



Enterprise Java
A Part of the Java: Under the Hood Series

Enterprise Java
A Part of the Java: Under the Hood Series
Version 2.0

August, 2002

Copyright © 1999-2002 JLicense, Inc. 13664 Red Fox Court, Rapid City, SD 57702,
U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of JLicense, Inc.

Under the Hood is a trademark of JLicense, Inc. in the U.S. and other countries. Sun, JDK, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All other names are used for identification purposes only and are trademarks of their respective companies.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

This manual prepared by:

JLicense, Inc.
13664 Red Fox Court
Rapid City, SD 57702
USA

Web: <http://www.javajicense.com>

Course Description

Course Length:

5 days

Audience:

This course is intended for experienced Java developers who want to learn how to develop Java applications that take advantage of the various technologies of J2EE, the Java 2 Platform, Enterprise Edition. Students should be knowledgeable in the writing and compiling of Java classes, JDBC, SQL and exception handling. Helpful prerequisites include knowledge of RMI and an understanding of distributed computing, serialization and Java interfaces.

Prerequisites:

Attend the Java: Under the Hood - Advanced Java Programming course, or possess the equivalent knowledge of Java programming, serialization, RMI and JDBC.

Objectives:

- Become familiar with the various Java technologies of J2EE.
 - Understand what CORBA provides.
 - Develop a CORBA application.
 - Understand the architecture of Java Servlets.
 - Develop a Servlet application.
 - Learn how to deploy a Servlet to a Web server.
 - Connect to a database from within a Servlet.
 - Understand the goals of JavaServer Pages.
 - Learn how to write scriptlets using JavaServer Pages.
 - Learn how to use JavaBeans from within a JavaServer Page.
 - Understand the goals of XML.
 - Understand how XML compares to HTML and SGML.
 - Write an XML document.
 - Learn the various elements of the Java API for XML Processing (JAXP).
 - Use the Simple API for XML (SAX).
 - Use the Document Object Model (DOM).
-

Weekly Schedule

The following is a tentative schedule for the pacing of the course. The actual flow of the course may vary.

Day One

- Chapter 1: An Overview of J2EE Technologies
- Chapter 2: Servlets

Day Two

- Chapter 2: Servlets (continued)
- Chapter 3: JavaServer Pages

Day Three

- Chapter 3: JavaServerPages

Day Four

- Chapter 4: CORBA

Day Five

- Chapter 5: XML
- Chapter 6: JAXP

Table of Contents

Course Description	iii
Objectives:	iv
Weekly Schedule	v
 Chapter 1 J2EE Overview	1
The J2EE Specification	2
The J2EE Technologies	3
The EJB Specification	4
Java Servlets	5
The Servlet API	6
JDBC	7
JDBC Drivers	8
JavaServer Pages	9
JavaServer Pages and Servlets	10
Overview of XML	11
What XML Looks Like	12
JNDI	13
Java Transactions	14
Java Message Service	15
Java IDL	16
RMI-IIOP	17
Java Mail	18
Connectors	19
Web Services	20
 Chapter 2 An Introduction to Servlets	23

An Overview of Servlets.....	24
Advantages of Servlets	26
Lifecycle of a Servlet	28
The Java Servlet API	30
The service() Method	32
The HttpServlet Class.....	33
The "Hello, World" Servlet.....	34
Compiling the Servlet.....	36
The Directory Structure of a Web Application	38
The Web Application Deployment Descriptor	40
Deploying a Servlet on Tomcat	42
Step 1: Start the Tomcat Manager.....	43
Step 2: Enter the path to web.xml	44
Step 3: Access the servlet.....	46
Lab 1: The RequestInfo Servlet.....	48
 Chapter 3 Writing Servlets	53
HTTP	54
Requests	56
Responses	58
HttpServletRequest	60
HttpServletResponse	62
Request Parameters	64
The Request Dispatcher	68
An Example of Including	70
Lab 1: The Servlet Controller.....	74
Object Scope.....	78
Request Scope	80
An Example of Request Scope	82
Using Internationalization	84
Application Scope	88
Session Scope	90

Cookies.....	92
The Cookie Class	94
An Example Using Cookies.....	96
The session Object	98
An Example of a Session	100
URL Rewriting	102
An Example of URL Rewriting	104
Lab 2: Using Resource Bundles	106
Lab 3 - The Shopping Cart.....	110
 Chapter 4 Using Servlets Effectively	113
Initializing Servlets	114
Destroying a Servlet	115
The Servlet Thread Models	116
The Single Thread Model	118
Cleanly Destroying a Servlet	120
Servlet Listeners	124
The Delegation Model	125
An Example of a Session Counter	126
Registering a Listener	128
Using Filters	130
The Filter Interface	131
An Example of a Filter Class.....	132
Deploying a Filter	134
Lab 1: Writing a Servlet Listener	136
 Chapter 5 An Introduction to JavaServer Pages	139
Overview of JavaServer Pages.....	140
JSP or Servlets?	141
JavaServer Pages and Servlets	142
The Lifecycle of a JSP Page	144
A Simple JavaServer Page	146

Deploying a JSP Page	148
JavaServer Page Tags	150
Declarations	152
The Generated Servlet Class.....	154
Expressions	156
Implicit Objects	158
The request Object	160
The Methods of the Request Object	161
Parameters	162
Scriptlets	166
A JSP Using a Data Access Object	168
The include Tag	172
An Example Using include	174
Directives	176
The Page Directive	177
JSP Actions	178
Lab 1: Working with JSPs.....	182

Chapter 6 Using JavaBeans within JSP	187
The MVC Model Revisited	188
JavaBeans in JSPs.....	189
Scope Parameter of <jsp:useBean>	190
Setting Bean Properties	192
Getting Bean Properties	193
JavaBeans and Form Properties	194
Validating Form Data	198
Lab 1: The ResourceBean	202
Scope	204
Request Scope	206
Application Scope	208
Session Scope	210
The Implicit Session Object	212

Beans with Session Scope	214
Lab 2: Create the ShoppingCart View	216
Error Pages	218
Lab 3: An Error Page for BEST	220
 Chapter 7 Custom JSP Tags	223
Overview of Custom Tags	224
Steps to Create Your Own Tags	226
The javax.servlet.jsp.tagext Package	227
The Lifecycle of a Tag Handler	228
Step 1: Write the Tag Handler Class	230
The Tag Library Descriptor	232
Step 2: Create a TLD File	234
Step 3: Use the tags in a JSP	236
Step 4: Deploy the files	238
Body Tags	240
The Lifecycle of a Body Tag	241
An Example of a Body Tag	242
Modifying the TLD	246
The TryCatchFinally Interface	248
Lab 1: Writing a Custom JSP Tag	250
 Chapter 8 Java and CORBA	253
The Object Management Group	254
Overview of CORBA	255
Java IDL	256
The ORB Architecture	258
Services of the ORB	260
The Portable Object Adapter (POA)	262
The Interface Definition Language	264
The IDL to Java Mapping	265
Writing an IDL Interface	268

Adding Attributes	269
Adding Methods	270
Out Parameters	271
Compiling the IDL	272
Lab 1: Writing an IDL interface	274
Defining Exceptions	276
Lab 2: Defining Exceptions	278
Implementing CORBA on the Server	280
Step 1: Implement the IDL Interfaces	282
Step 2: Initialize the ORB and the POA	284
Step 3: Create the POA	286
Step 4: Generate an IOR for the servant	288
Step 5: Run the ORB	290
Lab 3: Writing the Server Application	292
The Client Application	294
Lab 4: Writing the Client	296
The Naming Service	298
COS Naming	300
The NamingContext Interface	301
Using the Naming Service	302
Running the Name Service	305
Lab 5: Using the Naming Service	306
Lab 6: Returning a servant reference from a method	308
 Chapter 9 An Introduction to XML	311
XML Tags	312
The Root Element	313
Attributes	314
When to Use Attributes	315
Entity Elements	316
Well-formed vs. Valid	318
Document Type Definition	320

Creating a DTD	322
The <web-app> DTD	324
XML Namespaces	326
Lab 1: Designing an XML Document	328
Chapter 10 The Java API for XML Processing	331
The Java API for XML Processing	332
Overview of SAX	334
Overview of DOM	336
Creating a SAX Parser	338
A Simple Parser	340
Parsing an XML Document Using SAX	342
Handlers	344
Content Handlers	345
Error Handlers	350
An Example of an Error	352
Validating XML Using a DTD	354
Creating a DOM Parser	358
Parsing an XML Document Using DOM	360
Choosing a DOM Parser	362
Understanding DOM Nodes	364
The Document Interface	365
Creating an XML Document Using DOM	366
An Overview of XSLT	368
Output a DOM Document to a File	370
Lab 1: Generating an XML Document	372

This chapter contains an overview of each of the technologies in the Java 2 Platform, Enterprise Edition, known as J2EE. Topics discussed in this chapter include:

- Overview of J2EE
- Enterprise JavaBeans
- Servlets
- JavaServer Pages
- XML
- CORBA and RMI-IIOP
- Web Services

The J2EE Specification

J2EE stands for Java 2 Platform, Enterprise Edition, and it represents a collection of Java technologies involved in multi-tier enterprise development. The purpose of J2EE is to provide a uniform platform for developers to create multi-tier applications that are not tied to a specific vendor or tool.

- Some of the benefits of J2EE include:
- Platform-independent server-side development.
- Define a set of services that independent vendors provide.
- Developers target the J2EE specification, not vendor-specific features.

J2EE allows for developers to focus on the business logic of a distributed business application. Server vendors benefit from J2EE because it provides them with a blueprint for developing an application server that developers can rely on for specific services.

The J2EE Technologies

The J2EE specification includes the following technologies:

- Enterprise JavaBeans
- Java servlets
- JDBC - Java Database Connectivity
- Java Server Pages
- XML
- JNDI - Java Naming and Directory Interface
- JTA - Java Transaction API
- JTS - Java Transaction Service
- JMS - Java Messaging Service
- Java IDL
- JavaMail
- RMI-IIOP
- Connectors
- Web Services (to be added in a later version)

Software vendors that want to be J2EE-compliant need to provide the technologies listed above. Sun Microsystems provides a compatibility test for J2EE compliant products.

The EJB Specification

Enterprise JavaBeans is a specification. It defines a set of services that are provided by an EJB server, and also defines the interface between the server and the components.

The following is a definition of the EJB architecture, taken from the Sun Microsystems Enterprise JavaBeans Specification, v2.0:

“Enterprise JavaBeans is an architecture for component-based distributed computing. Enterprise beans are components of distributed transaction-oriented enterprise applications.”

The Features of Enterprise JavaBeans

The purpose of EJBs is to allow for rapid application development on the server-side of multi-tier business applications. Some of the features of the Enterprise JavaBeans technology include:

- Java server-side development of components.
- The developer focuses on the business logic, not the server framework.
- The deployer takes care of customizing the bean in the server.
- The server vendor provides the server framework.
- The EJB components reside in a container that provides services like transactions, security, persistence and naming.
- EJB components are scalable and portable.

Java Servlets

Servlets are Java applications that run in a Web server. They provide Web developers with the ability to use Java applications to create dynamic and robust Web pages. Servlets are notified of a request from a Web browser and provide a response. The Web server maintains the instances of the servlet.

Servlets have several advantages over CGI scripts, including performance and reliability on the server. When a servlet receives a response from a client, a method is invoked on the servlet object, which runs in a separate thread on the server. When a CGI script is executed, it runs in a separate process. If a CGI script goes awry, it can crash the entire server, not just the Web server application.

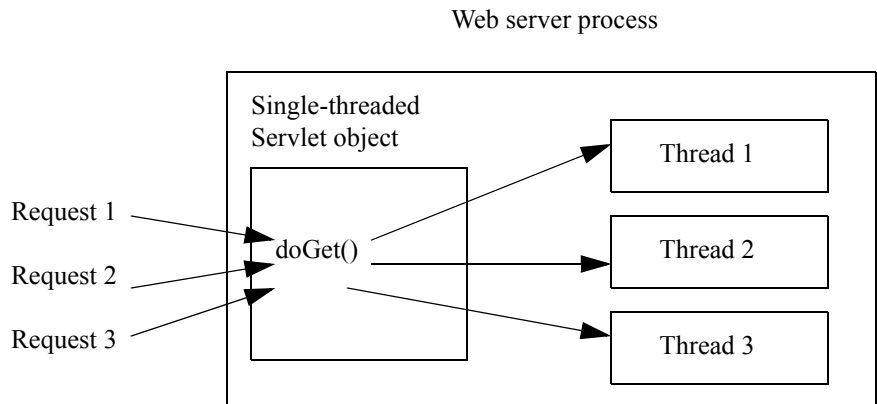


FIGURE 1. *Servlets can run in a single JVM process.*

The Servlet API

The Servlet API contains the various classes and interfaces required to develop servlets. There are two packages in the Servlet API: `javax.servlet` and `javax.servlet.http`.

A servlet is a class that extends either the `GenericServlet` or `HttpServlet`. When a client makes a request, either the `doGet()` or `doPost()` method of the servlet is invoked. GET and POST are protocols of HTTP. Most requests from Web clients are GET requests, when a user visits a particular Web page.

JDBC

JDBC stands for Java Database Connectivity. JDBC is an API that provides an interface for connecting to a database from within Java code. The interface contains classes to represent SQL statements and result sets from a query.

JDBC 2.0 is the latest version of the JDBC API, and it comes in two parts:

- JDBC 2.0 Core API - the classes found in the `java.sql` package.
- JDBC 2.0 Standard Extension API - the classes found in the `javax.sql` package.

Note the `javax.sql` package is not a part of the JDK and requires a separate download.

The goal of JDBC is provide a high-level access to databases without having to write all the low-level code often necessary with ODBC. The advantage of using JDBC is that it is database independent. The code for accessing the database does not depend on which database vendor you are using, or where the database server is located. A Java program that accesses a database can easily be moved from one database to another.

JDBC plays a major role in creating thin clients for two and three-tier architecture. JDBC is a thin API, and since it is Java, it can be deployed on any platform.

JDBC Drivers

JDBC uses a driver provided by the database vendor to connect to the database and perform SQL statements. The JDBC API provides a database-independent technique for performing these tasks.

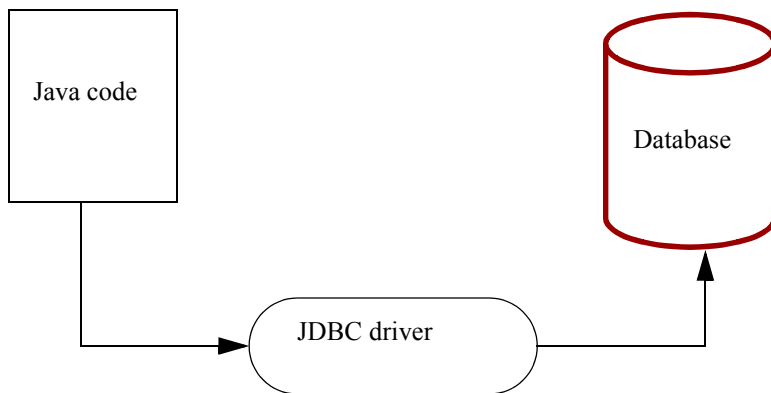


FIGURE 2. *JDBC allows Java code to access a database.*

JavaServer Pages

JavaServer Pages are HTML documents that can contain code from the Java programming language. This allows the HTML document to go from static to dynamic, since a JSP is not simply returned to the client. Instead, the page is "executed" in a Java object called a servlet.

Like ASP and server-side JavaScript, JSP requires a compatible Web server. However, JavaServer Pages have a considerable advantage over other server-side development options like CGI and ASP:

- **Java code:** JavaServer Pages use the Java programming language.
- **Portable:** Like Java, JavaServer Pages are platform-independent and do not rely on any native code to execute. All you need is a Web server that supports JSP.
- **Easy to learn:** The design of JavaServer Pages makes them usable by Web developers who have minimal programming skills but are well-versed in the areas of Web site development.
- **Powerful:** JavaServer Pages have the full Java API and Java extensions at their disposal, making them powerful programming entities with the ability to perform tasks like database access, multi-threading, RMI, CORBA, and much more.
- **Servlets:** JavaServer Pages are generated into servlets, Java objects that execute within a container of the Web server.

JavaServer Pages and Servlets

You may be asking yourself how an HTML document with Java code scattered throughout it gets executed on a Web server. The answer is simple: the Web server generates a servlet using the syntax of your JavaServer Page.

In other words, developing a JavaServer Page is really just a different technique for creating a Java servlet. When a JavaServer Page is initially accessed by a client, the JSP engine writes, compiles, and instantiates a servlet object that contains the Java code of the JSP.

The advantage to JavaServer Pages over servlets is that the developer of the JSP does not need to be a Java programmer. Creating JavaServer Pages only requires an introductory understanding of object-oriented programming and some familiarity with the syntax of Java.

The design of JSP is basically the opposite concept of the design of servlets:

- A Servlet is Java code that can be used to generate HTML documents.
- JavaServer Pages are HTML documents that can contain Java code.

Overview of XML

XML stands for eXtensible Markup Language. The goal of XML is to provide a mechanism for describing data in a format that is free of any content. This allows the data to be represented in any number of specific formats by separating the content of the documentation with how it is viewed.

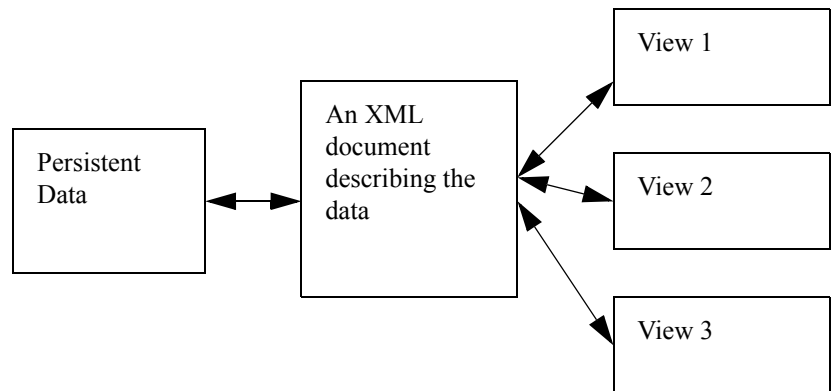


FIGURE 3. *XML separates data from its view.*

What XML Looks Like

XML looks a lot like HTML, the markup language that Web browsers use to display data. XML consists of a series of starting and ending tags, which are nested within each other.

