the implementation of a server-side architecture that enables customers to plug in their own components to communicate with server components, thereby making the myriad services available on the application server. A server-side architecture that allows for the use of software components is called a *server component architecture.*

Several server component architectures exist today. Microsoft's DNA (Distributed Internet Applications Architecture) ties together various Microsoft technologies with a Windows platform to provide a server-side component architecture. CORBA (Common Object Request Broker Architecture) is a specification for the use of distributed objects. J2EE (Java 2 Platform, Enterprise Edition), like CORBA, is a specification for a server-side architecture.

What About Web Applications?

Simply put, a Web application is an application with a Web browser serving as the client (implementation of the presentation layer) working with an application server known as a Web server. A Web server is an application server that is good at speaking HTTP (Hypertext Transfer Protocol), a browser's native tongue.

The information in the previous sections about n-tier applications also applies to n-tier Web applications. Any server-side architecture that supports Web applications requires additional features, or should provide additional services, to facilitate Web application construction. Notably, the server architecture should allow a customer to get to the server's services by using HTTP from the client, either directly or indirectly.

Some server component architectures, such as J2EE, permit the use of a Web server and an application server, possibly housed on separate hardware.

The Java programming language is a good choice for developing software components. Because Java is not tied to any particular hardware or operating system, components written in Java can, ideally, be hosted on different platforms (remember Sun's slogan "Write Once, Run Anywhere"?) that contain a Java runtime. Whereas the presence of a required runtime could be a maintenance issue for using Java on the client, it is not much of an issue for using Java on the server (unless the company has thousands of servers!).

Java's ease of use, combined with powerful features for component development, makes Java a natural for server-side development. Sun Microsystems, along with industry participants, created J2EE to provide a Java-based server-side component architecture. J2EE is the result of consolidating several Java-based server technologies into a single specification.

J2EE describes how a customer's component interacts with server components to use the application and Web server's services. Basically, J2EE is a set of Application Programming Interfaces (APIs) that defines the interaction between customer components and server services. Hence, the job of the enterprise application developer working within the J2EE framework is to understand the how, when, and why of using the various APIs to invoke various application and Web server services.

J2EE is based on the concept of *containers.* A container is an environment that exposes application and Web server

services to a housed software component. The component interacts with the server by declaring the services required by the component, and the container makes these services available to the housed component. This book focuses on two J2EE containers: the Web server container that houses JSPs (and Java servlets, too) and the application server container that houses EJBs. Figure 1-2 illustrates the basic ideas behind using containers.

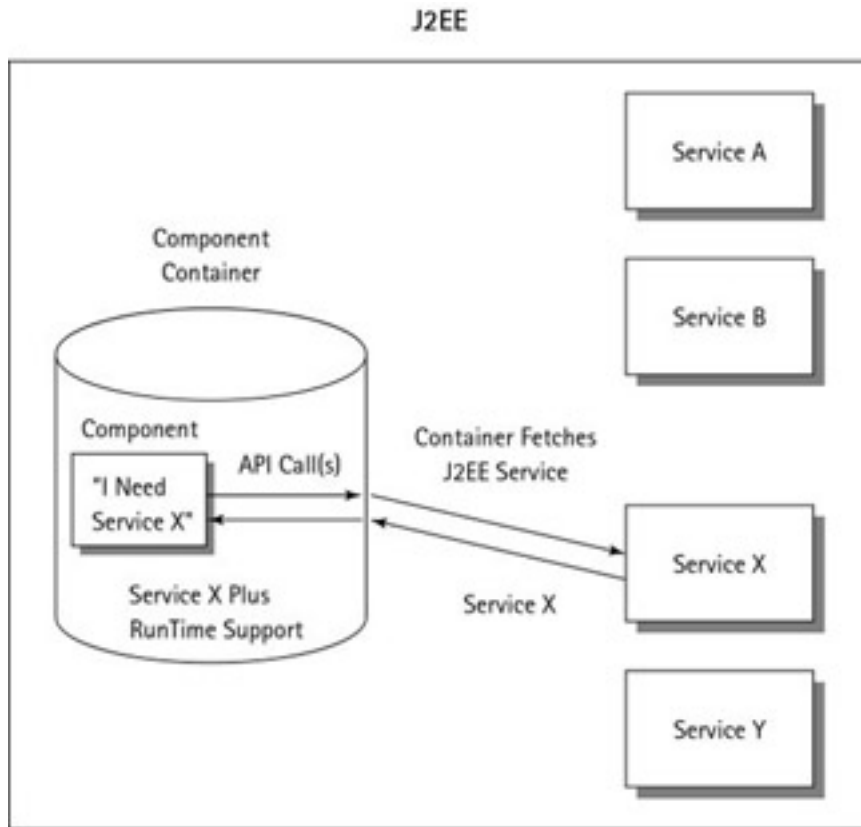**Figure 1-2:** A J2EE container at work

In Figure 1-2, a component living in a J2EE container needs service X from the application server. The component has code that makes one or more API calls to the service. The container fetches the service and makes that service available to the needy component. The container also establishes a runtime so the component can use the service without having to tend to runtime concerns, such as memory management.