To demonstrate what XML looks like, the following XML document is a descriptor of an Enterprise JavaBean, a software component used for middle-ware application development:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>Employee</ejb-name>
      <home>demos.payroll.EmployeeHome</home>
      <remote>demos.payroll.Employee</remote>
      <ejb-class>demos.payroll.EmployeeBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>demos.payroll.EmployeePK</prim-key-class>
      <reentrant>False</reentrant>

      <cmp-field>
        <field-name>hoursWorked</field-name>
      </cmp-field>

      <cmp-field>
        <field-name>name</field-name>
      </cmp-field>

      <cmp-field>
        <field-name>salary</field-name>
      </cmp-field>

      <cmp-field>
        <field-name>id</field-name>
      </cmp-field>

    </entity>
  </enterprise-beans>
  <assembly-descriptor></assembly-descriptor>
</ejb-jar>
```

FIGURE 4. *An EJB deployment descriptor in XML.*

JNDI

JNDI stands for Java Naming and Directory Interface and is an API that provides a protocol-independent interface for using the various naming and directory services available.

For example, you might be using an ORB for exporting and finding remote objects. Your Java application can use the classes and methods of the JNDI to obtain the objects and services provided by the ORB. If you decide to change protocols and use a different architecture besides CORBA, your Java code will not need to be modified, since it used JNDI and did not rely on any protocol-specific naming service.

The Java Naming and Directory Interface is compatible with the following naming and directory services:

- LDAP
- NIS (Network Information Service)
- COS - the naming service used by CORBA
- Any native file system
- Novell File Systems
- WebLogic's Tengah naming service
- PNDIS - used by smart cards
- Any J2EE-compliant application server

Java Transactions

The Java Transaction API is a set of interfaces for accessing a transaction manager. It consists of two packages: `javax.transaction` and `javax.transaction.xa`. The interfaces in the Java Transaction API allow for performing operations on a transaction and the transaction's boundaries.

The Java Transaction Service defines the implementation of a transaction manager that supports the Java Transaction API, which is found in the `javax.jts.TransactionService` interface. If you ever need to write a transaction service, this could be done by implementing the various interfaces of the Java Transaction Service.

The Java Transaction Service also defines the Java mapping of the OMG Object Transaction Service. This mapping can be found in the `org.omg.CosTransactions` and `org.omg.CosTSPortability` packages.

Java Message Service

The Java Message Service API provides for a platform-independent way to write message-based enterprise applications. The goal of this API is to simplify the development of applications that need to use messaging. The Java Message Service API is defined in the `javax.jms` package.

The `javax.jms.Message` interface defines the basic behaviors of a message. Messages consist of a header, which contains message routing information, and a body, which contains the application data. A message also has properties that can be application-specific.

However, there are too many message models used today for JMS to encompass all the existing message technologies. Therefore, the goals of JMS are:

- Provide a single messaging API.
- Provide an API that will work with existing messaging technologies.
- As with Java, create a platform-independent, portable messaging applications.
- Allow for Java objects to be a part of a message's body.
- Support messages that contain XML objects.

Java IDL

CORBA uses the Interface Definition Language (IDL) for creating language independent interfaces for distributed objects. The Java IDL is a mapping between IDL and Java. You use the `idltojava` compiler, freely downloadable from Sun's Web site, to create the necessary stubs and skeletons required by the ORB.

The Java IDL is not an API, rather a mapping of IDL data types to Java data types. The necessary classes and APIs for using CORBA are found in the JDK.

RMI-IIOP

RMI stands for Remote Method Invocation, which allows for distributing objects in a Java-to-Java solution. It is not as extensive and powerful as CORBA, but it is simpler to implement when Java is being used on both the client and server.

IIOP stands for Internet Inter-ORB Protocol, which provides for a standard interface between ORBs.

With RMI and IIOP combined, a Java interface can be implemented in a different language like C++ or other CORBA supported languages. Unlike CORBA, however, no separate IDL interface is required.

Developers use the RMI over IIOP IDL compiler to generate the necessary stubs and skeletons used by the ORB.

Java Mail

The JavaMail API consists of a collection of classes defined in the `javax.mail` package. The classes can be used to create an E-mail application in Java that can send and read mail.

The implementations included in the JavaMail API are for the IMAP and SMTP protocols. Private service vendors provide other implementations like POP3.

Connectors

The connector architecture defines a standard for connecting J2EE components with existing Enterprise Information Systems (EIS). The architecture defines a mechanism for the following services:

- Connection management
- Transaction management
- Security

In addition, the connector architecture defines the Common Client Interface (CCI), which provides an EIS-independent API for accessing the native functions of an existing EIS. This technology is important because many EIS systems do not represent relational databases.

The connector architecture also defines how EIS vendors can provide resource adapters, which are drivers that Java applications can use to connect to the EIS. Resource adapters are to connectors what database drivers are to JDBC.

Web Services

Web services is a term used to describe a mechanism for the creation and deployment of B2B (business-to-business) applications. The concept of B2B is nothing new: businesses have been striving to communicate with other businesses through software applications since the advent of the network.

However, many previous attempts at B2B have come up short, and we still today do not have a full-fledged solution for applications to easily communicate with each other. The barrier lies in the network. With the growth and popularity of the Internet, the Web has changed the way computers are networked together. It is conceivable to visualize the Web as one giant network that any computer can connect to. HTTP, the HyperText Transfer Protocol, and HTML (HyperText Markup Language) have together provided a way for two computers to transfer data between each other using a format and protocol common to both computers, no matter what platform or operating system is being used.

The promise of Web services is to extend the concept of HTML to more than just sending documents and nice-looking Web pages back and forth between networked computers. The objective is to create an environment where applications can communicate effectively with other applications by taking advantage of the services that a particular application provides.

Web services consist of the following process:

- An application is written not to solve a specific problem or perform a specific task, but instead an application is written to provide specific services.
- An application that provides services can publish their services in a directory.
- Other applications that want to use these Web services can see which services are provided by searching for them in the directory.
- Once these services are discovered, they can be used by another application.

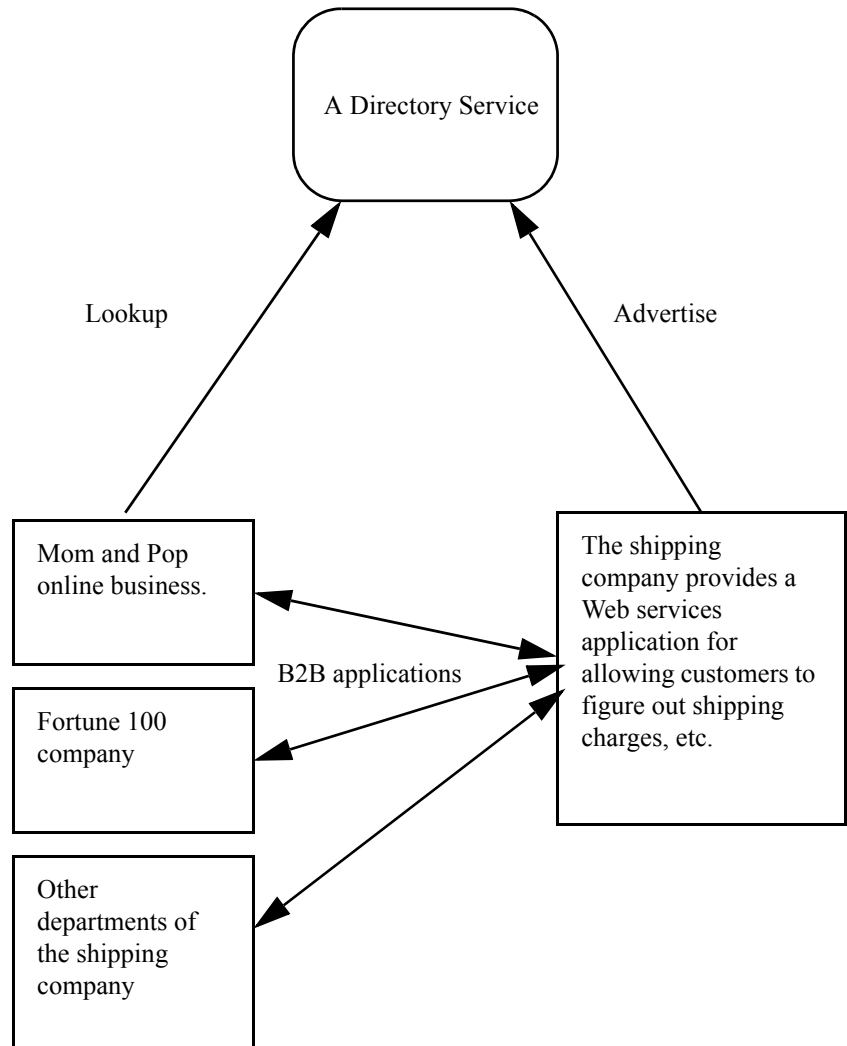


FIGURE 5. *An overview of Web services.*



An Introduction to Servlets

In this chapter, we will discuss Java servlets, including their lifecycle and the Java Servlet API. We will also write and deploy a simple “Hello, World” servlet. Topics discussed in this chapter include:

- Overview of Servlets
- Advantage of Servlets
- Lifecycle of a Servlet
- The Java Servlet API
- The `service()` Method
- The `HttpServlet` class
- The “Hello, World” Servlet
- The Directory Structure of a Web Application
- The Web Application Deployment Descriptor

An Overview of Servlets

According to Sun Microsystems, a servlet is defined as follows:

A *servlet* is a Java programming language class used to extend the capabilities of servers that host applications accessed via a request-response programming model.

More specifically, a servlet is a Java object that resides in a Web server. When they say “request-response programming model”, they are referring to HTTP, the Hyper Text Transfer Protocol. With HTTP and servlets, the following sequence of events occurs:

1. An HTTP request is sent to a URL that represents a servlet.
2. The Web server recognizes that the request is for a servlet and passes the request to the *servlet container*.

NOTE

The term *servlet container* refers to that portion of the Web server that provides support for servlets.

3. The servlet container invokes the `service()` method on the appropriate servlet object.
4. Within the `service()` method, a servlet can generate a response and/or perform just about anything, since the servlet has the entire Java API at its disposal.
5. The servlet response is passed from the container back to the Web server. The Web server takes the response and passes it back to the client who sent the request.

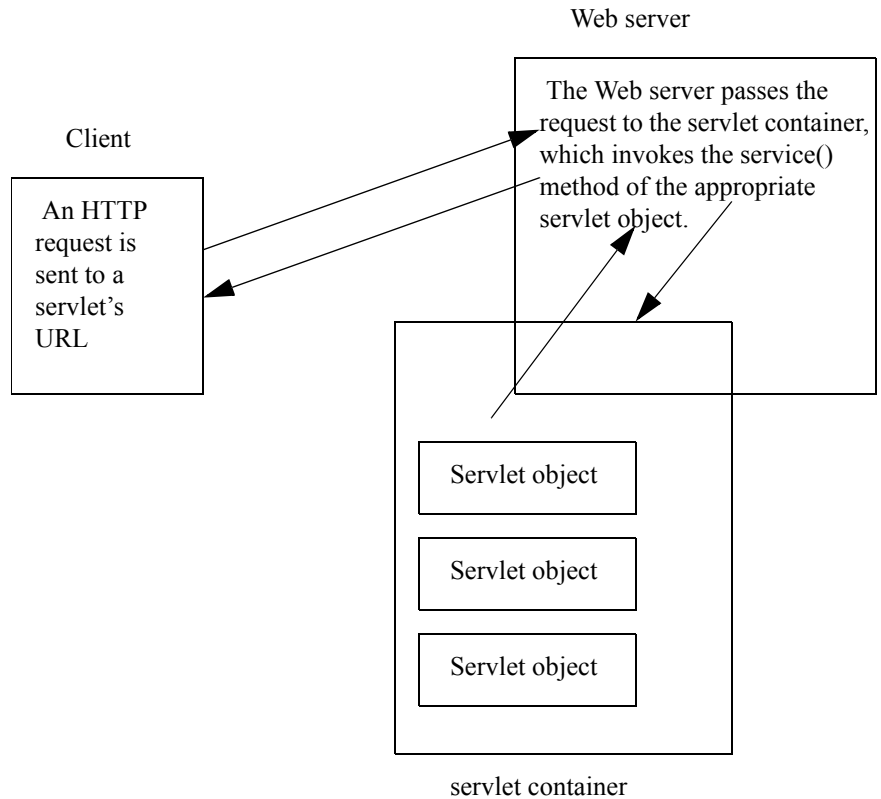


FIGURE 1. *An overview of the request-response mechanism of servlets.*

Advantages of Servlets

A Java servlet is an object that resides in a Web server, much like an applet resides in a Web browser. Unlike applets, however, servlets do not have a GUI interface. Some of the advantages of using servlets include:

- Servlets run entirely in the Web server, so there are no plug-ins or requirements placed on the client's web browser.
- Since they are written in Java, servlets are portable. This allows them to be moved from one Web server to the next.
- Web application developers can write programs that focus on the business logic and not the Web server being used.
- Another advantage of servlets is that they have access to the entire Java API. This means servlets can use JNI to invoke native methods, JDBC to connect to a database, JMS to use a message service, JavaMail to send email messages, JTS to use transactions, and on and on.
- Security: servlets inherit all the built-in security features of the Java programming language.
- Performance: even though servlets are interpreted by a JVM, they can have performance advantages because each client request executes within a thread, not a separate process, as illustrated in Figure 2.

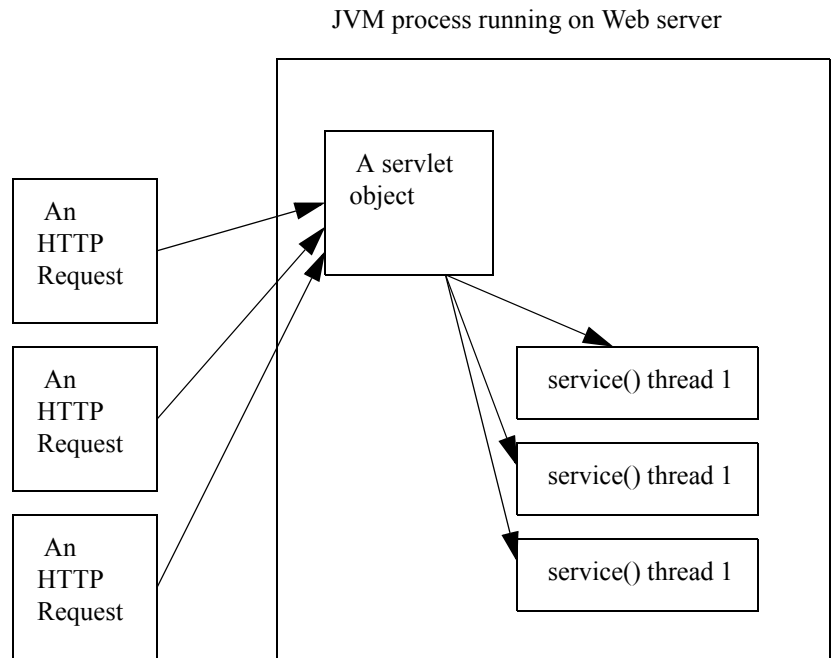


FIGURE 2. *The `service()` method executes in a new thread.*

Lifecycle of a Servlet

A servlet resides in a servlet container. The container controls the lifecycle of the servlet by instantiating the servlet and invoking methods on it.

The lifecycle of a servlet is as follows:

1. The container, if necessary, creates an instance of the servlet class after loading the necessary class or classes.
2. The container invokes the `init()` method on the servlet.
3. When a request is made for the servlet, the container invokes the `service()` method of the servlet, passing in the request and response objects.
4. A container invokes the `destroy()` method on the servlet before removing the servlet from the container.

NOTE

The `init()`, `service()` and `destroy()` methods are invoked by the container at the appropriate time in the lifecycle of a servlet. How does the container know that your servlet has written these methods? Because servlets are required to implement the `javax.servlet.Servlet` interface, and this interface contains each of these three methods.

Notes

The Java Servlet API

Servlet technology is dictated by the Java Servlet Specification, managed by the Java Community Process. The specification has undergone various changes over the years, and servlets has become a fairly mature and established technology.

The latest version of the Java Servlet Specification is 2.4, which contains only minor changes from the 2.3 version.

The Java Servlet API contains the various classes and interfaces required to develop servlets. There are two packages in the Servlet API for writing servlets:

- `javax.servlet`
- `javax.servlet.http`

There are two types of servlets that can be written:

- **Generic servlet:** a generic servlet can be written by extending the `javax.servlet.GenericServlet` class. Generic servlets do not use any specific transfer protocol. Specifically, they can be used in environments that do not use HTTP.
- **HTTP servlet:** an HTTP servlet can be written by extending the `javax.servlet.http.HttpServlet` class. In most situations, this is the class you will subclass, since HTTP is the common protocol used today in web applications.

The following diagram is an overview of the classes and interfaces found in the Servlet API.

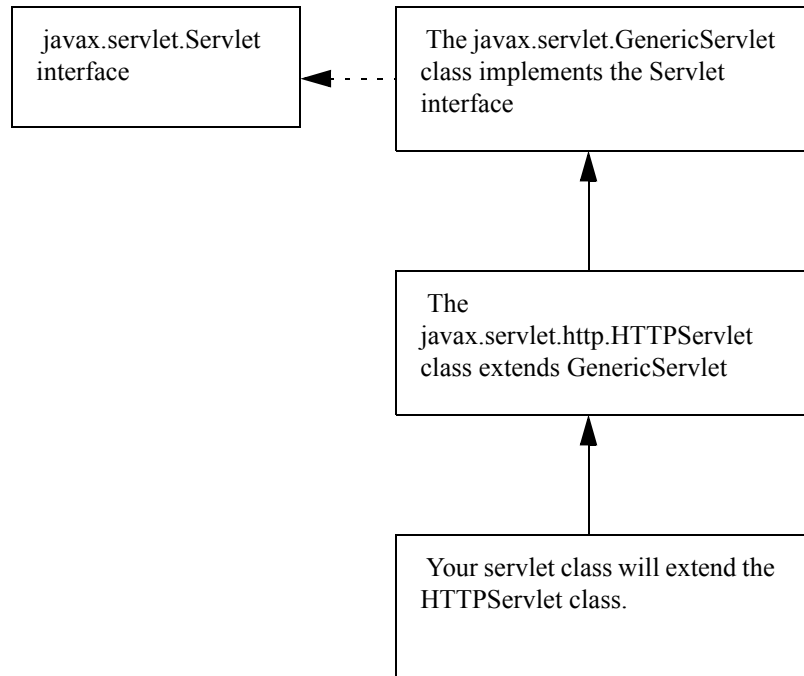


FIGURE 3. *A servlet class typically extends the HttpServlet class.*

NOTE

The `javax.servlet.Servlet` interface must be implemented by all servlets. The `GenericServlet` class implements all of the methods of `Servlet` except for the `service()` method, which must be implemented by any subclass.

NOTE

The `javax.servlet.http.HttpServlet` class extends `GenericServlet` and implements the `service()` method. Subclasses of `HttpServlet` will override the `doGet()`, `doPost()` and similar methods, as discussed next.

The service() Method

Clients access a servlet using the HyperText Transfer Protocol (HTTP). A request is made using a URL, which causes the service() method of the servlet to be invoked. The service() method in the HttpServlet class determines the HTTP method of the request and invokes one of the doXXX() methods of the HttpServlet class. contains methods that match the six HTTP protocols GET, POST, PUT, HEAD, OPTIONS and DELETE.

For example, when you enter a URL in your Web browser, here is what happens:

- The Web browser creates a GET request in the HTTP format.
- The GET request is transmitted over the Internet to the appropriate Web server.
- The Web server determines that the GET request is for a servlet and passes the request to the servlet container of the Web server.
- The servlet container determines which servlet object the request is meant for, and invokes the service() method on that servlet object in a new thread.
- If the servlet object is of type HttpServlet, the service() method in HttpServlet is invoked. This service() method determines that the request is a GET request and invokes the doGet() method.
- Your servlet class that extended HttpServlet will override doGet(). Therefore, your doGet() method is invoked on your servlet object.
- Your servlet generates an HTTP response, which is sent back to the browser that made the request.

The HttpServlet Class

The following is a list of some of the more frequently used methods in the `HttpServlet` class and an explanation of when the methods are invoked.

```
protected void service(HttpServletRequest req,
                        HttpServletResponse res)
                        throws ServletException, IOException;
```

The `service()` method is invoked when a client accesses the servlet. The `service()` method dispatches the request and response to the corresponding `doXXX()` method of the servlet. This method should not be overridden.

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse res)
                     throws ServletException, IOException;
```

The `doGet()` method is invoked from a GET request from the client. The `doGet()` method should be overridden for servlets that need to handle GET requests. The `doGet()` method is useful for returning dynamic HTML, but not for retrieving large amounts of information from the client. It should be safe for clients to invoke the `doGet()` multiple times without side-effects to the client. For example, a database query could be safely repeated; ordering a product would not be.

```
protected void doPost(HttpServletRequest req,
                      HttpServletResponse res)
                      throws ServletException, IOException;
```

The `doPost()` method is invoked from a POST request from the client. The `doPost()` method should be overridden for servlets that need to handle POST requests. The behavior of `doPost()` is typically not repeated and therefore does not need to be client-safe. For example, purchasing an item or updating a database would be done with POST.

The "Hello, World" Servlet

The HelloWorldServlet class in Figure 4 demonstrates an HTTP servlet. Notice it subclasses HttpServlet and overrides both the doGet() and doPost() methods. This particular servlet sends back a simple "Hello" greeting to the client.

Notice the following about the HelloWorldServlet:

- It extends HttpServlet, which all HTTP servlets must do.
- The servlet uses the getWriter() method of the HttpServletResponse object to obtain a reference to the output stream of the client.
- Any data sent to the PrintWriter returned from getWriter() is read from the client's Web browser in the response. Notice the HelloWorldServlet sends back an HTML document created dynamically in the doGet() method.
- The doPost() method does nothing but simply invoke doGet(). We did not need to define doPost(), but it is commonly done when creating a servlet. This way, if a client happens to reach our servlet via a GET request or a POST request, the servlet will still respond accordingly. It is not likely that a client will access the HelloWorldServlet using a POST, but our servlet can handle POST requests just in case.
- The init() method is overridden to illustrate when it is invoked. It also initializes the String field of the servlet.


```
1  import javax.servlet.*;
2  import javax.servlet.http.*;
3  import java.io.*;
4
5  public class HelloWorldServlet extends HttpServlet
6  {
7      private String message;
8
9      public void init(ServletConfig config)
10         throws ServletException
11     {
12         super.init(config);
13         System.out.println("Initializing a HelloWorldServlet");
14         message = "Hello, Servlets!";
15     }
16
17     public void doGet(HttpServletRequest req,
18                       HttpServletResponse res)
19         throws ServletException, IOException
20     {
21         System.out.println("A GET request has occurred.");
22         res.setContentType("text/html");
23         PrintWriter out = res.getWriter();
24
25         out.println("<HTML>");
26         out.println("<HEAD><TITLE>HelloWorld</TITLE></HEAD>");
27         out.println("<BODY>");
28
29         out.println("<H1>" + message + "</H1>");
30         out.println("<P>You are at " + req.getRemoteHost() + "</
31 P>");
32
33         out.println("</BODY></HTML>");
34         out.close();
35     }
36
37     public void destroy()
38     {
39         System.out.println("Destroying a HelloWorldServlet");
40     }
41
42     public String getServletInfo()
43     {
44         return "This is a simple servlet that displays a greeting";
45     }
46 }
```

FIGURE 4. *The "Hello, World" Servlet.*

Compiling the Servlet

Before you can compile classes that use the Java Servlet API, you need to include the appropriate JAR files in your CLASSPATH environment variable. The implementation of the Servlet API for Tomcat is found in the following JAR file:

- %CATALINA_HOME%\common\lib\servlet.jar

For example, if you installed Tomcat in the c:\Tomcat 4.1 folder, then you need to set your CLASSPATH to be:

- C:\Tomcat 4.1\common\lib\servlet.jar

You can either set your CLASSPATH in the Windows Control Panel in the System settings, or you can set the CLASSPATH at the command prompt using the command:

```
set CLASSPATH=%CATALINA_HOME%\common\lib\servlet.jar;.;
```

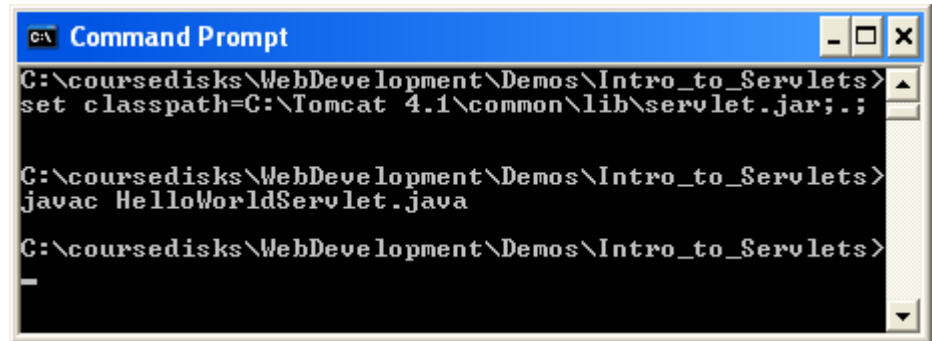
NOTE

The "." in the CLASSPATH represents the current directory, which is often needed in your CLASSPATH environment variable.

- Once your CLASSPATH is set to contain the servlet.jar file, you can compile the HelloWorldServlet using javac:

```
javac HelloWorldServlet.java
```

Figure 5 demonstrates setting the CLASSPATH and compiling the HelloWorldServlet.



```
C:\coursedisks\WebDevelopment\Demos\Intro_to_Servlets>
set classpath=C:\Tomcat 4.1\common\lib\servlet.jar;;

C:\coursedisks\WebDevelopment\Demos\Intro_to_Servlets>
javac HelloWorldServlet.java

C:\coursedisks\WebDevelopment\Demos\Intro_to_Servlets>
_
```

FIGURE 5. *Compiling the HelloWorldServlet.*

The Directory Structure of a Web Application

Java Web applications are deployed in a specific directory structure. Some tools create this directory structure for you when you package your Web applications into WAR files.

- When using Tomcat, it is a good idea to keep save your files in the same directory structure required of the Web server. This allows for simpler deployment.

The following directories are involved in a Java Web application:

- `<root>/` - the root directory is for HTML pages, JSP pages, and other files that need to be directly available to the client's browser, like images and graphics.
- `<root>/WEB-INF/` - the WEB-INF directory is for any .class files that your Web application needs. The .class files do not go directly in WEB-INF, but in one of its subdirectories (listed next).

NOTE

A Java Web application uses a Web Application Deployment Descriptor to define the components of the application. This deployment descriptor is always in a file named "web.xml" which must appear in the WEB-INF directory.

- `<root>/WEB-INF/classes` - the .class files that do not appear in a JAR are stored in the `/WEB-INF/classes` directory.

NOTE

Classes in packages must appear in their proper directory structure as well. If `HelloWorldServlet` is in the "demo" package, then the path to its bytecode file needs to be:

```
/WEB-INF/classes/demo/HelloWorldServlet.class
```

- `<root>/WEB-INF/lib` - this directory is for JAR files. Place any JAR files that need to be found by your Web application in this folder.

NOTE

If you have JAR files that need to be available to all your Web applications, these can be placed in the following directory:

`%CATALINA_HOME%/shared/lib`

NOTE

If you have JAR files that need to be used by both your Web applications and also by Tomcat (most often JDBC drivers), then place these JAR files in the following directory:

`%CATALINA_HOME%/common/lib`

The Web Application Deployment Descriptor

A Java Web application has an XML file associated with it known as the Web Application Deployment Descriptor. The name and path of this file is:

```
/WEB-INF/web.xml
```

The web.xml file contains important information about your Web application that the server needs to know. For example, if your Web application contains a servlet, then an entry in web.xml of type `<servlet>` is required.

Figure 6 shows the Web Application Deployment Descriptor for the HelloWorldServlet defined in Figure 4 on page 35.

```
1  <!DOCTYPE web-app
2    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
3    "http://java.sun.com/dtd/web-app_2_3.dtd">
4
5  <web-app>
6    <display-name>Hello, World Servlet</display-name>
7    <description>
8      A simple servlet that displays Hello, World.
9    </description>
10
11   <servlet>
12     <servlet-name>helloworld</servlet-name>
13     <servlet-class>HelloWorldServlet</servlet-class>
14   </servlet>
15
16   <servlet-mapping>
17     <servlet-name>helloworld</servlet-name>
18     <url-pattern>/helloworld</url-pattern>
19   </servlet-mapping>
20
21   <session-config>
22     <session-timeout>30</session-timeout>
23   </session-config>
24 </web-app>
```

FIGURE 6. *The web.xml file for the HelloWorldServlet.*

Some comments about the deployment descriptor in Figure 6:

- The root element is `<web-app>`
- The `<display-name>` and `<description>` elements are for use by Web server tools and for readability, but do not affect any features of the Web application.
- The `<servlet>` element is for defining a servlet.
- The `<servlet-name>` element is for assigning a name to the servlet for use later in the deployment descriptor. It does not affect the URL of the servlet.
- The `<servlet-class>` element is the fully-qualified name of the servlet class.
- To make the servlet accessible to clients and other Web components, you need to define at least one mapping for the servlet using the `<servlet-mapping>` element.
- The `<servlet-mapping>` element has at least two subelements: `<servlet-name>`, which is the servlet that this mapping is being defined for, and `<url-pattern>`, which is the actual URL of the request for the servlet.
- The `<url-pattern>` for this servlet is `"/hello"`, so the URL to access this servlet will look similar to:

`http://localhost:8080/<context>/hello`

NOTE

The context of a Web application is determined when the application is deployed, not by any settings in the deployment descriptor.

Deploying a Servlet on Tomcat

There are several ways to deploy a servlet (or any Web application) onto Tomcat, including:

- Create a WAR file and copy it into the %CATALINA_HOME%/webapps/ folder. This is useful if you are given a WAR file and just need to deploy it.
- Copy your /<root>/ directory and all subdirectories into the %CATALINA_HOME%/webapps/<context> directory, where <context> is a directory you create. This directory becomes a part of the URL for the elements in the Web application.

NOTE

With the deployment techniques above, you may or may not have to restart Tomcat before the new Web application will be made available. This depends on the version of Tomcat you are running. If restarting the server is not an option for you, then use the following deployment technique:

- Use the Tomcat Manager to deploy the Web application. The Tomcat Manager is accessible from the URL (assuming you are accessing it from your localhost):

```
http://localhost:8080/manager/html/
```

Let's go through the steps of using the Tomcat Manager to deploy our HelloWorldServlet.

Step 1: Start the Tomcat Manager

Open your Web browser and enter the URL:

```
http://localhost:8080/manager/html/
```

You will need to enter your username and password, which are both “admin”. (See Figure 7.)



FIGURE 7. *Login to the Tomcat Manager*

Step 2: Enter the path to web.xml

Once you have the Tomcat Manager open, page down past the list of currently installed Web applications to the “Install” section.

NOTE

There are several ways to deploy a Web application using the Tomcat Manager. For example, if you have a WAR file, you can enter its path in the “WAR or Directory URL:” field. (Be sure to use proper URI syntax for files, which requires the file: protocol. For example, “file:/c:/myapps/hello.war”).

- Since we have written a Web Application Deployment Descriptor (see Figure 6 on page 40), we can enter its path in the “XML Configuration file URL:” field, as shown in Figure 8. For example:

```
file:c:/coursedisks/webdevelopment/demos/Intro_to_Servlets/  
HelloWorld/WEB-INF/web.xml
```

Be sure to use the appropriate path on your PC for this file.

- Enter the Context for the Web application. The context becomes a part of the URL for all components in the Web application. For this example, enter “/hello” as shown in Figure 8.
- Click the “Install” button when you have entered the location of your web.xml file to be deployed.

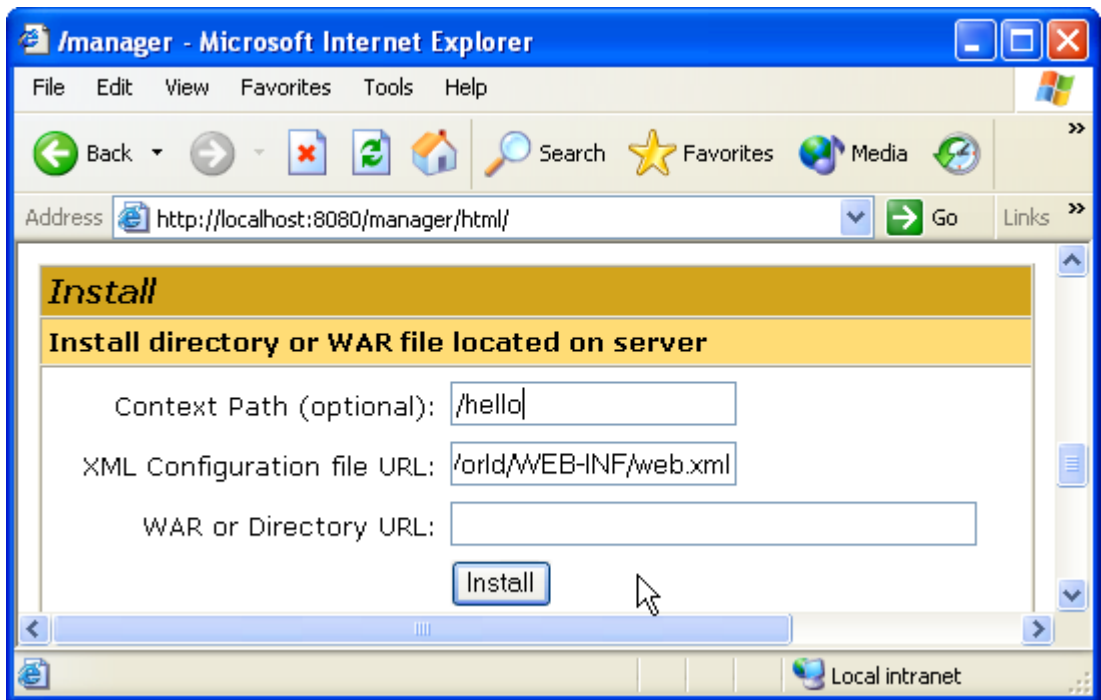


FIGURE 8. Enter the path to your web.xml file.

- If the Web application is successfully installed, you will see a message displayed at the top of Tomcat Manager similar to:

```
OK - Installed application from context file file:c:/  
coursedisks/webdevelopment/demos/Intro_to_Servlets/  
HelloWorld/WEB-INF/web.xml
```

Step 3: Access the servlet

The HelloWorldServlet is now ready for requests. To view the servlet, open your Web browser and enter the URL:

```
http://localhost:8080/hello/helloworld
```

The output should be similar to the one in Figure 9.

NOTE

The “/hello” portion of the URL is the context, which was determined at deployment time using the Tomcat Manager (see Figure 8).

NOTE

The “/helloworld” portion of the URL was defined in the Web Application Deployment Descriptor (web.xml) in Figure 6 on page 40.

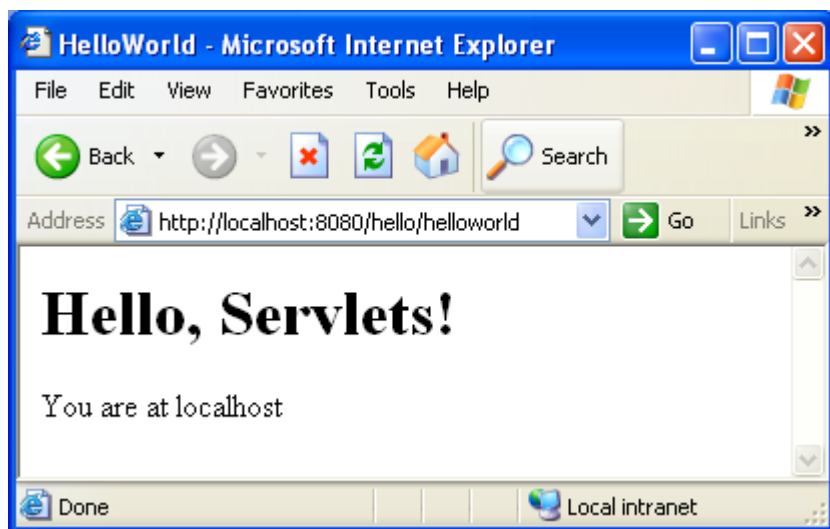


FIGURE 9. *The output of the HelloWorldServlet.*

NOTE

You may or may not see the output in the command prompt window where Tomcat is running. However, the following output is sent to System.out:

```
Initializing a HelloWorldServlet  
A GET request has occurred.  
Destroying a HelloWorldServlet
```

If you do not see any output in the Tomcat window, look for the output in a log file in the %CATALINA_HOME%/logs folder.

Lab 1: The RequestInfo Servlet

Objective: The purpose of this lab is to become familiar with writing and deploying a servlet. You will write a servlet that displays the information about a client's HTTP request using the various methods of the `HttpServletRequest` interface. You will also learn a different technique for deploying a servlet on Tomcat.

Perform the following tasks:

1. Create a new directory named `/lab1`.
2. Within the `lab1` directory, create a new directory named `/WEB-INF`.
3. Within `WEB-INF`, create a new directory named `/classes`.
4. Write a class named `RequestInfo` that extends `HttpServlet`, declaring the class in the "demos" package. Declare the `doGet()` method within this class.
5. Within `doGet()`, set the content type of the response to `text/html` and obtain a reference to the response's `PrintWriter`. Send back the `<html>` and `<body>` tags.
6. Use the `...` tags to create an unnumbered list. The `` tags consists of any number of list items created using the `...` tags. Send back to the client as part of the list the remote host and HTTP method. Use the `getRemoteHost()` and `getMethod()` methods of the `HttpServletRequest` interface.
7. Use the `getHeaderNames()` method of the `HttpServletRequest` to obtain an Enumeration of header names. You can use the `getHeader(String headerName)` method of `HttpServletRequest` to determine the value of each header. Iterate through the enumeration and print out each header along with its value as part of the unnumbered list.
8. Send back the `</body>` and `</html>` elements at the end of `doGet()` and close the `PrintWriter` stream.

9. Save the file `RequestInfo.java` in the folder `/lab1/WEB-INF/classes`.

10. Compile `RequestInfo.java` using the `-d` flag. For example:

```
javac -d . RequestInfo.java
```

11. You should notice a new directory named `/lab1/WEB-INF/classes/demos`, which is where `RequestInfo.class` is stored.

12. Using the `"web.xml"` file from the `HelloWorldServlet` example as a template, create a Web Application Deployment Descriptor for the `RequestInfo` servlet. You can give the servlet any name you like, but map the servlet's URL to be the value `"/request"`.

13. Save your `"web.xml"` file in the `/lab1/WEB-INF` directory.

14. Copy your `/lab1` directory into the `%CATALINA_HOME%/webapps` folder.

15. Open your Web browser enter the URL:

```
http://localhost:8080/lab1/request
```

16. If you get a 404 error saying the URL is not found, try restarting Tomcat.

What you should see: Your servlet should display the various information of the HTTP request, similar to Figure 10.

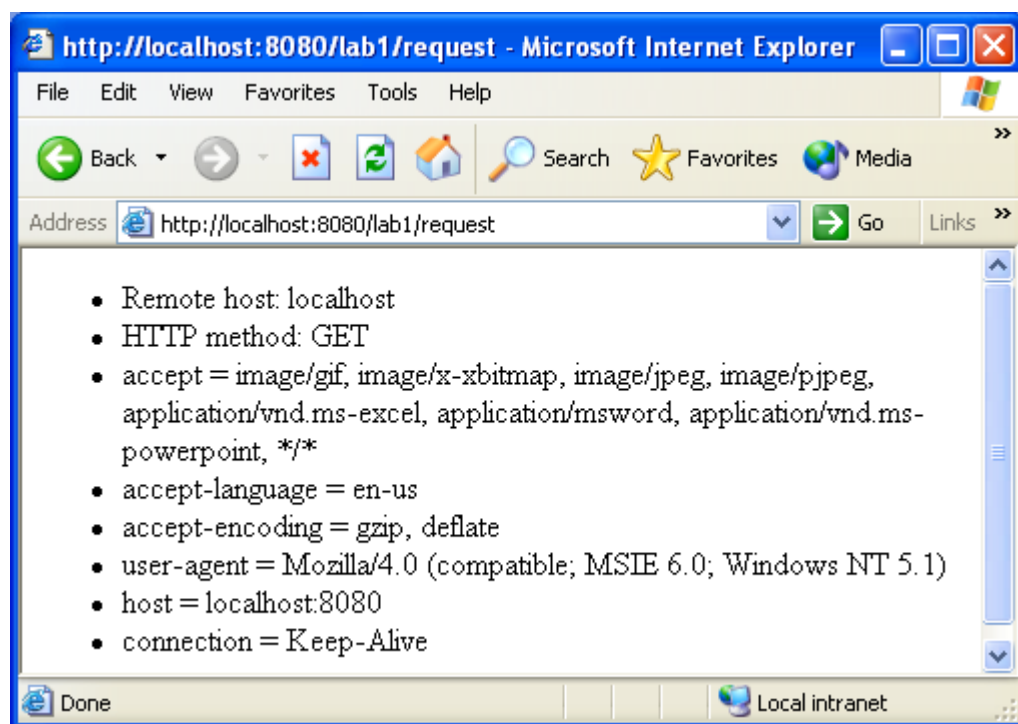


FIGURE 10. The output of the *RequestInfo* servlet.



In this chapter, we will discuss more of the details of how servlets work and what servlets can do. Topics discussed in this chapter include:

- An Overview of HTTP
- HTTP Requests and Responses
- The `HttpServletRequest` and `HttpServletResponse` interfaces
- Request Parameters
- Including and Forwarding
- Scope Objects
- Sessions

HTTP

The HyperText Transfer Protocol is the foundation of the Internet. The request and response mechanism between a browser and a web server uses the HTTP protocol. HTTP is actually a fairly simple protocol, consisting of some of the following methods:

- **GET:** Probably the most common type of request, the GET method means the client is making a request for information. A GET request contains the name of the resource being requested, and can also contain parameters. The parameters are a part of the URL, and therefore are limited by length. However, this allows for the URL to be bookmarked and visited at a later time. The GET method is somewhat analogous to someone looking on a billboard for information.
- **POST:** Another common HTTP method, the POST method is used for clients who want to post information on to the server. The POST method is designed for those situations where the client is sending data to the server, perhaps to be stored by the server. The mechanism for passing in this data is different than the GET method, and the URL does not change for POST requests. This means a client cannot bookmark a URL corresponding to a POST, which is appropriate since the data has already been "posted". The POST method is somewhat analogous to someone posting information on to a billboard.
- **PUT:** The PUT method can be used to place a document onto the server.
- **DELETE:** The DELETE method can be used to delete a document from the server.
- **HEAD:** The HEAD method is sent by a client when the client only wants to see the header of the response being sent back.
- **TRACE:** The TRACE method simply returns to the client's request. This is done for debugging purposes.
- **OPTIONS:** A client sends the OPTIONS method to determine which HTTP methods a particular resource provides.

HTTP 1.1 is the latest version of the protocol, and there are other methods in the protocol. However, when used in a typical web environment, the GET and POST methods are the most common and should provide all the functionality you need.

Requests

HTTP *requests* from a client consist of a set of headers called the request message. Request messages appear in the following format:

```
GET /training/index.html HTTP/1.1
Connection: close
Host: www.javalicense.com
Referer: http://www.google.com/search
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 95)
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
```

FIGURE 1. *An example of an HTTP GET request.*

- The first line of the header is called the request line. It contains the HTTP method, the resource desired, and the version of HTTP.

```
GET /training/index.html HTTP/1.1
```

- The Connection header tells the server to close the connection after the response is sent.
- The Host header represents the host server where the resource is located. The Referer header represents the location of where the request originated.
- The User-Agent header contains information about the client originating the request. This header can be used by the server to customize the response to a specific web browser.

- The Accept header denotes which type of media is acceptable for the response. The */* value denotes that any media is acceptable. For example, the header

```
Accept: text/*, audio/*
```

tells the server that this client will accept any type of text and/or audio.