JSP and EJB with the J2EE Specification

JSP is a key component in developing Web-based applications. A JSP is a Web page that contains static HTML or XML, as well as JSP scripting tags. JSPs are compiled into Java servlets on the Web server.

Because JSPs eventually become Java servlets, one can rightfully assume that a developer can use Java servlets to dynamically generate HTML and XML, thereby removing the need to learn and use yet another API. However, JSPs are a better solution than servlets to the problem of providing dynamic Web content because JSPs are easier to write and provide a more straightforward way of separating presentation layer code from application logic code.

EJB is the J2EE component that defines a component model, or architecture, for creating software components. These components are the heart and soul of the enterprise application. Although the long list of J2EE APIs is necessary for the implementation of any substantial n-tier application, most of the API sets define interfaces to support services and external resources. EJB is *the* API that deals with creating application components and how these components interact with the other J2EE API sets (see Figure 1-3).
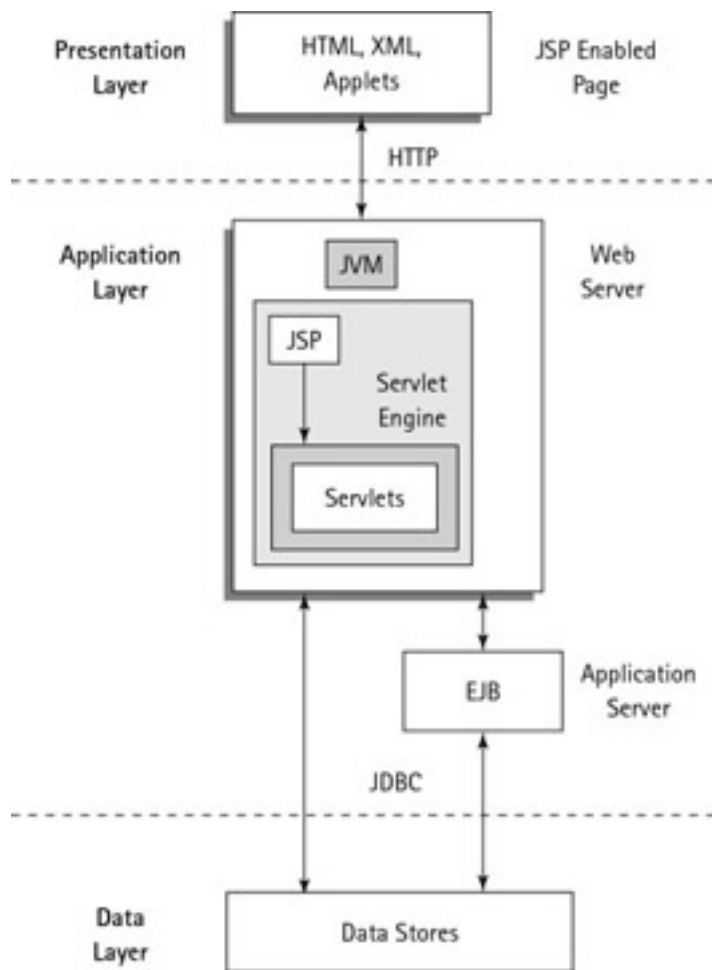
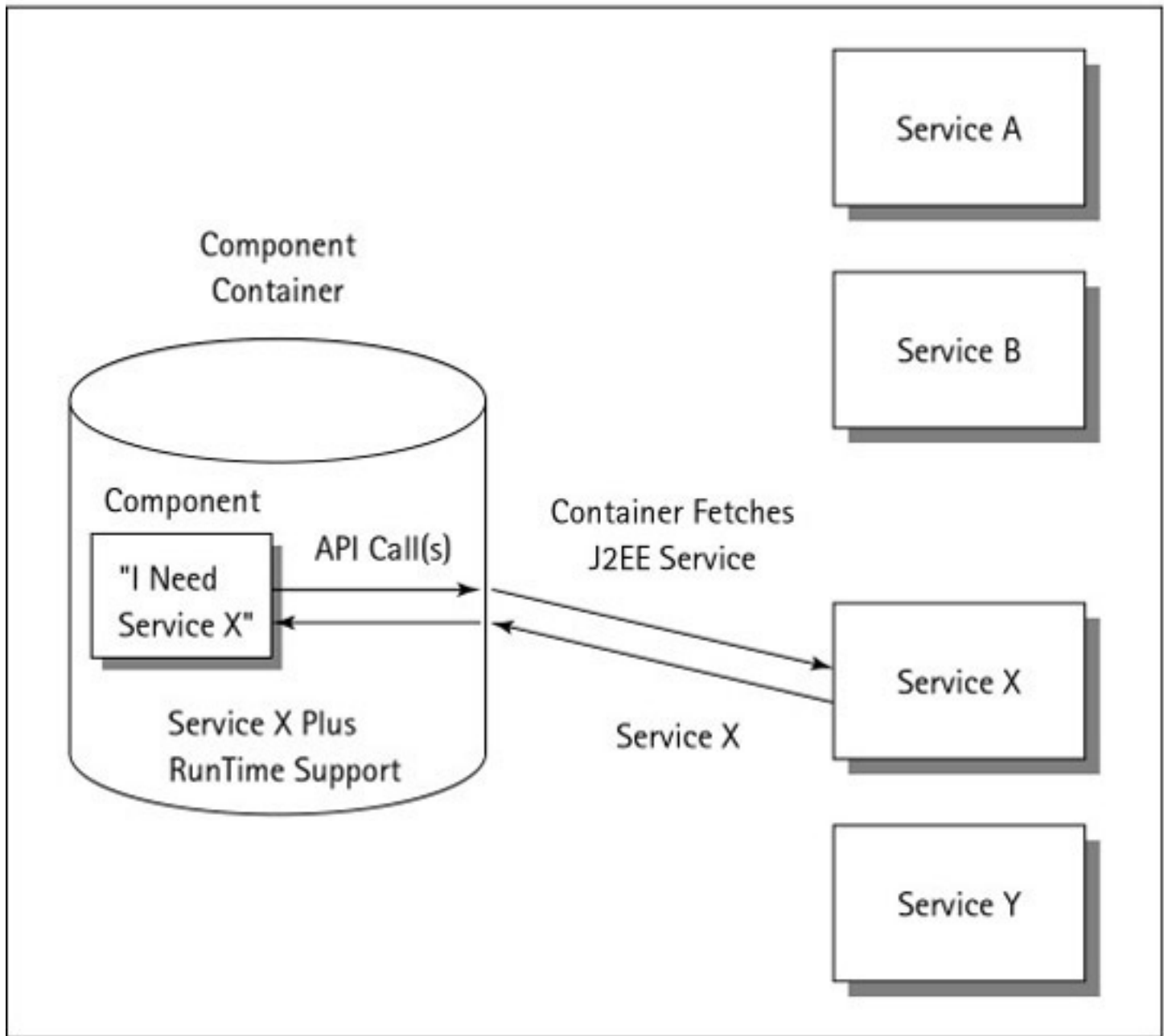**Figure 1-3:** A high-level overview of a Web application request