- The Accept-Encoding header denotes which type of encoding the client can accept. Common values are gzip and compress.
- The Accept-Language header denotes the language preferred by the client. The us-en value denotes American English.

Responses

A *response* consists of a header similar to the request header, along with the resource that the client requested. The response message looks something like:

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 19 Oct 1999 10:27:58 GMT
Server: JavaWebServer/2.0
Last-Modified: Mon, 16 Oct 1999 21:08:43 GMT
Content-Length: 242
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>
JLicense, Inc. Training Solutions
</TITLE>
</HEAD>
<BODY>
This is the body of the message...
</BODY>
</HTML>
```

FIGURE 2. *An example of an HTTP response.*

- The first line of the response is called the status line. This includes the version of HTTP used by the server, a status code and a message. The status code 200 means everything went OK. Status codes in the 200's mean the request was handled successfully. Status codes in the 300's are for redirected requests. The 400's denote an error typically blamed on the client, such as the infamous status code 404 Not Found. Status codes in the 500's typically represent a problem blamed on the server.
- The Connection header denotes that the connection should be closed. The Date and Last-Modified are self-explanatory.
- The Server is the brand of the web server.

- The Content-Length represents the size of the response in bytes.
- The last part of the response is called the *response body* and represents the resource that the client requested.

HttpServletRequest

The information contained in a HTTP request header is encapsulated into an object of type `javax.servlet.http.HttpServletRequest`. The container creates this object and passes it in to the `service()` method of the servlet.

`HttpServletRequest` is an interface that contains various methods for retrieving information about the HTTP request. Some of the methods include:

```
public String getHeader(String name)
```

returns the requested header information. The header names can be retrieved using the `getHeaderNames()` method of the request object.

```
public String getMethod()
```

returns the HTTP method of the request, e.g. GET or POST

```
public HttpSession getSession(boolean create)
```

returns a session object for this client. If `create` is true, a new one will be created if one does not exist. If `create` is false, then null will be returned if a session object is not found for this client.

```
public Cookie [] getCookies()
```

returns any `Cookie` objects sent with the request.

```
public RequestDispatcher getRequestDispatcher(String path)
```

returns a reference to the request dispatcher of another resource of the Web server, typically another servlet or `JavaServer Page`. You use the request dispatcher to include another resource or forward the request to another resource.

```
public void setAttribute(String name, Object x)
```

adds the given Object to the request. It can be retrieved later using the name argument and the `getAttribute()` method. Objects stored in the request object are said to have request scope.

```
public Object getAttribute(String name)
```

returns a reference to the attribute matching the name.

```
public void removeAttribute(String name)
```

removes the given attribute from the request object.

```
public String getParameter(String name)
```

used for obtaining the parameters of the request.

HttpServletResponse

The container creates a `javax.servlet.http.HttpServletResponse` object for each client access and passes this response object to the `servlet service()` method (similar to the request object). The response object contains HTTP information about the response, including the output stream for the client.

The methods in the `HttpServletResponse` interface allow for the servlet to send a response to the client using a specific content. Some of the methods of the `HttpServletResponse` interface include:

```
public void setContentType(String type)
```

sets the content type of the response. For example, an HTML response would have type "text/html".

```
public void setContentLength(int len)
```

tells the container how large the body of the response is.

```
public ServletOutputStream getOutputStream() throws IOException
```

use the Servlet's output stream to send binary data back to the client. A servlet can invoke either `getOutputStream()` or `getWriter()`, but not both.

```
public PrintWriter getWriter() throws IOException
```

use the Servlet's print writer to return character data back to the client. A servlet can invoke either `getOutputStream()` or `getWriter()`, but not both.

`public void sendError(int sc, String message) throws IOException`
used for informing the client that an HTTP error has occurred. The `sc` argument is one of the static fields defined in the `HttpServletResponse` interface. The `message` is optional and will be displayed to the client. Invoking `sendError()` is only successful if you have not already sent data back to the client. Once this method is invoked, no other data can be returned.

`public void addCookie(Cookie c)`
adds a Cookie to the client's response.

Request Parameters

Parameters are data contained in the request object of a servlet that appear in (name, value) pairs. The parameters can be added to the request in various ways, but typically appear in HTML forms.

If the HTTP method of the request is GET, then all parameter information will appear in the Servlet's URL appearing after a ?. For example, the following URL has two request parameters, new and locale:

```
http://www.javalicence.com/RegisterServlet?new=yes&locale=en
```

In this example, new is equal to yes and locale is equal to en.

NOTE

If the HTTP method is POST, then parameters do not appear in the URL. In either case (either GET or POST), your servlet uses the `getParameter()` method of the request object to obtain the values of the parameters.

You use the following `getParameter()` method to obtain the value of a parameter:

```
public String getParameter(String name)
```

returns the value of the parameter whose name is name. If no matching parameter is found, then the method returns null.

The `HttpServletRequest` interface also contains the following method for determining the parameter names:

```
public String [] getParameters()
```

returns an array containing the names of all parameters sent with this client.

The following `ParameterDemo` servlet demonstrates using request parameters. It verifies that a parameter named “id” is present.

```
1  package demos;
2
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  import java.io.*;
6
7  public class ParameterDemo extends HttpServlet
8  {
9      public void doGet(HttpServletRequest req,
10                      HttpServletResponse res)
11                      throws ServletException, IOException
12      {
13
14          String id = req.getParameter("id");
15          if(id == null)
16          {
17              res.sendError(HttpServletResponse.SC_BAD_REQUEST,
18                          "id parameter missing");
19          }
20          else
21          {
22              res.setContentType("text/html");
23              PrintWriter out = res.getWriter();
24
25              out.println("<html>");
26              out.println("<body>");
27              out.println("<h2>Welcome</h2>");
28              //Here, we would validate that id is a valid id, then display
29              //the data associated with the particular id
30              out.println("<p>Here is the information for id "
31                          + id + "</p>");
32              out.println("</body>");
33              out.println("</html>");
34              out.close();
35          }
36      }
37  }
38
```

FIGURE 3. *The `ParameterDemo` demonstrates using parameters.*

Figure 4 shows a sample output of the ParameterDemo servlet when the id parameter appears in the URL.

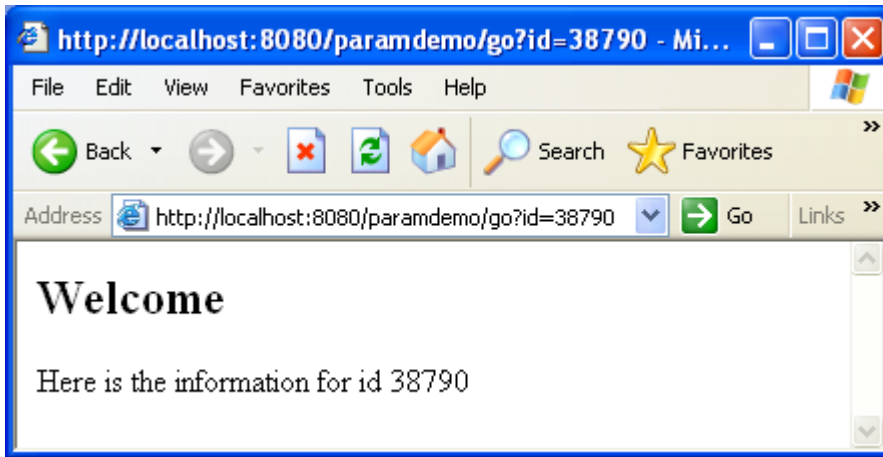


FIGURE 4. *A sample output of the ParameterDemo servlet.*

Figure 5 shows the output of the ParameterDemo servlet when the id parameter does not appear on the URL:

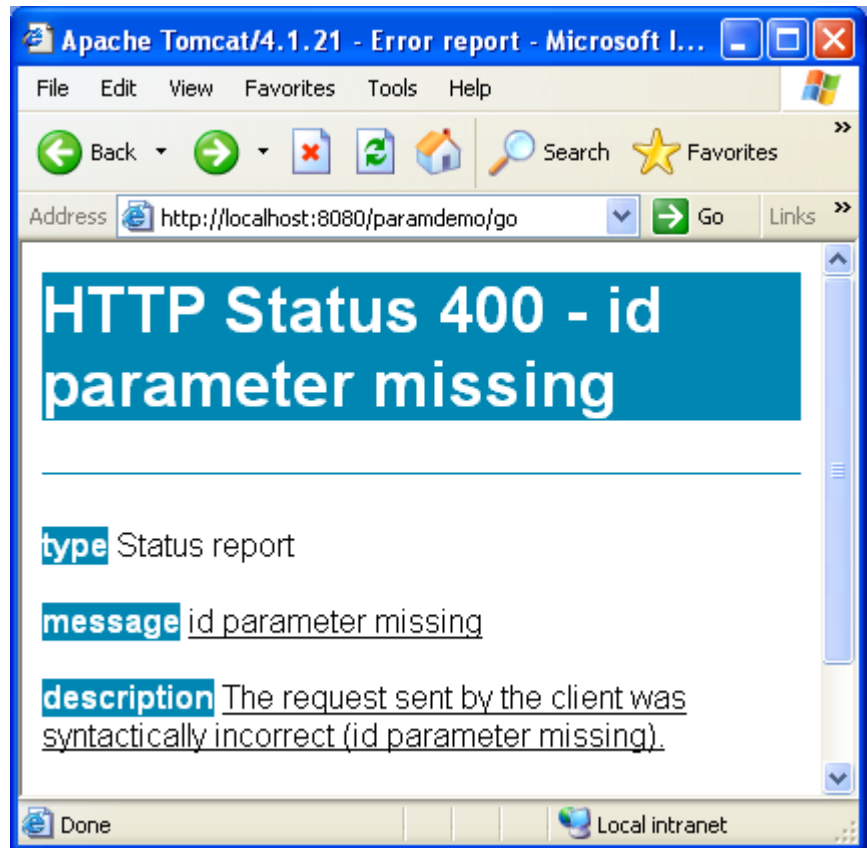


FIGURE 5. The *ParameterDemo* servlet output when no *id* parameter is defined.

The Request Dispatcher

A servlet can utilize the services of another Web component in two ways:

- **Including:** the response of a servlet can include the output of another Web component.

NOTE

The included Web component can only send output to the client; it can not alter the header information of the response object.

- **Forwarding:** a servlet can forward a client request to another Web component.

NOTE

The original servlet can not send any output to the client, but instead forwards the request and response to another Web component.

For a servlet to use including or forwarding, the servlet needs to obtain the request dispatcher for the other Web component being utilized. There are three steps involved in including or forwarding:

1. Obtain a reference to the servlet context by invoking `getServletContext()`. The return value is a `javax.servlet.ServletContext` object, which contains various methods for the servlet to communicate with the container.
2. Obtain a reference to the `RequestDispatcher` object for the particular Web component being utilized. Use the `getRequestDispatcher(String name)` method, which takes in the path and name of the Web component and returns the request dispatcher object.
3. Using the `RequestDispatcher` object, invoke either the `include()` or `forward()` method. Each method takes in two arguments: the `HttpServletRequest` and `HttpServletResponse` objects.

The `javax.servlet.RequestDispatcher` interface has two methods:

```
public void include(ServletRequest req, ServletResponse res)
               throws ServletException, IOException
```

Includes the content of the Web component associated with this `RequestDispatcher` instance.

```
public void forward(ServletRequest req, ServletResponse res)
               throws ServletException, IOException
```

Forwards the request to the Web component associated with this `RequestDispatcher` instance.

NOTE

In the Model-View-Controller design, forwarding and including becomes an important feature of servlets. A servlet can be used to redirect requests to an appropriate servlet or JSP page.

An Example of Including

The following `WelcomePage` servlet in Figure 6 demonstrates how to include the output from a JavaServer Page named `DisplayHeading`. The `DisplayHeading.jsp` page is defined in .

```
1  package demos;
2
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  import java.io.*;
6
7  public class IncludeDemo extends HttpServlet
8  {
9      public void doGet(HttpServletRequest req,
10                      HttpServletResponse res)
11                      throws ServletException, IOException
12      {
13          res.setContentType("text/html");
14          PrintWriter out = res.getWriter();
15
16          out.println("<html>");
17          out.println("<body>");
18
19          ServletContext context = getServletContext();
20          RequestDispatcher dispatch =
21              context.getRequestDispatcher("/DisplayHeading.jsp");
22          if(dispatch != null)
23          {
24              dispatch.include(req, res);
25          }
26
27          out.println("This page demonstrates including the output
28 of a JavaServer Page in a Servlet");
29          out.println("This portion of the output is generated
30 from the Servlet.");
31          out.println("</body>");
32          out.println("</html>");
33          out.close();
34      }
35  }
```

FIGURE 6. *The `IncludeDemo` servlet includes the `DisplayHeading.jsp` page.*

Notice the DisplayHeading.jsp page is written to be used as included HTML. The HTML that it sends to the response would not stand on its owns as a Web page since it only outputs a <table> element.

```
1  <table width="75%" border="0">
2    <tr>
3      <td width="34%"></td>
5      <td width="66%">Welcome to BEST
6        <%! String username; %>
7        <% username = request.getParameter("username");
8          if(username != null)
9            {
10              out.println(", " + username);
11            }
12        %>
13      </td>
14    </tr>
15  </table>
```

FIGURE 7. *The DisplayHeading.jsp page.*

Figure 8 shows the Web page generated from the IncludeDemo servlet.

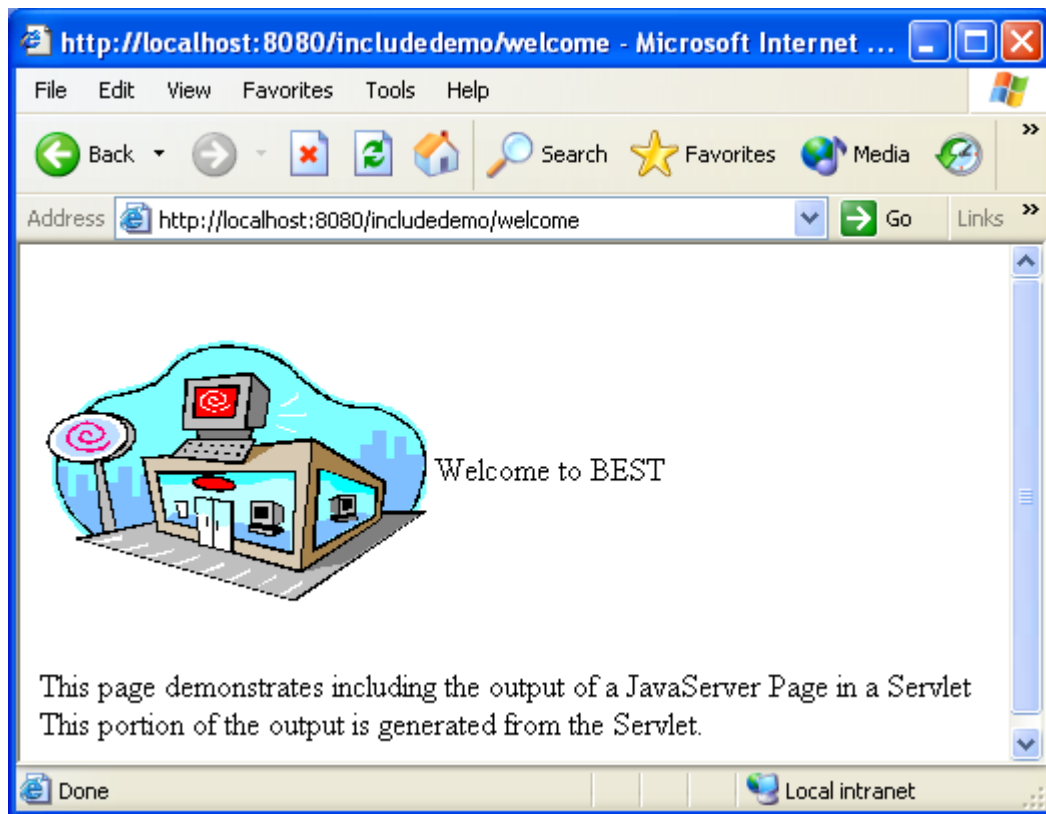


FIGURE 8. *The output of the IncludeDemo servlet.*

Notes

Lab 1: The Servlet Controller

Objective: In this lab, you will write a servlet that acts as a controller for the BEST Web application. While a customer is browsing the BEST Web site for items to purchase, all requests will go through the `BestController` servlet that you develop in this lab. This will have several design advantages that will become apparent as our BEST application evolves.

Perform the following tasks:

1. Create a new directory named `/best` whose path is `%CATALINA_HOME%/webapps/best`. Create a `/WEB-INF` directory within `/best` and a `/classes` directory within `WEB-INF`.
2. Write a class named `BestController` and save it in the `/best/WEB-INF/classes` directory. This class should be declared in the `"com.best"` package, extend `HttpServlet`, and contain the `doGet()` method.
3. Within `doGet()`, use the `getParameter()` method to obtain the value of a parameter named `"action"`.
4. If `"action"` is not defined as a parameter, then send back a Web page with a simple welcome greeting. Include the `DisplayHeading.jsp` from Figure 7 on page 71. Copy `DisplayingHeading.jsp` from the accompanying lab files to the `best/WEB-INF` directory.
5. In addition, if the `"action"` parameter is not defined, add the following link (or similar) to the page that is displayed. (Note that the `getContextPath()` method is useful here because it allows you to not hard-code information like the server name, port number and Web context.)

```
out.println("<p><a href=\"" + req.getContextPath() + "/"
browse\">Browse products</a></p>");
```


6. If "action" is defined, it should be one of the following values: "browse", "addToCart", "removeFromCart", "showCart" or "checkout". If it is not one of these values, send back an error code as the response. Use error code 400 and the message "action parameter missing". (This will be used for debugging purposes.)
7. If "action" is one of the five valid values, then the request is going to be forwarded to another Web component. If "action" equals "addToCart", "removeFromCart" or "showCart", forward the request to the URL "/managecart".
8. If "action" is equal to "browse", forward the request to "/browse".
9. If "action" equals "checkout", send the request to "/checkout".
10. Save and compile the BestController.java class. Be sure to use the "-d ." option when compiling.
11. Create a Web Application Deployment Descriptor and save it in the file /best/WEB-INF/web.xml. Create two URL mappings for the BestController servlet: "/controller" and "/welcome", similar to the following:

```
<servlet>
  <servlet-name>controller</servlet-name>
  <servlet-class>com.best.BestController</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>controller</servlet-name>
  <url-pattern>/controller</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>controller</servlet-name>
  <url-pattern>/welcome</url-pattern>
</servlet-mapping>
```

12. Open a Web browser and access the servlet, making sure both your URL mappings in the previous statement work. Verify your servlet is working properly.

What you should see: If the “action” parameter is not defined, you should see a simple welcome page similar to Figure 9. If the “action” parameter is defined but not one of the valid values, you should see an error page. If “action” is properly defined, you should see a 404 Not Found error since you have created the other Web components yet.

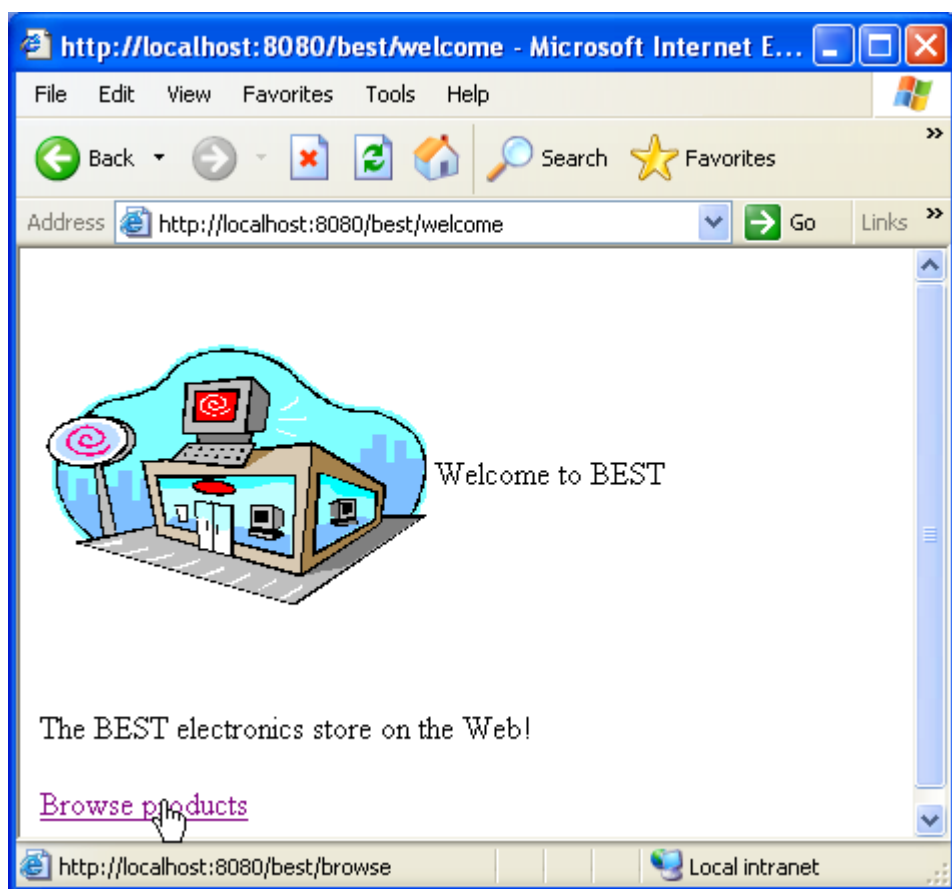


FIGURE 9. A sample output when the action parameter is not defined.

Object Scope

Objects within a servlet container have four different types of scope:

- **Page scope:** An object that has page scope is a field in the object class. The term "page scope" is used because the object is only available to that particular servlet.
- **Request scope:** An object can be associated with a client request. Objects with request scope can be accessed by another servlet that is forwarded the request.
- **Application scope:** An object with application scope is accessible to any servlet in the container.
- **Session scope:** A session is the term used to describe the time a client spends at a particular Web site. Objects can be associated with a client's session, making the objects available to any servlet accessed during the session.

Page scope does not involve any Servlet API classes, since it is only a term used to describe the fields of a servlet class. The other scopes are obtained by using various classes and methods in the Servlet API. We will now discuss how to achieve request, application, and session scope.

Notes

Request Scope

An object with request scope is attached to the `HttpServletRequest` object passed in to the `doXXX()` method of the servlet. The following methods used to achieve request scope are found in the `ServletRequest` interface, the parent interface of `HttpServletRequest`:

```
public void setAttribute(String name, Object x)
```

adds the given object to the request. The name is used to retrieve the object later.

```
public Object getAttribute(String name)
```

returns a reference to the object with request scope whose name is `name`. If no object exists in the request with the given name, the method returns `null`.

```
public java.util.Enumeration getAttributeNames()
```

returns a collection of `String` objects containing the names of all attributes associated with the request.

```
public void removeAttribute(String name)
```

removes the object with the given name from the request.

NOTE

Attributes are reset between each request. If an object needs to exist beyond the life of the request, then application or session scope needs to be used.

Notes

An Example of Request Scope

Internationalization (i18n) is a built-in feature of J2SE, and therefore is readily available for use in servlets. The `java.util.ResourceBundle` class allows you to define values for specific words and phrases, making it easy for your Java applications to switch languages.

- The following `RequestDemo` and `ShowText` servlets use i18n to demonstrate a Web page that is not directly tied to any language. By changing the language settings of your Web browser, the text can change from English to Spanish.

```
1    package demos;
2
3    import javax.servlet.*;
4    import javax.servlet.http.*;
5    import java.io.*;
6    import java.util.*;
7
8    public class RequestDemo extends HttpServlet
9    {
10        public void doGet(HttpServletRequest request,
11                           HttpServletResponse response)
12            throws ServletException, IOException
13        {
14            Locale locale = request.getLocale();
15            System.out.println("Using locale " + locale);
16
17            ResourceBundle text =
18                ResourceBundle.getBundle("text.Messages", locale);
19            request.setAttribute("text", text);
20
21            ServletContext context = getServletContext();
22            RequestDispatcher dispatch =
23                context.getRequestDispatcher("/showtext");
24            if(dispatch != null)
25            {
26                dispatch.forward(request, response);
27            }
28        }
29    }
```

FIGURE 10. *The `RequestDemo` servlet attaches a `ResourceBundle` to the request object.*

- The RequestDemo servlet attaches an object of type ResourceBundle to the request object, thereby giving this object request scope.
- When the request is forwarded to the “/showtext” Web component, the ResourceBundle object “text” goes along with the request.
- The ShowText servlet in Figure 11 uses the `getAttribute()` method of the request object to obtain the ResourceBundle object.

```
1  package demos;
2
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  import java.io.*;
6  import java.util.*;
7
8  public class ShowText extends HttpServlet
9  {
10     public void doGet(HttpServletRequest request,
11                        HttpServletResponse response)
12                        throws ServletException, IOException
13     {
14         ResourceBundle text = (ResourceBundle)
15             request.getAttribute("text");
16
17         response.setContentType("text/html");
18         PrintWriter out = response.getWriter();
19
20         out.println("<html>");
21         out.println("<body>");
22         out.println("<h2>" + text.getString("Welcome")
23             + "</h2>");
24         out.println("<form>" + text.getString("Name")
25             + "<input type=\"text\"><br>");
26         out.println(text.getString("CCNumber")
27             + "<input type=\"text\"><br>");
28
29         out.println("</body>");
30         out.println("</html>");
31         out.close();
32     }
33 }
```

FIGURE 11. *The ShowText servlet uses `getAttribute()` to obtain the “text” object from the request.*

Using Internationalization

The `java.util.ResourceBundle` class contains a static method named `getBundle()` that takes in a locale and a “base name”. The base name represents the name of a set of classes that each has a different name, but their “base” name is the same.

In the `RequestDemo` example, the base name was “`text.Messages`”. There are two classes (although we could write any number of them) that were used:

- `text.Messages` - which happens to match the base name used in the call to `getBundle()`. This class is selected if no other locale is requested. This class is defined in .
- `text.Messages_es` - which is used when the client’s locale is “es”, which represents an international version of Spanish. This class is defined in .

```
1  package text;
2  import java.util.*;
3  public class Messages extends ListResourceBundle {
4      public Object[][] getContents() {
5          return contents;
6      }
7      static final Object[][] contents = {
8          {"ShoppingCart", "Shopping Cart"},
9          {"Products", "Products"},
10         {"Welcome", "Welcome to BEST"},
11         {"RemoveItem", "Item removed "},
12         {"ShoppingCart", "Your shopping cart contains "},
13         {"AddItem", "You added "},
14         {"AddToCart", "Add to Cart"},
15         {"ItemQuantity", "Quantity"},
16         {"Description", "Description"},
17         {"Price", "Price"},
18         {"RemoveItem", "Remove Item"},
19         {"Checkout", "Check Out"},
20         {"Amount", "Your total is"},
21         {"Checkout", "Click here to checkout"},
22         {"Name", "Name"},
23         {"CCNumber", "Credit Card Number"},
24         {"Submit", "Purchase Items"},
25         {"ThankYou", "Thank you for purchasing your
26             electronics from us " } };
27 }
```

FIGURE 12. *The `text.Messages` class is for the English locale.*

```
1 package text;
2 import java.util.*;
3 public class Messages_es extends ListResourceBundle {
4     public Object[][] getContents() {
5         return contents;
6     }
7
8     static final Object[][] contents = {
9         {"ShoppingCart", "Carro de Compras"},
10        {"Products", "Cat" + "\u00e1" + "logo"},
11        {"Welcome", "Ola BEST"},
12        {"RemoveItem", "Acaba de quitar "},
13        {"ShoppingCart", "Su carro de compras tiene "},
14        {"AddItem", "A" + "\u00f1" + "adi" + "\u00f3"},
15        {"AddToCart", "A" + "\u00f1" + "adir al Carro de Compras"},
16        {"ItemQuantity", "Cantidad"},
17        {"Description", "T" + "\u00ed" + "tulo"},
18        {"Price", "Precio"},
19        {"RemoveItem", "Quitar art" + "\u00ed" + "culo"},
20        {"Checkout", "Salir"},
21        {"Amount", "Su importe total es"},
22        {"Checkout", "Entregar la Informaci" + "\u00f3" + "n"},
23        {"Name", "Nombre"},
24        {"CCNumber", "N" + "\u00fa" + "mero de Tarjeta de Cr" +
25        "\u00e9" + "dito"},
26        {"Submit", "Entregar la Informaci" + "\u00f3" + "n"},
27        {"ThankYou", "Gracias"}
28    };
```

FIGURE 13. *The text.Messages_es class is for Spanish.*

NOTE

You can change the locale of Internet Explorer by clicking on “Tools” then “Internet Options”. On the “General” tab, click the “Languages” button. A list of language appears, along with their locale value. If you want to view the RequestDemo in Spanish, move the “es” locale to the top of the list of languages.

Figure 14 shows the output of the RequestDemo servlet using the text.Messages class as the ResourceBundle. Figure 15 shows the output when text.Messages_es is used as the ResourceBundle.

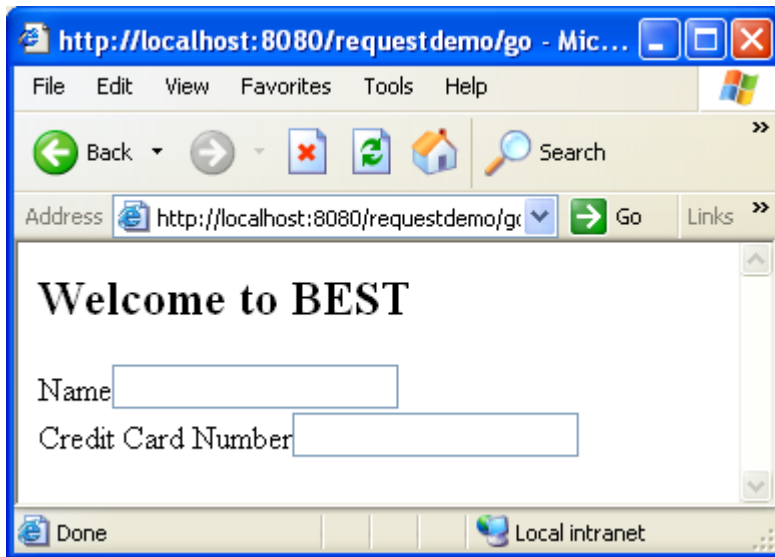


FIGURE 14. *The RequestDemo servlet using the default locale (English).*

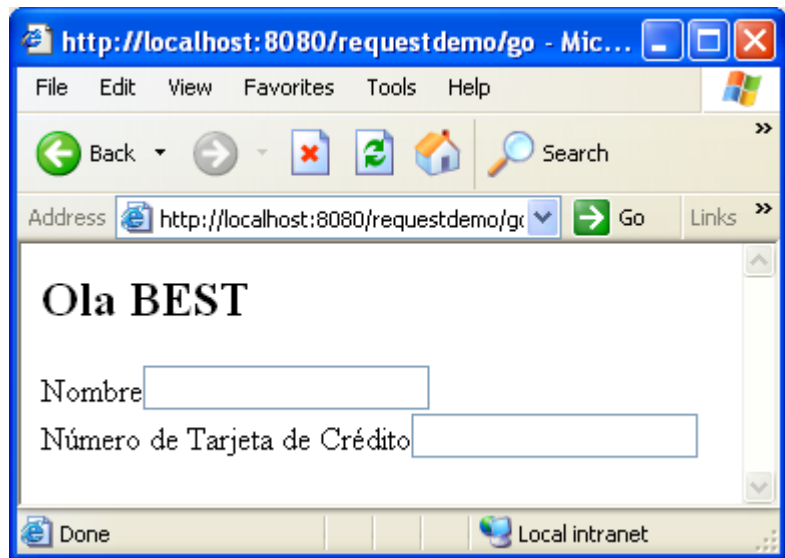


FIGURE 15. *The RequestDemo servlet using the Spanish locale.*

Application Scope

An object can be given application scope by assigning it as an attribute of the servlet container. Methods for communicating with the container are found in the `ServletContext` interface. Servlets obtain a reference to the servlet context in the `init()` method using the `ServletConfig` argument.

The following methods of the `ServletContext` interface are used when working with application scope. Notice they have the same signatures as the methods used for request scope in the `ServletRequest` interface.

```
public void setAttribute(String name, Object x)
```

adds the given object to the servlet container. The name is used to retrieve the object later.

```
public Object getAttribute(String name)
```

returns a reference to the object with application scope whose name is name. If no object exists in the container with the given name, the method returns null.

```
public java.util.Enumeration getAttributeNames()
```

returns a collection of strings containing the names of all attributes associated with the container.

```
public void removeAttribute(String name)
```

removes the object with the given name from the container.

The following `ApplicationDemo` class demonstrates a servlet that locates an object of type `InventoryDB` that is an attribute of the servlet context, meaning the `InventoryDB` object has application scope.

NOTE

The idea behind this example is that the `InventoryDB` object represents a `JavaBean` that accesses and underlying database. This servlet obtains the object in its `init()` method and frees the object in its `destroy()` method.

```
1  public class ApplicationDemo extends HttpServlet
2  {
3      InventoryDB invDB;
4
5      public void init()
6      {
7          ServletContext context = getServletContext();
8          invDB = (InventoryDB) context.getAttribute("invDB");
9
10         if(invDB == null)
11         {
12             System.out.println("Could not locate invDB in
application scope");
13         }
14         else
15         {
16             invDB.open();
17         }
18     }
19
20     public void destroy()
21     {
22         invDB.close();
23     }
24 }
```

FIGURE 16. *The `ApplicationDemo` servlet demonstrates using application scope.*

Session Scope

A session is the term used to describe the time a client spends at a Web site, assuming that subsequent requests are related to previous requests. Sessions are commonplace in today's Web sites:

- Having users log on to access private data without having to verify their username and password on each page visited.
- Shopping carts.
- Online banking.
- Tracking the pages that a client visits to determine their personal likes and dislikes.

These are only a few of the practical uses of client session tracking. There are three techniques used by servlets to create sessions:

- Cookies: small amounts of information stored on the client's machine.
- URL Rewriting: appending a unique ID to the end of each URL that the client visits.
- The session object: an object of type `HttpSession` that is created and maintained by the Web server. The session object simplifies the creation and tracking of sessions by creating cookies or using URL rewriting for you.

Notes

Cookies

Cookies are small amounts of data stored on a client's machine. Each cookie is associated with a particular Web site. When a client accesses that Web site, all associated cookies are attached to the HTTP request and sent to the server. The servlet container creates a `javax.servlet.http.Cookie` object for each cookie and attaches them to the `HttpServletRequest` object passed to the `service()` method.

A servlet can obtain the cookies from a client using the following method of the `HttpServletRequest` interface:

```
public Cookie [] getCookies()
```

returns an array containing the cookies sent with the client request. If no cookies are present, method returns null.

A servlet can add a cookie to a client's machine by instantiating a `Cookie` object and attaching it to the `HttpServletResponse` object of the client using the method:

```
public void addCookie(Cookie cookie)
```

add the cookie to the client's response. The cookie will be stored on the client's machine.

A Web site can (possibly) add up to 20 cookies to a client's machine with a size of 4KB each. Different Web browsers handle cookies differently, so the number and size of cookies may vary.

Notes

The Cookie Class

The `javax.servlet.http.Cookie` class represents the information of a cookie. The `Cookie` class has only one constructor:

```
public Cookie(String name, String value)
```

instantiates a `Cookie` object with the given name and value.

A cookie consists of a name and a value, both of which are `String` objects. The name of the cookie must be determined at construction time and can not be changed. The value, of course, can be changed at any time.

The following is a list of some of the methods of the `Cookie` class:

```
public void setValue(String value)
```

assigns the value of the cookie.

```
public String getValue()
```

returns the value of the cookie.

```
public String getName()
```

returns the name of the cookie.

```
public void setMaxAge(int expiry)
```

sets the maximum age of the cookie, which determines when the cookie will expire. A negative value means the cookie should be deleted when the browser exits. A zero value causes the cookie to be deleted immediately.

```
public void setComment(String purpose)
```

a cookie can have a comment associated with it, which is stored as a String object.

```
public String getComment()
```

returns the comment of the cookie, or null if no comment was assigned.

An Example Using Cookies

The following VerifyLogon servlet demonstrates using cookies to track a session of a customer that logs on to an online banking application. If the username and password are valid, a cookie is created and attached to the response. This cookie is then retrieved on all subsequent requests.

```
1  import javax.servlet.*;
2  import javax.servlet.http.*;
3  import java.io.*;
4  import java.util.*;
5
6  public class VerifyLogon extends HttpServlet
7  {
8      //the fields and init() method are left out for brevity
9      public void doPost(HttpServletRequest req,
10                          HttpServletResponse res)
11          throws ServletException, IOException
12      {
13          res.setContentType("text/html");
14          PrintWriter out = res.getWriter();
15          out.println("<html>");
16          out.println("<body>");
17          String accountstring = req.getParameter("account");
18          String password = req.getParameter("password");
19          int accountnumber = Integer.parseInt(accountstring);
20          if(av.verifyUser(accountnumber, password))
21          {
22              //create a BankAccount object and
23              //attach it to a new Cookie object
24              Cookie c = new Cookie("accountnumber",
25                                  "" + accountnumber);
26              res.addCookie(c);
27
28              out.println("<p> Your logon was successful. </p>");
29              out.println("<p> <a href=\"\"/BankingOptions\">
Continue...</a> </p> ");
30          }
31          out.println("</body>");
32          out.println("</html>");
33          out.close();
34      }
35  }
```

FIGURE 17. *The VerifyLogon form attaches a cookie to a client's response.*

When the user clicks on the link sent back by the VerifyLogon form, they are sent to the “/BankingOptions” servlet, which is defined in the following BankingOptions class.

```
1  import javax.servlet.*;
2  import javax.servlet.http.*;
3  import java.io.*;
4
5  public class BankingOptions extends HttpServlet
6  {
7      public void doGet(HttpServletRequest req,
8                          HttpServletResponse res)
9                          throws ServletException, IOException
10     {
11         res.setContentType("text/html");
12         PrintWriter out = res.getWriter();
13         out.println("<html>");
14         out.println("<body>");
15
16         Cookie [] cookies = req.getCookies();
17         int accountnumber = 0;
18         for(int i = 0; i < cookies.length; i++)
19         {
20             if(cookies[i].getName().equals("accountnumber"))
21             {
22                 accountnumber =
23                     Integer.parseInt(cookies[i].getValue());
24                 break;
25             }
26         }
27
28         if(accountnumber == 0) //cookies wasn't found
29         {
30             out.println("<p>Sorry, but your account information
was not found!</p>");
31             out.println("</body></html>");
32             out.close();
33             return;
34         }
35
36         BankAccount account = new BankAccount(accountnumber);
37         out.println(" <h2>Welcome, " + account.getName()
38                     + " </h2> ");
39     }
```

FIGURE 18. *The BankingOptions servlet determines who the client is by using the cookie found in the request.*

The session Object

A Web server that supports servlets uses a session object to track the sessions of each client. A servlet or JavaServer Page can obtain a reference to this session object and attach objects to it, thereby giving the object session scope.

The session object is of type `javax.servlet.http.HttpSession`, and the servlet obtains a reference to the session object using one of the following methods (found in the `HttpServletRequest` interface):

```
public HttpSession getSession()
```

returns the client's session object or creates a new one if one does not exist.

```
public HttpSession getSession(boolean create)
```

returns the client's session object. If a session object has not been created yet, then a new one will be instantiated if `create` is true; otherwise, the method returns null.

The container is responsible for managing the session objects and ensuring each client is assigned a unique session object. The `HttpSession` interface contains methods for storing objects within the session object, thereby giving the stored objects session scope:

```
public void setAttribute(String name, Object x)
```

adds the given object to the session object. The name is used to retrieve the object later.

```
public Object getAttribute(String name)
```

returns a reference to the object within the session scope whose name is `name`. If no object exists with the given name in the session object, the method returns null.

Q&A

How does the servlet container manage a session object for each user? More specifically, how does the Web server know that it's you coming back each time you request a resource? Well, the Web server uses cookies to keep track of the clients. If the only reason you are using cookies is for session tracking, then you might as well use the session object.

What if the client has turned off support for cookies in their Web browser? Then you can still use the session object, but you need to make sure that each link you send back to the client contains a unique ID. This is known as URL rewriting, which we will discuss next.

An Example of a Session

The following servlet demonstrates the usage of HttpSession objects. The first time a client access this servlet, a session object is created and a ShoppingCart object is added to the session object. On subsequent visits, the ShoppingCart is retrieved from the existing session object.

```
1 import javax.servlet.*;
2 import javax.servlet.http.*;
3 import java.io.*;
4 import java.util.*;
5
6 public class Catalog extends HttpServlet
7 {
8     public void doGet(HttpServletRequest req,
9                        HttpServletResponse res)
10        throws ServletException, IOException
11     {
12         res.setContentType("text/html");
13         PrintWriter out = res.getWriter();
14
15         HttpSession session = req.getSession(false);
16         ShoppingCart cart = null;
17
18         if(session == null)
19         {
20             session = req.getSession(true);
21             cart = new ShoppingCart();
22             session.setAttribute("cart", cart);
23         }
24         else
25         {
26             cart = (ShoppingCart) session.getAttribute("cart");
27         }
28
29         //the cart can now be used...
30     }
31 }
```

FIGURE 19. *The Catalog Servlet uses a ShoppingCart object with session scope.*

Notes

URL Rewriting

Another option for session tracking is *URL rewriting*, a technique where a unique identifier is appended to the end of each URL. The identifier is appended in the form of a parameter, so that as a client visits subsequent locations, the parameter is passed along with each client request.

URL rewriting has the advantage of being able to work even for clients who do not have support for cookies. The disadvantage is that URL rewriting can be tedious, since each link on the Web page that applies to the client's session must be encoded. The good news is that the `HttpServletResponse` interface contains a method for encoding a URL:

```
public String encodeURL(String url)
```

the string passed in to this method is appended with a unique ID if necessary.
The return value is the encoded URL.

NOTE

The `encodeURL()` method only rewrites the URL if necessary. In other words, if a client supports cookies, then URL rewriting is not necessary.

The general rule of thumb is that you should encode all of your URLs listed within a Web page that apply to the current session. If the client's browser does not support session tracking, then all URLs will be encoded properly. If the client does support some other type of session tracking, then encoding the URL does nothing. The container is responsible for ensuring this behavior.

Notes

An Example of URL Rewriting

The following servlet demonstrates encoding the URLs of the page.