shows how a request made from a Web page containing a reference to a JSP could flow up and down through the various components and tiers of a Web application. The dashed lines represent tier boundaries. The Web server and the application server can reside on the same hardware. Also, please note that the Web server, Java Virtual Machine (JVM), and the application server contain more components than are shown here.
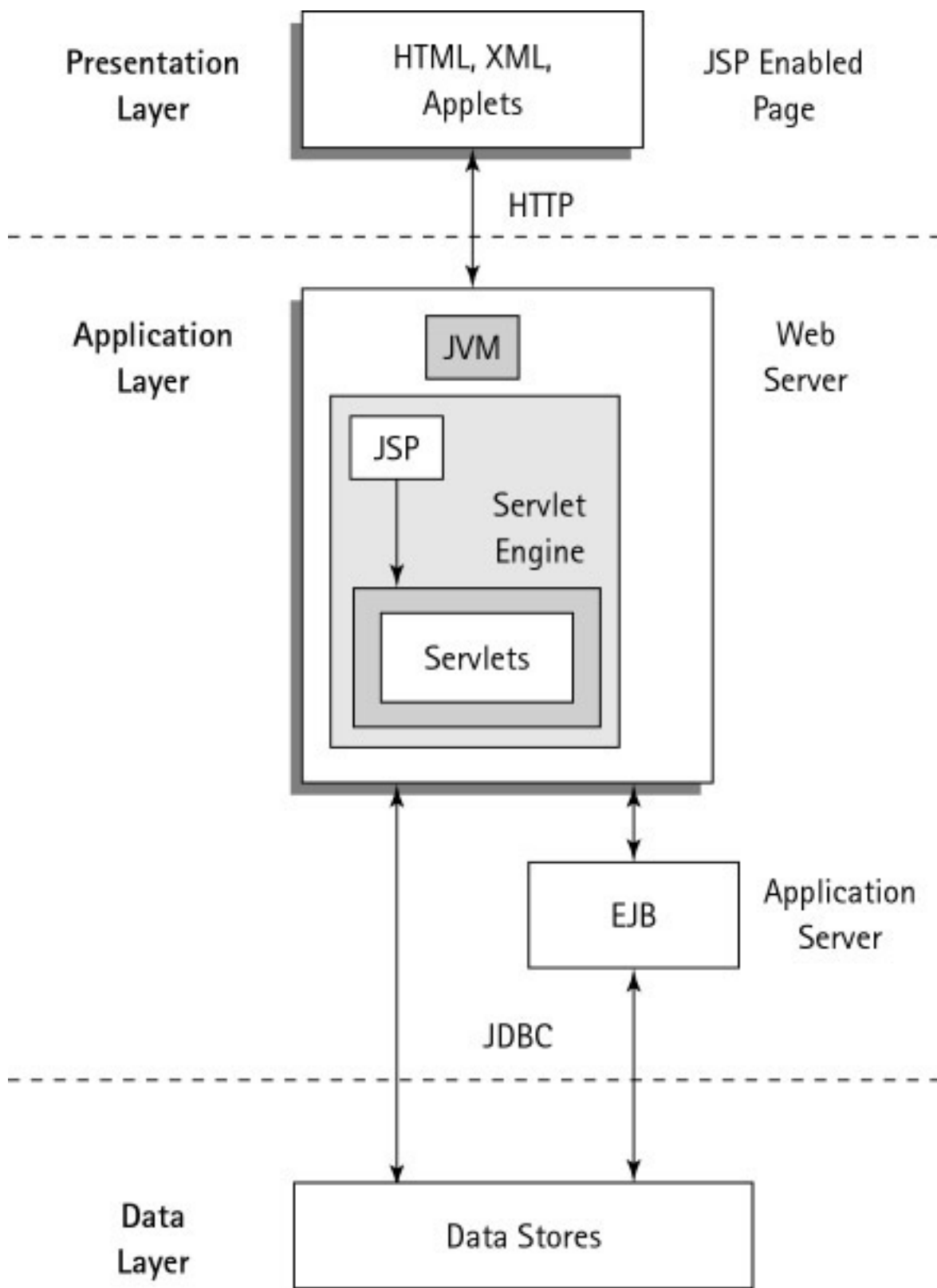
Here's a high-level overview of a client interacting with a Web application that uses J2EE technology:

1. Client makes a request by filling out a form on a Web page that contains a reference to one or more JSPs and clicking a submit button.

2. The request (entered data) with other information coded on the Web page goes to the Web server via HTTP.

3. The Web server recognizes the page as JSP and sends the page to be compiled into a Java servlet.

4. The newly compiled servlet communicates with one or more EJBs on the application server to get data or goes right to the data store (bypassing the EJBs), or both.

5. The EJB components fetch data from the data store and return this data to the Web server.

6. The JSP creates a new Web page based on the data fetched by the servlet or EJB and sends this page to the client via HTTP.

7. The client displays the new, JSP-generated Web page in the browser.

The preceding scenario is a condensed version of what happens. Every step of this process is explained in detail in the remaining chapters of this book. Note that every step involves using either JSP or EJB, which underscores the importance of JSP and EJB in the J2EE architecture.

J2EE

Service A

Service B

Component
Container

Component

"I Need
Service X"

API Call(s)

Container Fetches
J2EE Service

Service X

Service X

Service X Plus
RunTime Support

Service X

Service Y

| Presentation Layer | HTML, XML, Applets | JSP Enabled Page |

HTTP

Application Layer	JVM	Web Server
	JSP	
	Servlet Engine	
	Servlets	

| | EJB | Application Server |

JDBC

| Data Layer | Data Stores | |

EJB & JSP: Java On The Edge, Unlimited Edition

by Lou Marco   ISBN: 0764548026

Your Guide to Cutting-Edge J2EE Programming
Techniques.

Summary

Client-server architectures are used to implement enterprise applications as discrete layers of functionality. Any
serious application built on an n-tier component architecture can access services needed through application servers.
Java 2 Enterprise Edition and its API sets have been introduced as a way to develop n-tier applications. In the J2EE
architecture, information flows from JSP-enabled client Web pages, to EJBs, to a data store, and back.