```
1 import javax.servlet.*;
2 import javax.servlet.http.*;
3 import java.io.*;
4 import java.util.*;
5
6 public class Catalog extends HttpServlet
7 {
8     public void doGet(HttpServletRequest req,
9                         HttpServletResponse res)
10        throws ServletException, IOException
11     {
12         res.setContentType("text/html");
13         PrintWriter out = res.getWriter();
14         HttpSession session = req.getSession(false);
15         ShoppingCart cart = null;
16
17         if(session == null)
18         {
19             session = req.getSession(true);
20             cart = new ShoppingCart();
21             session.setAttribute("cart", cart);
22         }
23         else
24         {
25             cart = (ShoppingCart) session.getAttribute("cart");
26         }
27
28         out.println("<HTML><BODY>");
29         out.println("<a href="
30                    + res.encodeURL("/servlet/Clothes")
31                    + "> Clothing </a>");
32         out.println("<a href="
33                    + res.encodeURL("/items/Music.jsp")
34                    + "> Music </a>");
35         out.println("</BODY></HTML>");
36         out.close();
37     }
38 }
```

FIGURE 20. *The Catalog Servlet demonstrates URL rewriting.*

Notice the status bar of the Web browser in Figure 21, which shows the Clothing link.



FIGURE 21. *The URL shows the effect of `encodeURL()`.*

Lab 2: Using Resource Bundles

Objective: To become familiar with working with the session object. You will use the session object to associate a ResourceBundle with a client depending on their locale.

NOTE

Notice that the RequestDemo servlet in Figure 10 on page 82 attaches a ResourceBundle to a request object. This means the lifetime of that ResourceBundle is very short, since the request is lost once a response is sent back to the client. A better design would be to attach a ResourceBundle to each client's session. This greatly reduces the amount of effort in constructing and destroying ResourceBundle objects, and it is probably safe to assume that a client will not change their locale during a session.

In this lab, you will attach a ResourceBundle to the session object of each client.

Perform the following tasks:

1. Locate the directory `\requestdemo\WEB-INF\classes\text` (from the accompanying lab files), which contains two files: `Messages.class` and `Messages_es.class`. Copy the `\text` directory into your `%CATALINA_HOME%\webapps\best\WEB-INF\classes\` directory. This allows your BEST Web application to be able to use these two classes.
2. Open the `BestController` class from Lab 1 in your editor. Within the `doGet()` method, obtain a reference to the session object, creating a new session object if necessary.
3. Using the `getAttribute()` method, get the attribute named "text" of type `ResourceBundle` from the session object. If this attribute is not in the session object, create a new `ResourceBundle` using the `getBundle()` method and attach this object to the session object, assigning it a value "text". The base name of the `ResourceBundle` is "text.Messages" and use the `getLocale()` method of the request object to obtain the client's locale.
4. Save and recompile your `BestController` class.
5. Write a class named `BrowseServlet`. Declare it in the `com.best` package and have it extend `HttpServlet`.

6. Within the `doGet()` method of `BrowseServlet`, obtain a reference to the `ResourceBundle` attribute named "text" in the session object. This parameter should be in the request, but if it's not, be sure to avoid a `NullPointerException` by assigning your `ResourceBundle` reference to the English version of `text.Messages`.
7. Using the `ResourceBundle`, print out the "Welcome" string within the `doGet()` method of `BrowseServlet`. Also, create a `<form>` and have the label on the submit button be the "AddToCart" string, similar to the following:

```
out.println("<form>");  
out.println("<input type=\"submit\" value=\"" +  
            text.getString("AddToCart") + "\">");  
out.println("</form>");
```

8. Save the `BrowseServlet.java` file in the `%CATALINA_HOME%\webapps\best\WEB-INF\classes` directory.
9. Compile `BrowseServlet.java`.
10. Modify the deployment descriptor `%CATALINA_HOME%\webapps\best\WEB-INF\web.xml`, adding a `<servlet>` entry for the `BrowseServlet`. The servlet URL mapping should be `/browse`.
11. Open your Web browser and access the URL `http://localhost:8080/best/welcome`. This will create a new session for you. Click on the `/browse` link to view the `BrowseServlet` in the English locale.
12. Change your language preference of your browser to be "es" (Spanish) and access the `/browse` servlet again. You may need to close and re-open the browser to start a new session.

What you should see: You should be able to view the `/browse` page. The appearance of the page will vary, but a sample output in English is shown in Figure 22. You should see the text and button label in Spanish if you change the language settings on your Web browser, similar to Figure 23.

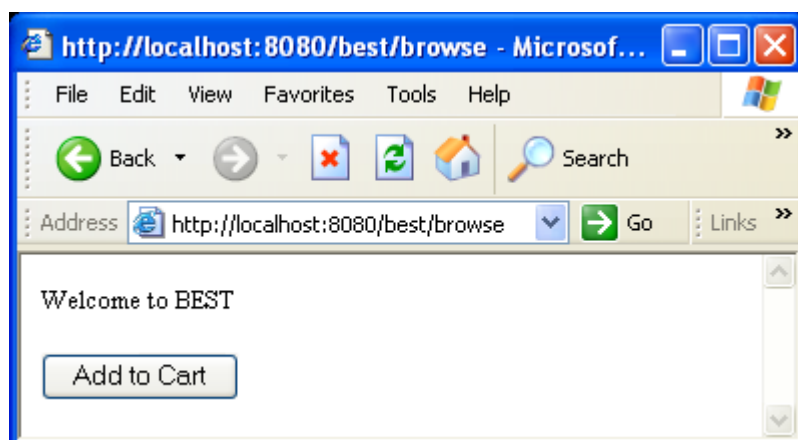


FIGURE 22. The output of the BrowseServlet in English.

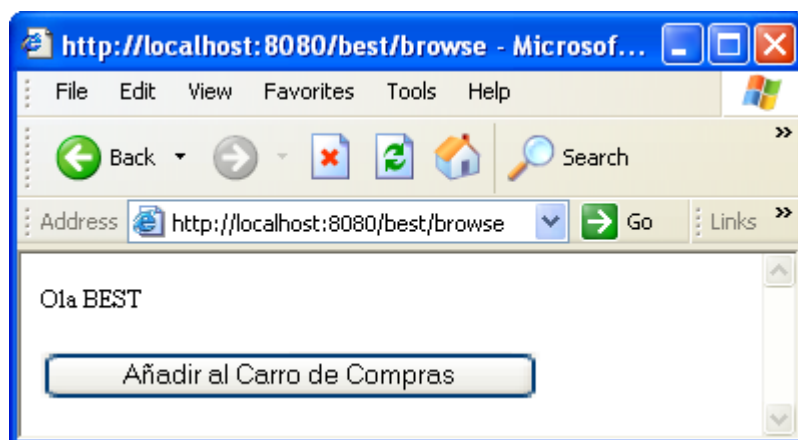


FIGURE 23. The output of the BrowseServlet in Spanish.

Lab 3 - The Shopping Cart

Objective: In this lab, you will attach a ShoppingCart object to the session object of each client that visits the BEST Web site.

Perform the following tasks:

1. The `com.best.ShoppingCart` is already written for you. `ShoppingCart.java` is in the `..\WebDevelopment\Solutions\Chapter_03\Lab3` directory. Copy and paste the file into the `%CATALINA_HOME%\webapps\best\WEB-INF\classes\` directory and compile it (using the `-d .` option). Open the `ShoppingCart.java` file in your editor and view its methods to familiarize yourself with the class.
2. Within the `doGet()` method of your `BestController` servlet, you are going to add a new `ShoppingCart` object to the session object if the client does not already have one. Check the session object for an attribute named `"cart"` of type `ShoppingCart`. If the `"cart"` attribute is null, instantiate a new `ShoppingCart` and set it as a session attribute named `"cart"`. Use `System.out.println()` to print out the message `"Creating a new cart"`.
3. If the session object already has a `"cart"` attribute, then print out `"Using existing cart"`. (There is no need to do anything else.)
4. Compile your `BestController` servlet and view it in your Web browser.

What you should see: At the standard output of your installation of Tomcat, you should see `"Creating a new cart"` the first time you visit the `/best/welcome` page and `"Using existing cart"` while the session is still current.



Using Servlets Effectively

In this chapter, we will discuss the details of servlet development and how to use them effectively. Topics discussed in this chapter include:

- Initializing servlets
- Destroying servlets
- Threading issues
- Cleanly destroying a servlet
- Servlet listeners
- Writing and deploying a servlet listener
- Servlet filters
- Writing and deploying a filter

Initializing Servlets

The `init()` method is invoked on a servlet object by the container exactly once after the servlet is instantiated in memory. The purpose of the `init()` method is to allow the servlet to initialize any fields and resources that need be used throughout the lifecycle of the servlet.

- If a servlet can not properly be initialized within `init()`, then the `init()` method should throw a `javax.servlet.UnavailableException`
- If the `init()` method throws a `javax.servlet.ServletException`, then the container will not make the servlet available for client requests.

For example, the following servlet obtain a reference to a data access object:

```
1  public class BestController extends HttpServlet
2  {
3      public void init() throws ServletException
4      {
5          try
6          {
7              InventoryDB invDB = new InventoryDB();
8              getServletContext().setAttribute("inventoryDB", invDB);
9          } catch (Exception e)
10         {
11             throw new UnavailableException(e.getMessage());
12         }
13     }
14 }
```

FIGURE 1. *Preparing a data access object in the `init()` method.*

Destroying a Servlet

A container may determine at any time that a servlet is no longer needed within the container and “destroy” the servlet, thereby taking the servlet out of service. Just before the container removes the servlet instance, it invokes the servlet’s `destroy()` method. This allows the servlet to free any resources and make sure all data integrity is maintained before the servlet is removed.

- The container attempts to not invoke `destroy()` on a servlet if a `service()` thread is still actively running.
- However, if a `service()` method takes a long time to execute, the servlet may time-out and the `destroy()` method may be invoked while `service()` threads are executing.
- After the `destroy()` method is invoked, the container will not invoke `service()` on the servlet instance again.

Typically, the `destroy()` method frees any resources allocated in the `init()` method. For example, the following servlet removes an object from the `ServletContext` that was added in `init()`.

```
1  public class BestController extends HttpServlet
2  {
3      public void init() throws ServletException
4      {
5          try
6          {
7              InventoryDB invDB = new InventoryDB();
8              getServletContext().setAttribute("inventoryDB", invDB);
9          } catch (Exception e)
10         {
11             throw new UnavailableException(e.getMessage());
12         }
13     }
14
15     public void destroy()
16     {
17         getServletContext().removeAttribute("inventoryDB");
18     }
19 }
```

FIGURE 2. *Using the `destroy()` method.*

The Servlet Thread Models

When multiple clients access a servlet simultaneously, the Web server does not typically instantiate a new servlet instance for each client. (This behavior can be achieved by implementing the `SingleThreadModel` interface, which is discussed later in the module.)

Instead, when a client accesses a servlet, the `service()` method is invoked in a separate thread. Therefore, each client is sharing the data of the servlet. This means your data should be synchronized if necessary. This behavior has many benefits, because it allows the Web server to efficiently handle requests in separate threads as opposed to instantiating many servlet objects on the server.

The following diagram illustrates multiple clients accessing a single servlet:

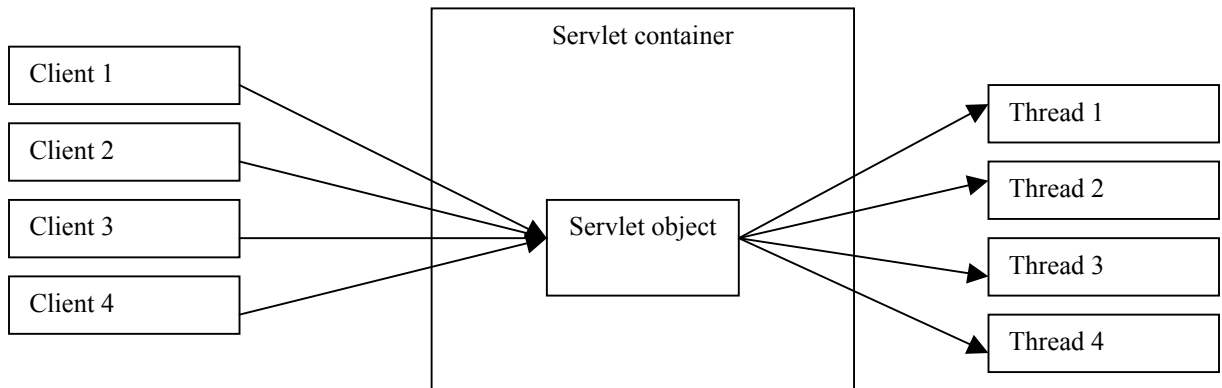


FIGURE 3. *Multiple clients of a Servlet run in multiple threads.*

There are two threading models in servlets:

- **The default threading model** has been used, where each client request on a servlet causes the `service()` method to be invoked in a separate thread.
- **The single thread model**, where no two client requests occur simultaneously on the same servlet instance.

In the default model, every client has access to each field in the servlet. In other words, the fields of the servlet are being shared by each client. If a field contains client-specific data, then the access to the field would have to be synchronized somehow.

However, you need to be careful when synchronizing the methods of a servlet. If you have many clients waiting to access a single synchronized object, then the performance can easily become unacceptable.

In those situations, it is better to use the single thread model for a servlet, which is discussed next.

The Single Thread Model

A servlet has the option of executing under the single thread model, which is:

- The container ensures that no two clients invoke a Servlet's `service()` method concurrently.

Put simply, every client request is handled serially. No data is being shared between clients and, therefore, there is no need for the servlet to worry about synchronization. A container will typically ensure this behavior by instantiating a new servlet object for each client that accesses the servlet.

- A Servlet tags itself as using the single thread model by implementing the `javax.servlet.SingleThreadModel` interface.

NOTE

The `SingleThreadModel` interface does not contain any methods and therefore does not require any other changes to a servlet class. It is a tagging interface that lets the container know you want instances of this servlet class to use the single thread model.

The following servlet demonstrates implementing the `SingleThreadModel` interface.

```
1  package com.best;
2
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  import java.io.*;
6  import java.util.*;
7  import com.best.db.*;
8  import com.best.exception.*;
9
10 public class BrowseServlet extends HttpServlet
11         implements SingleThreadModel
12 {
13     private InventoryDB invDB;
14
15     public void init() throws ServletException
16     {
17         invDB = (InventoryDB)
18             getServletContext().getAttribute("inventoryDB");
19     }
20
21     public void doGet(HttpServletRequest req,
22                       HttpServletResponse res)
23         throws ServletException, IOException
24     {
25         //body of doGet()
26     }
27 }
```

FIGURE 4. *A servlet implementing the `SingleThreadModel` interface.*

- The body of the class does not change, just the declaration stating that the `SingleThreadModel` is to be implemented.

Cleanly Destroying a Servlet

If you have a servlet that uses the default threading model, then there can be any number of threads executing its `service()` method at the same time. If a `service()` method takes a long time to execute, it is possible that the servlet may time-out before all requests are completed.

The following `LongRunningDemo` servlet demonstrates a servlet whose `service()` method takes at least 25 seconds to execute. The class keeps track of how many threads currently running the `service()` method for this particular instance.

This example demonstrates both how to cleanly destroy a servlet and also how to keep an accurate count of how many clients are currently accessing the servlet.

```
1  import javax.servlet.*;
2  import javax.servlet.http.*;
3  import java.io.*;
4
5  public class LongRunningDemo extends HttpServlet
6  {
7      private int counter;
8      private boolean shuttingDown;
9
10     public void init()
11     {
12         shuttingDown = false;
13         counter = 0;
14     }
15
16     public void destroy()
17     {
18         System.out.println("Destroying servlet...");
19         setShuttingDown(true);
20         while(getCounter() > 0)
21         {
22             System.out.println("destroy() is waiting...");
23             this.sleep(1000);
24         }
25     }
26
27     private void sleep(int time)
28     {
29         try
30         {
31             Thread.sleep(time);
32         }catch(InterruptedException e)
```

```
33         {}
34     }
35
36     public void doGet(HttpServletRequest req,
37                       HttpServletResponse res)
38         throws ServletException, IOException
39     {
40         for(int i = 0; i < 5 && !isShuttingDown(); i++)
41         {
42             System.out.println("loop " + i);
43             this.sleep(5000);
44         }
45
46         res.setContentType("text/html");
47         PrintWriter out = res.getWriter();
48
49         out.println("<html><body>");
50         out.println("<p>Sorry that took so long</p>");
51         out.println("<p>There are " + getCounter() + " threads
left</p>");
52         out.println("</body></html>");
53     }
54
55     private synchronized void enterService()
56     {
57         counter++;
58     }
59
60     private synchronized void leaveService()
61     {
62         counter--;
63     }
64
65     private synchronized int getCounter()
66     {
67         return counter;
68     }
69
70     protected synchronized void setShuttingDown(boolean flag)
71     {
72         shuttingDown = flag;
73     }
74
75     protected synchronized boolean isShuttingDown()
76     {
77         return shuttingDown;
78     }
79
80     protected void service(HttpServletRequest req,
```

```
81         HttpServletResponse resp)
82         throws ServletException, IOException
83     {
84         enterService();
85         try
86         {
87             super.service(req, resp);
88         }
89         finally
90         {
91             leaveService();
92         }
93     }
94 }
```

FIGURE 5. *The LongRunningDemo servlet has a doGet() method that takes a long time to complete.*

NOTE

If you want to see how this servlet works in action, open four or five browsers windows and enter the URL:

`http://localhost:8080/demo/longtime`

Then open the Tomcat Manager in another browser window:

`http://localhost:8080/manager/html`

Stop the /demo application by clicking the “Stop” link next to /demo. Look at the standard output in the console window of the Tomcat program. It should look similar to Figure 6.

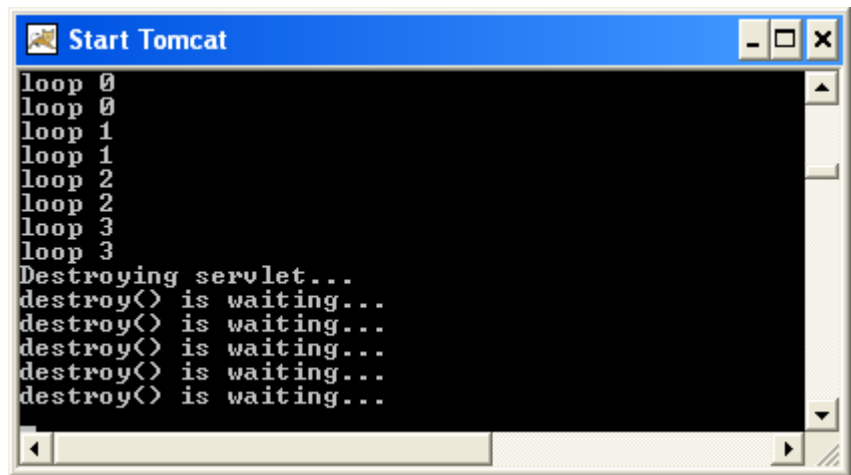


FIGURE 6. *The `destroy()` method waited several seconds for the `doGet()` methods to run to completion.*

Servlet Listeners

You can define a listener that is notified of events that occur during a servlet's lifecycle. Java's delegation model for event handling is used, where a listener interface is implemented and an event object is passed in to each listener method.

The listener interfaces and their corresponding events include:

- `ServletContextListener` and `ServletContextEvent` - for listening to servlet context events. More specifically, a listener is notified of when the context has just been initialized and also when it is about to be destroyed. You can use the event object to obtain a reference to the `ServletContext`.
- `ServletContextAttributeListener` and `ServletContextAttributeEvent` - for listening to when an attribute is added, removed, or replaced in the servlet context. You can use the event object to determine the name and value of the affected attribute.
- `HttpSessionListener` and `HttpSessionEvent` - for listening to when a session has been created or destroyed. The event object can be used to obtain a reference to the affected session object.
- `HttpSessionAttributeListener` and `HttpSessionBindingEvent` - for listening to when an attribute is added, removed, or replaced in a session object. You can use the event object to obtain a reference to the affected session object and also the name and value of the affected attribute.
- `HttpSessionActivationListener` and `HttpSessionEvent` - for listening to when a session has just been activated or is about to be destroyed. The event object can be used to obtain a reference to the affected session object.
- `HttpSessionBindingListener` and `HttpSessionBindingEvent` - an object can be a listener for when it gets added or removed to a session object. The object can use the event object to a reference to the session object that is was just added to or removed from.

The Delegation Model

If you are not familiar with Java event handling, Java uses the delegation model. This means a listener of an event must implement an agreed upon interface and register itself with the source of the event. When an event occurs, the source of the event notifies the listener by invoking one of the methods in the interface.

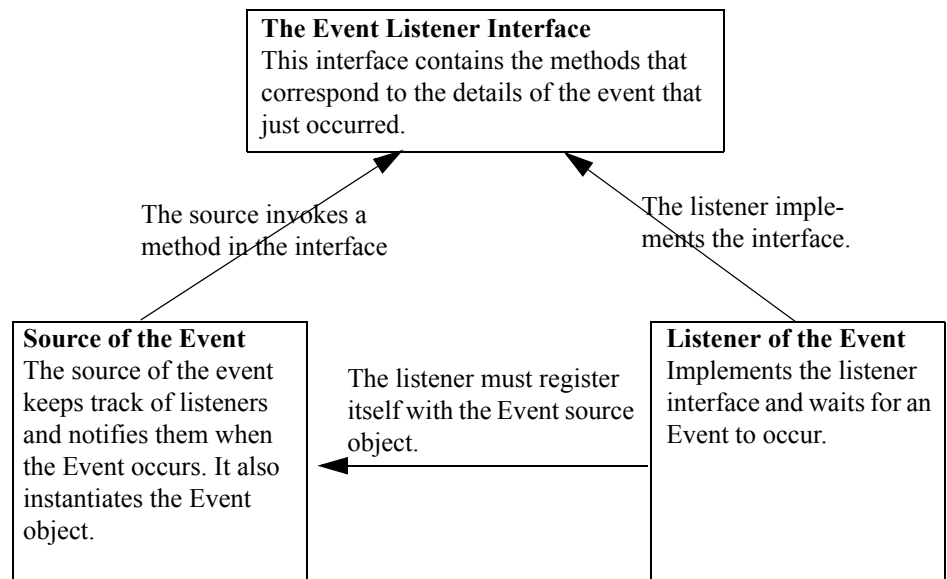


FIGURE 7. *Java's delegation model for event handling.*

An Example of a Session Counter

To demonstrate writing a servlet listener, we will look at a class that keeps track of how many sessions are being created and destroyed on a Web site. This is useful information, as you may use it to determine the time-out value of a session to increase the performance of your Web server.

The following `MySessionCounter` class is a JavaBean that implements the `HttpSessionListener` interface. This means this bean will be notified whenever a session is created and whenever a session is destroyed. It keeps track of the total number of sessions created, the total number of currently active sessions, and also the maximum number of sessions that were active at any point.

```
1  package my.listeners;
2
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5
6  public class MySessionCounter implements HttpSessionListener
7  {
8      private int total;
9      private int active;
10     private int max;
11     private boolean first = true;
12
13     public void sessionCreated(HttpSessionEvent event)
14     {
15         total++;
16         active++;
17         if (active > max)
18         {
19             //update max
20             max = active;
21         }
22
23         if (first)
24         {
25             ServletContext context =
event.getSession().getServletContext();
26             context.setAttribute("sessionCounters", this);
27             first = false;
28         }
29     }
30 }
```

```
31     public void sessionDestroyed(HttpSessionEvent event)
32     {
33         active--;
34     }
35
36     public int getTotal()
37     {
38         return total;
39     }
40
41     public int getActive()
42     {
43         return active;
44     }
45
46     public int getMaximum()
47     {
48         return max;
49     }
50 }
```

FIGURE 8. *The MySessionListener class.*

- Notice an instance of this class is an attribute of the servlet context. This gives it “application” scope, allowing any servlet or JSP in the container to access this bean.

Registering a Listener

Once you have written a class that implements the listener interface you want to handle events from, you need to register the listener class with your Web application. This is done by modifying the deployment descriptor, adding the following entry:

```
<listener>
  <listener-class>class name</listener-class>
</listener>
```

The following web.xml file shows the my.listeners.MySessionCounter registered with the /demo application:

```
1  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN"
2    "http://java.sun.com/dtd/web-app_2_3.dtd">
3
4  <web-app>
5    <display-name>Demo Web Application</display-name>
6
7    <listener>
8      <listener-class>
9        my.listeners.MySessionCounter
10      </listener-class>
11    </listener>
12
13    <servlet>
14      <servlet-name>longtime</servlet-name>
15      <servlet-class>LongRunningDemo</servlet-class>
16    </servlet>
17
18    <servlet-mapping>
19      <servlet-name>longtime</servlet-name>
20      <url-pattern>/longtime</url-pattern>
21    </servlet-mapping>
22
23    <session-config>
24      <session-timeout>1</session-timeout>
25    </session-config>
26  </web-app>
```

FIGURE 9. Add the `<listener>` element to the web.xml file.

Figure 10 shows the output of a JSP page written to display the information in the `MySessionCounter` listener object registered with the `/demo` application.

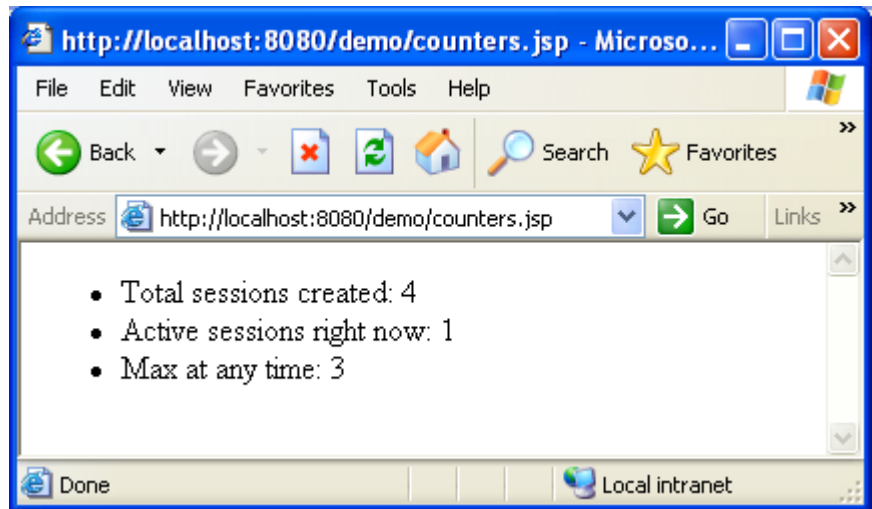


FIGURE 10. A JSP page that displays the various counters of the *MySessionCounter* bean.

Using Filters

A filter is an object that can transform the header and content (or both) of a request or response. A filter “intercepts” a request from a client, allowing the request to be pre-processed before being passed on to its intended recipient.

Examples of using filters include:

- Authentication.
- Logging purposes
- XML transformations
- Data encryption
- Data compression

A filter is a part of a chain referred to as the *filter chain*. It is not the job of a filter to generate a response of any type. The filter is handed a request, can modify aspects of the request and response, then passes the request/response to the next filter in the chain.

To write a filter for your Web application, perform the following steps:

- Write a class that implements the `javax.servlet.Filter` interface.
- Implement the `doFilter()` method, which takes in the request and response.
- Perform whatever decision making or tasks on the request and response.
- When processing is complete, the filter either passes the request and response to the next filter in the chain (or its final destination), OR it does not pass along this information, thereby blocking the request from being processed.
- Deploy the Filter object by declaring it in the deployment descriptor.

The Filter Interface

The first step in creating a filter is to write a class that implements the `javax.servlet.Filter` interface. The `Filter` interface contains three methods:

```
public void init(FilterConfig fc)
```

Invoked by the Web container when the filter is being placed into service. Typically, your `init()` method stores the given `FilterConfig` in a field of the class.

```
public void destroy()
```

Invoked by the Web container when the filter is being taken out of service.

```
public void doFilter(ServletRequest request, ServletResponse  
                    response, FilterChain chain)  
                    throws IOException, ServletException
```

Invoked when a request needs to pass through this filter. The chain is used to pass the request and response on to the next intended recipient.

An Example of a Filter Class

The following `TimerFilter` class records the time in milliseconds from when a request goes through the filter and a response is sent back. This is roughly the number of milliseconds that it took to process a request, which is quite useful for profiling your Web applications.

```
1  package my.filters;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  public class TimerFilter implements Filter
8  {
9      private FilterConfig config;
10
11     public void init(FilterConfig config)
12         throws ServletException
13     {
14         this.config = config;
15     }
16
17     public void destroy()
18     {
19         config = null;
20     }
21
22     public void doFilter( ServletRequest request,
23                         ServletResponse response,
24                         FilterChain chain)
25         throws IOException, ServletException
26     {
27         long before = System.currentTimeMillis();
28         chain.doFilter(request, response);
29         long after = System.currentTimeMillis();
30
31         System.out.println(request.getRemoteHost() +
32                             " was here " + (after - before) + " milliseconds");
33     }
34 }
```

FIGURE 11. *The `TimerFilter` records the time a request takes in milliseconds.*

A few comments about the TimerFilter class:

- The FilterConfig field is initialized in the init() method and freed in the destroy() method, which is typical of filter classes. This was done here for demonstration only, because TimerFilter did not use the FilterConfig object.
- A filter uses the doChain() method to pass the request and response to the next resource in the chain. The doChain() method is invoked on the FilterChain argument.
- The current time is obtained before and after doChain(), and the difference is displayed using System.out. For profiling, you could log this information to a file instead.

Deploying a Filter

A filter is defined in the Web application deployment descriptor using the `<filter>` element. This element requires the name and class name of the filter. For example:

```
<filter>
  <filter-name>timer</filter-name>
  <filter-class>my.filters.TimerFilter</filter-class>
</filter>
```

The next step is to create one or more mappings to denote which requests get filtered by the “timer” filter, which is done using the `<filter-mapping>` element:

```
<filter-mapping>
  <filter-name>timer</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The mapping above is a “catch all” for any request. Other examples might include:

```
<filter-mapping>
  <filter-name>timer</filter-name>
  <url-pattern>/* .jsp</url-pattern>
</filter-mapping>
```

for all JSP pages. Similarly, “*.html” and another for “*.htm” for all HTML pages.

NOTE

The `<filter>` and `<filter-mapping>` appear after the `<description>` element and before the `<listener>` elements in the `web.xml` file.

Figure 12 shows the console window for Tomcat, which is where the output of the TimerFilter is sent.

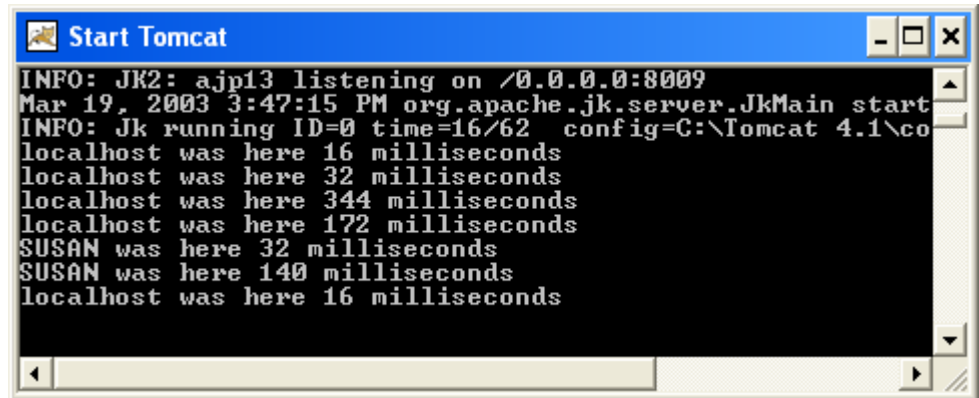


FIGURE 12. *The output generated by the TimerFilter.*

Lab 1: Writing a Servlet Listener

Objective: In this lab, you will write a class that listens to attribute changes of the session objects in the container. More specifically, you will log when a shopping cart is added to a session and when a shopping cart is removed from a session. When a shopping cart is removed, you will print out the items that were in the cart.

Perform the following steps:

1. Write a class named `ShoppingCartListener` that implements the `javax.servlet.http.HttpSessionAttributeListener` interface. Declare the class in the `my.listeners` package and save the file your `\best\WEB-INF\classes` folder.
2. Declare the `attributeAdded()` method. Within this method, if the attribute added to the session object is named "cart", then print out "Adding a new shopping cart" using `System.out.println()`. If the attribute is named something else, don't do anything.
3. Declare the `attributeRemoved()` method. Within this method, if the attribute removed from the session object is named "cart", then print out "Removing a shopping cart" using `System.out.println()`. Also, use the `getItems()` method of `ShoppingCart` to obtain the `ArrayList` of items in the cart. Write a loop to print out each of these items.
4. Declare the `attributeReplaced()` method with an empty method body.
5. Save and compile your `ShoppingCartListener` class.
6. Modify your `web.xml` file and add the `<listener>` element for your `ShoppingCartListener` class.
7. In addition, add the following entry in your deployment descriptor (after the `<servlet-mapping>` and before the `<resource-ref>` entries). This will make your sessions time-out in one minute.

```
<session-config>  
  <session-timeout>1</session-timeout>
```

```
</session-config>
```

8. Restart Tomcat and open the `http://localhost:8080/best/welcome` page in your browser. This will cause a new shopping cart to be added to a new session.
9. Close your browser, which will start the time-out of your session.

What you should see: In the Tomcat window, you should see the “Adding a new shopping cart” statement. Wait a minute and you should see the “Removing a shopping cart” statement, similar to Figure 13. Since we do not have the mechanism setup to add items to the cart yet, that is all that will be displayed.

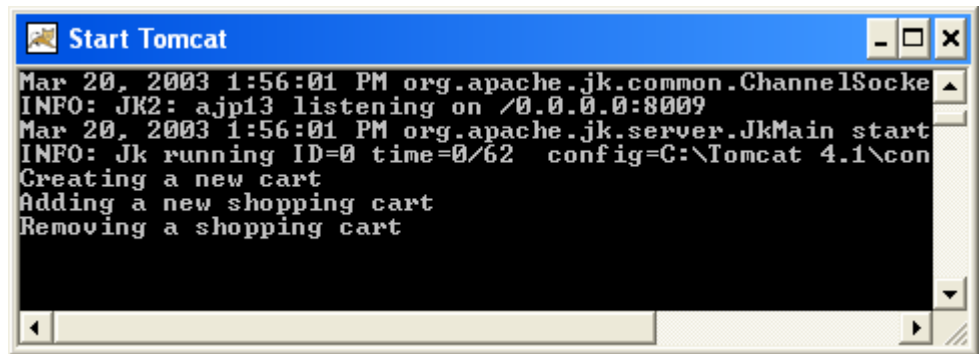


FIGURE 13. *The output of the ShoppingCartListener.*



An Introduction to JavaServer Pages

JavaServer Pages provide for the creation of dynamic Web pages, using the power of the Java programming language in a syntax similar to HTML. This chapter discusses how JSPs work behind-the-scene and how to develop and deploy them. Topics discussed in this chapter include:

- an overview of JavaServer Pages
- a comparison of JSP to Servlets
- the JSP engine
- the lifecycle of a JSP page
- writing and deploying a JSP page
- the JSP tags
- the implicit objects
- scriptlets
- JSP actions

Overview of JavaServer Pages

JavaServer Pages are HTML documents that can contain code from the Java programming language. This allows the HTML document to go from static to dynamic, since a JSP is not simply returned to the client. Instead, the page is "executed" in a Java object called a servlet.

Advantages using JavaServer Pages include:

- **Java:** JavaServer Pages use the Java programming language.
- **Portable:** Like Java, JavaServer Pages are platform-independent and do not rely on any native code to execute. All you need is a server that supports JSP.
- **Easy to learn:** The design of JavaServer Pages makes them usable by Web developers who have minimal programming skills but are well-versed in the areas of Web site development.
- **Customizable:** You can create your own tags in JSP, allowing you to separate tasks from the visual presentation.
- **Automatic compilation:** When you make a change to a JSP, the Web server automatically compiles the JSP for you, making them much easier to modify and deploy than a servlet.
- **Separate content from the view:** By using JavaBeans, the view of a JSP is separate from the content

JSP or Servlets?

JavaServer Pages are generated into Servlets, Java objects that execute within a container of the Web server. This means a JSP page can perform any task a Servlet can, although servlets and JSP pages are typically used for different reasons.

The question now becomes: What do I use, JSP or servlets?

The answer is simply: both!

- Use servlets as an extension of your Web server technology, including creating controllers or performing tasks like authenticating users, validating data input, and handling sessions.
- Use JavaServer Pages for presenting the content to the client.

In particular, JavaServer Pages are used to present the View of data in the MVC model, while servlets are often used as Controllers.

JavaServer Pages and Servlets

You may be asking yourself how an HTML document with Java code scattered throughout it gets executed on a Web server. The answer is simple: the Web server generates a servlet using the syntax of your JavaServer Page.

NOTE

When JSP first evolved, it was promoted as a different technique for creating a Java servlet. This is not the case anymore, as JSP pages and servlets are typically used in conjunction with each other.

- When a JavaServer Page is initially accessed by a client, the JSP engine writes, compiles, and instantiates a servlet object that contains the Java code of the JSP.
- The developer of the JSP does not need to be a Java programmer. Creating JavaServer Pages only requires an introductory understanding of object-oriented programming and some familiarity with the syntax of Java.

The design of JSP is basically the opposite concept of the design of Servlets:

- A servlet is Java code that can be used to generate HTML documents.
- JavaServer Pages are HTML documents that can contain Java code.

Notes

The Lifecycle of a JSP Page

A JavaServer Page is a text file similar to an HTML document. In fact, JavaServer Pages look much like an HTML document since a JSP can use any of the tags of HTML.

The following is a list of the some of the stages in the lifecycle of a JavaServer Page:

- A JavaServer Page is a text file that is saved using the *.jsp format.
- A JavaServer Page is deployed on a Web server just like any HTML document would be deployed.
- Before a JavaServer Page can be accessed by a client, it must go through what is called the translation phase.
- The JSP engine is the term used to describe that portion of the Web server application that provides the services of the JSP architecture.
- The JSP engine translates a JavaServer Page into a Java servlet the first time it is accessed.
- The `jspInit()` method is invoked on the JSP implementation servlet.
- Each time a request is made for a JSP, the JSP Engine invokes the `_jspService()` method on the JSP implementation servlet.
- The `jspDestroy()` method is invoked when the JSP Engines removes the servlet from the container.

NOTE

Subsequent client requests for a JSP do not need require translation. The Web server uses a time-stamp to determine when changes were made to the JSP and a new translation is required.

- As you might expect, it is a good idea to test your JSPs before making them available to clients. You will most likely be the first client to access the JSP when deploying and debugging the JSP. Most problems with a JSP will typically arise during the translation phase.

The following diagram illustrates the lifecycle of a typical JavaServer Page.

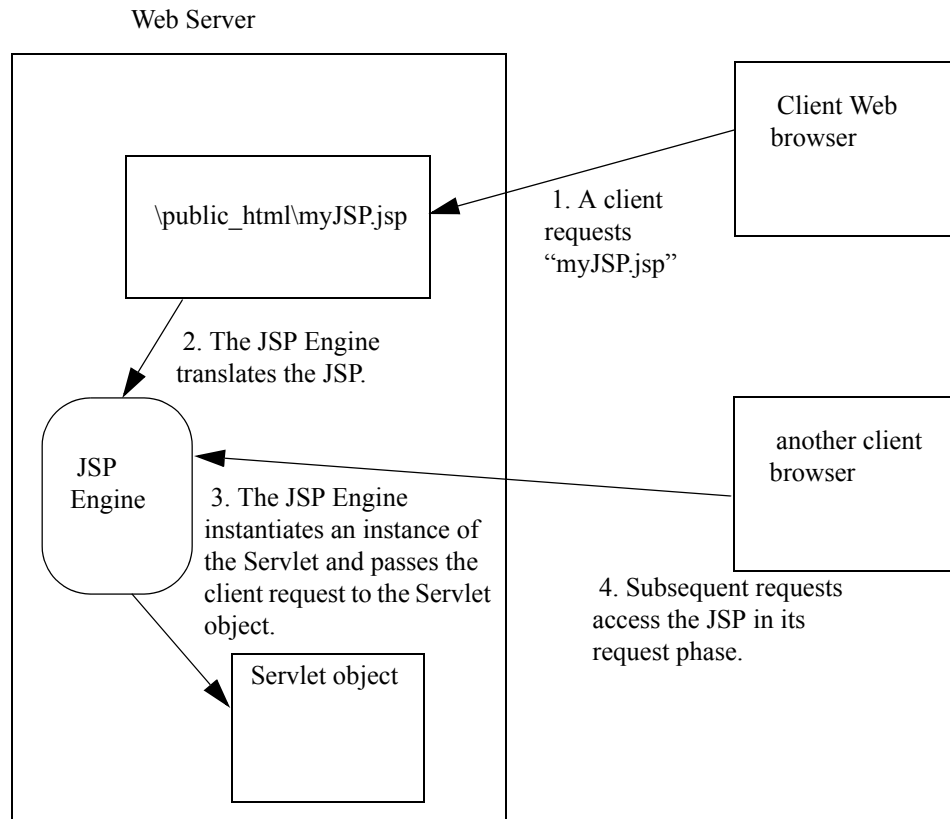


FIGURE 1. *The lifecycle of a JavaServer Page*

A Simple JavaServer Page

The following is a simple JavaServer Page that displays a greeting and also the current time on the server. It is a text file saved as "HelloWorld.jsp".

```
1  <%-- Filename: "HelloWorld.jsp"  --%>
2  <%-- Created: Jan 1, 2003    --%>
3
4  <%@ page import="java.util.Date" %>
5
6  <HTML>
7      <HEAD>
8          <TITLE>
9              HelloWorld JSP
10         </TITLE>
11     </HEAD>
12
13     <BODY>
14         <H2>Welcome to JavaServer Pages!</H2>
15         <P>
16             Time: <%= new Date() %>
17         </P>
18     </BODY>
19 </HTML>
```

FIGURE 2. The "HelloWorld.jsp" file demonstrates a simple JavaServer Page.

Some comments about the HelloWorld.jsp page:

- It uses three types of JSP tags: the comment tag, directive tag, and expression tag.
- The first two lines of text are comments, which are generated using the `<%--` and `--%>` tags.
- The directive tag looks like `<%@ ... %>`.
- The expression tag looks like `<%= ... %>`. An expression in a JSP gets evaluated and sent back to the client as HTML text.

We will discuss each of these types of tags (and others) in detail.

Deploying a JSP Page

One nice feature about JavaServer Pages is that are easy to deploy and easy to modify.

- To deploy a JavaServer Page on Tomcat, you simply copy the .jsp file into the %CATALINA_HOME%\webapps\ROOT folder.
- You can also copy it into one of your Web context folders in %CATALINA_HOME%\webapps.

If “HelloWorld.jsp” is copied into the %CATALINA_HOME%\webapps\ROOT directory, then the URL to access the page is:

```
http://localhost:8080/HelloWorld.jsp
```

Figure 3 shows the “HelloWorld.jsp” JavaServer Page viewed in a Web browser:



FIGURE 3. *The Web page generated by “HelloWorld.jsp”.*

NOTE

If you modify the file "HelloWorld.jsp", then the JSP Engine will know about the changes and update the implementation servlet automatically.

Figure 4 shows the page after a new line of text was added to the "HelloWorld.jsp" file.



FIGURE 4. *Modifying the .jsp file automatically updates the page.*

JavaServer Page Tags

JavaServer Pages provide for Java code to be embedded within HTML. As with HTML, JSPs use tags. JSP tags have the form:

```
<% JSP tag %>
```

There are five different types of JSP tags:

- **Declarations:** for declaring variables
- **Expressions:** contain Java code whose results are made into strings and returned with the client response.
- **Scriptlets:** Java code statements that are executed on the server.
- **Directives:** for declaring attributes pertaining to the servlet associated with the JSP.
- **Comments**

Each of the tags above uses the `<% %>` syntax. However, since each type of JSP tag causes a different result, there is a special syntax to distinguish them.

TABLE 1. The syntax for the various JSP tags.

JSP Tag	Syntax	Example
comment	<code><%-- --%></code>	<code><%-- Author: Mark Twain --%></code>
declaration	<code><%! %></code>	<code><%! int x = 10; %></code>
expression	<code><%= %></code>	<code><%= x * 3 %></code>
scriptlet	<code><% %></code>	<code><% if (x >= 100) %></code>
directive	<code><%@ %></code>	<code><%@ include file="top.jsp" %></code>

There is also another type of tag in JSP to denote various directives like forwarding or using JavaBeans. The syntax for these JSP directive tags looks like:

```
<jsp:directive />
```

where directive is the task to be performed.

Declarations

JavaServer Pages use the `<%! %>` tags for declaring variables and methods for a JSP. Declaring a variable involves stating the data type and assigning the variable a name. Declaring a method involves denoting the signature of the method and defining the method as with any Java class.

Be aware of the following when using declarations:

- You must declare the variable or method before attempting to use it.
- If the item being declared is a variable, then the variable will become a member variable of the corresponding servlet class.
- If the item being declared is a method, then the method will become a method of the corresponding servlet class.
- Declared items are said to have servlet scope or page scope, which makes sense since they end up being members of the servlet.
- Declarations of variables are executed only one time, when the corresponding servlet object is instantiated.

The following DeclareDemo.jsp page in Figure 5 declares a variable of type `java.sql.Connection` and initializes the field in a declared method called `jspInit()`.

```
1  <!-- A demonstration of the declaration tags -->
2  <!-- Filename: "DeclareDemo.jsp" -->
3
4  <%@ page import="java.sql.*, javax.naming.*" %>
5  <%! Connection connection; %>
6
7  <%! public void jspInit()
8      {
9          try
10         {
11             Class.forName("org.gjt.mm.mysql.Driver");
12             connection =
13             DriverManager.getConnection("jdbc:mysql://localhost:3306/best");
14         } catch (Exception e)
15         {
16             System.out.println(e);
17         }
18     }
19
20 <HTML>
21     <BODY>
22         <p>
23             You are connected to <%= connection.getCatalog() %>
24         </p>
25     </BODY>
26 </HTML>
```

FIGURE 5. *An example of declaring a variable and method within a JSP.*

The Generated Servlet Class

To demonstrate how declarations work, the following code in Figure 6 shows part of the servlet class generated by Tomcat for the DeclareDemo.jsp page in Figure 5.

Notice the class contains a field named “connection” of type Connection and also contains the jspInit() method just as it was defined in DeclareDemo.jsp.

NOTE

To view the entire source code, look in the directory:

%CATALINA_HOME%\work\Standalone\localhost__

Tomcat places all generated servlets in a subfolder of the \work directory.

```
1  package org.apache.jsp;
2
3  import java.sql.*;
4  import javax.naming.*;
5
6  public class DeclareDemo_jsp extends HttpJspBase {
7
8      Connection connection;
9      public void jspInit()
10     {
11         try
12         {
13             Class.forName("org.gjt.mm.mysql.Driver");
14             connection =
15 DriverManager.getConnection("jdbc:mysql://localhost:3306/best");
16         }catch(Exception e)
17         {
18             System.out.println(e);
19         }
20     }
21
22     public void _jspService( HttpServletRequest request,
23                             HttpServletResponse response)
24         throws java.io.IOException, ServletException {
25
26         JspWriter _jspx_out = null;
27
28         try {
29             response.setContentType("text/html;charset=ISO-8859-1");
30             out = pageContext.getOut();
31             _jspx_out = out;
32
33             out.write("<HTML> \r\n \t");
```



```
34         out.write("<BODY>\r\n \t\t");
35         out.write("<p>\r\n \t\t You are connected to ");
36         out.print( connection.getCatalog() );
37         out.write("</p>\r\n \t");
38         out.write("</BODY>\r\n");
39         out.write("</HTML>");
40     } catch (Throwable t) {}
41 }
42 }
43
```

FIGURE 6. *The servlet class generated by Tomcat for the DeclareDemo.jsp page.*

Figure 7 shows the output of the DeclareDemo.jsp page in a browser.

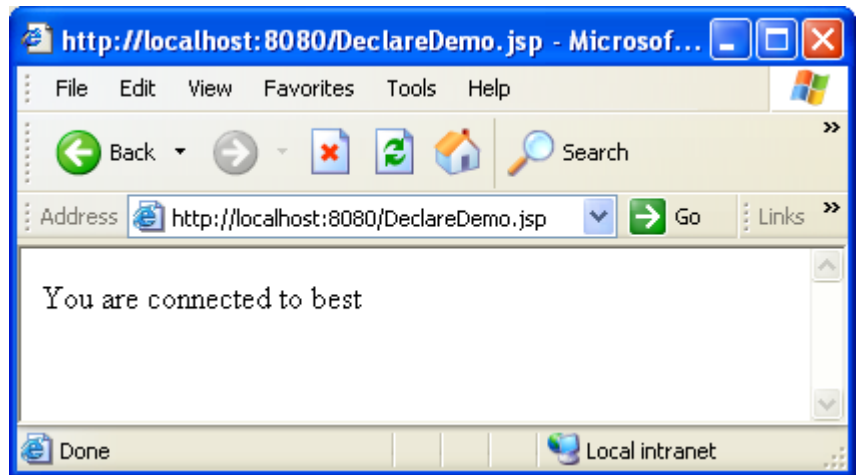


FIGURE 7. *The output of the DeclareDemo.jsp page.*

Expressions

JavaServer Pages can contain expressions, Java statements that are evaluated and returned to the client as part of the body of the response. Expressions are created using the `<%= %>` tags, and are useful for displaying the contents of a variable or the return value of a method.

For example,

```
1  <%-- Filename: ExpressionDemo.jsp --%>
2
3  <%! int counter = 0; %>
4  <HTML>
5      <BODY>
6          <H2>Welcome</H2>
7          <P>
8              This page has been accessed <%= ++counter %> times.
9          </P>
10     </BODY>
11 </HTML>
```

FIGURE 8. *A JSP using an expression tag.*

In the JSP in Figure 8, the counter variable is incremented within the expression tag. In addition, since it is an expression, the result is sent back to the client.

The Java code in the implementing servlet looks like:

```
out.write("<P>\r\n \t\tThis page has been accessed ");
out.print( ++counter );
out.write(" times.\r\n \t\t");
out.write("</P> \r\n \t");
```

- Notice the servlet outputs the HTML preceding an expression, then sends back the result of the expression, then sends back the HTML following the expression. This is a logical technique since the HTML is static data, but the expression must be evaluated with each client request.

Figure 9 shows the output of the ExpressionDemo.jsp page after it has been accessed four times.

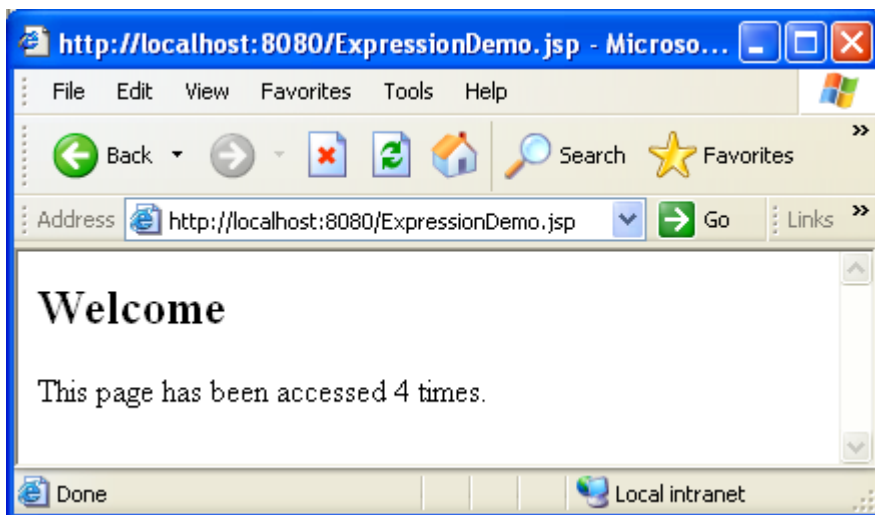


FIGURE 9. *The output of the ExpressionDemo.jsp page.*

NOTE

The counter in the ExpressionDemo is incremented inside the `_jspService()` method of the implementing servlet. It keeps track of how many times this method is invoked on this particular instance of the servlet.

In other words, it is not a page counter in the sense of keeping track of the number of visitors to the ExpressionDemo.jsp page. If the servlet times-out, is destroyed, or is reloaded, then the counter will be reset to 0.

In addition, your servlet container might create a pool of several servlet instances, and each instance would have their own counter.

Implicit Objects

JavaServer Pages contain a collection of implicit objects that can be used within the JSP tags. Each implicit object maps to a corresponding interface of the Servlet API. You do not instantiate an implicit object to use on a page; it is available and can be used by simply referring to the object by its name.

The following is a list of the implicit objects available to a JSP and also the data type that each object maps to.

- **request:** an `HttpServletRequest` object that represents the client request.
- **response:** an `HttpServletResponse` object that represents the response sent to the client.
- **out:** a `javax.servlet.jsp.JspWriter` object that represents the output stream of the response.
- **config:** the `javax.servlet.ServletConfig` object for this page.
- **pageContext:** a `javax.servlet.jsp.PageContext` object representing the page context for this JSP.
- **application:** a `javax.servlet.ServletContext` object representing the servlet context for this JSP.
- **page:** a `java.lang.Object` analogous to the `this` reference.
- **session:** an `HttpSession` object for managing a client session.
- **exception:** a `java.lang.Throwable` object that is only accessible to JSPs using the page directive `isErrorPage="true"`. The object represents the uncaught exception that caused this page to occur.

The following code in Figure 10 shows part of the implementing servlet of the ExpressionDemo.jsp page. Notice that within the `_jspService()` method, the implicit objects are each initialized to their appropriate value.

```
1      public void _jspService(HttpServletRequest request,
2                          HttpServletResponse response)
3          throws java.io.IOException, ServletException {
4          JspFactory _jspxFactory = null;
5          javax.servlet.jsp.PageContext pageContext = null;
6          HttpSession session = null;
7          ServletContext application = null;
8          ServletConfig config = null;
9          JspWriter out = null;
10         Object page = this;
11         JspWriter _jspx_out = null;
12
13         try {
14             _jspxFactory = JspFactory.getDefaultFactory();
15             response.setContentType("text/html;charset=ISO-8859-1");
16             pageContext = _jspxFactory.getPageContext(this,
17                 request, response, null, true, 8192, true);
18             application = pageContext.getServletContext();
19             config = pageContext.getServletConfig();
20             session = pageContext.getSession();
21             out = pageContext.getOut();
22             _jspx_out = out;
23
24             //the body of the method determined by the JSP page goes here
25
26         } catch (Throwable t) {
27             out = _jspx_out;
28             if (out != null && out.getBufferSize() != 0)
29                 out.clearBuffer();
30             if (pageContext != null)
31                 pageContext.handlePageException(t);
32             finally {
33                 if (_jspxFactory != null)
34                     _jspxFactory.releasePageContext(pageContext);
35             }
36         }
37     }
```

FIGURE 10. *The implementing servlet initializes each of the implicit objects.*

NOTE

Each Web server uses their own unique way of initializing the implicit objects. This example demonstrates how it's done by Tomcat.

The request Object

When a client accesses a URL using HTTP, the client's browser generates a header of information referred to as the request header. This contains various information about the request, like the HTTP method, the file requested, the location where the request originated, and so on.

The following is an example of a request header:

```
GET /counter.jsp HTTP/1.1
Connection: close
Host: 127.0.0.1
Referer: http://www.yahoo.com/
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 95)
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-us
```

A JavaServer Page can obtain the information found in the request header by using the implicit JSP object named request.

The request object is instantiated by the Web server using the information in the request header. The request object is a Java object of type `javax.servlet.http.HttpServletRequest`, and the JSP can invoke any of the methods in this interface by simply referencing the request object.

The Methods of the Request Object

`HttpServletRequest` is an interface that contains various methods for retrieving information about the HTTP request. Some of the methods include:

```
public String getHeader(String name)
```

returns the requested header information. The header names can be retrieved using the `getHeaderNames()` method of the request object.

```
public String getMethod()
```

returns the HTTP method of the request, e.g. GET or PUT

```
public HttpSession getSession(boolean create)
```

creates a session object for this client. If `create` is true, a new one will be created if one does not exist. If `create` is false, then null will be returned if a session object is not found for this client.

```
public Cookie [] getCookies()
```

returns any `Cookie` objects sent with the request.

```
public RequestDispatcher getRequestDispatcher(String path)
```

returns a reference to the request dispatcher of another resource of the Web server, typically another servlet or JavaServer Page. You use the request dispatcher to include another resource or forward the request to another resource.

```
public void setAttribute(String name, Object x)
```

adds the given `Object` to the request. It can be retrieved later using the `name` argument and the `getAttribute()` method. Objects stored in the request object are said to have request scope.

```
public Object getAttribute(String name)
```

returns a reference to the attribute matching the name.

```
public String getParameter(String name)
```

used for obtaining the parameters of the request.

Parameters

A client request can contain parameters, string values that appear in (name, value) pairs. The parameters can be added to the request in various ways, but typically appear in HTML forms.

If the HTTP method of the request is GET, then all parameter information will appear in the Servlet's URL appearing after a ?. For example,

```
http://localhost:8080/RequestDemo.jsp?user=John+Elway
```

A JavaServer Page uses the `getParameter()` method of the request object to obtain the values of the parameters.

```
public String getParameter(String name)
```

returns the value of the parameter whose name is `name`. If no matching parameter is found, then the method returns `null`.

The request object can also be used to invoke the following method for determining the parameter names:

```
public String [] getParameters()
```

returns an array containing the names of all parameters sent with this client.

The `RequestDemo.jsp` page in Figure 11 contains a parameter whose name is "user". The value of this parameter is displayed using the `getParameter()` method of the implicit request object.

NOTE

If the "user" parameter is not defined, then the value returned by `getParameter()` will be `null`.


```
1  <!-- Filename: RequestDemo.jsp -->
2
3  <HTML>
4      <BODY>
5          <H2>RequestDemo</H2>
6          <p>Welcome, <%= request.getParameter("user") %> </p>
7          <p>You have made a <%= request.getMethod()%> request</p>
8          <ul>
9              <li>Host: <%= request.getHeader("host") %> </li>
10             <li>Referer: <%= request.getHeader("referer") %> </li>
11             <li>User-Agent: <%= request.getHeader("user-agent") %>
</li>
12             <li>Accept: <%= request.getHeader("accept") %> </li>
13             <li>Accept-Encoding: <%= request.getHeader("Accept-
encoding") %> </li>
14             <li>Accept-Language: <%= request.getHeader("Accept-
Language") %> </li>
15             <li>Connection: <%= request.getHeader("Connection") %>
</li>
16         </ul>
17     </BODY>
18 </HTML>
```

FIGURE 11. *The RequestDemo.jsp page demonstrates using the request object to get a parameter from the HTTP request.*

Figure 12 shows the output of the RequestDemo.jsp page when the “user” parameter is set equal to “John Elway”.

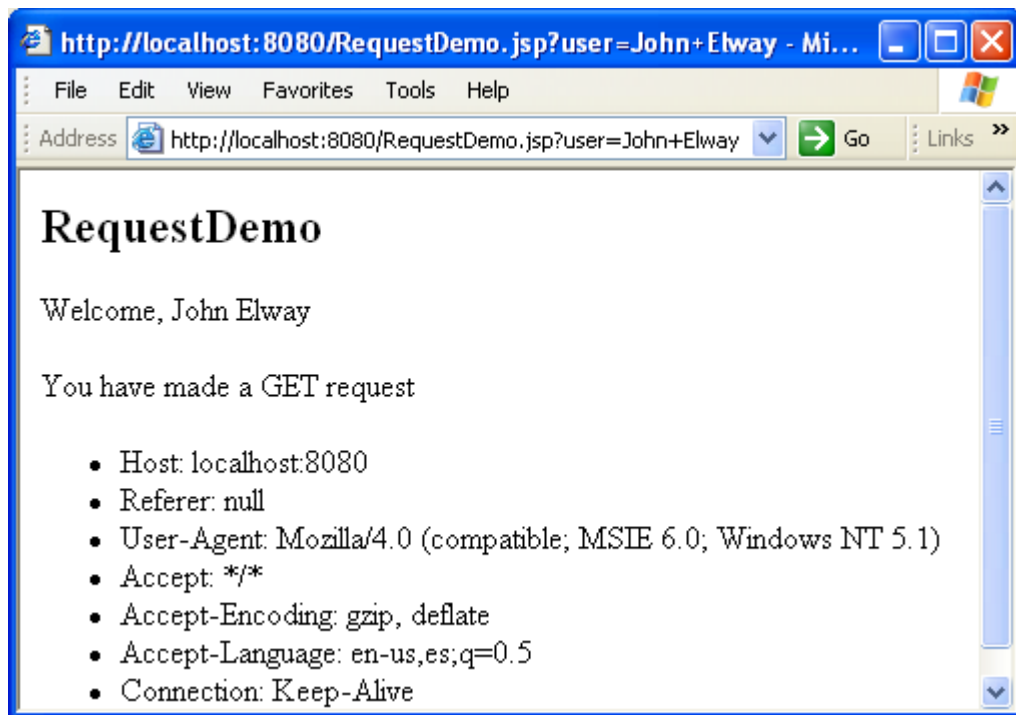


FIGURE 12. A sample output of the RequestDemo.jsp page.

Figure 13 shows the output when the “user” parameter is not defined in the URL.

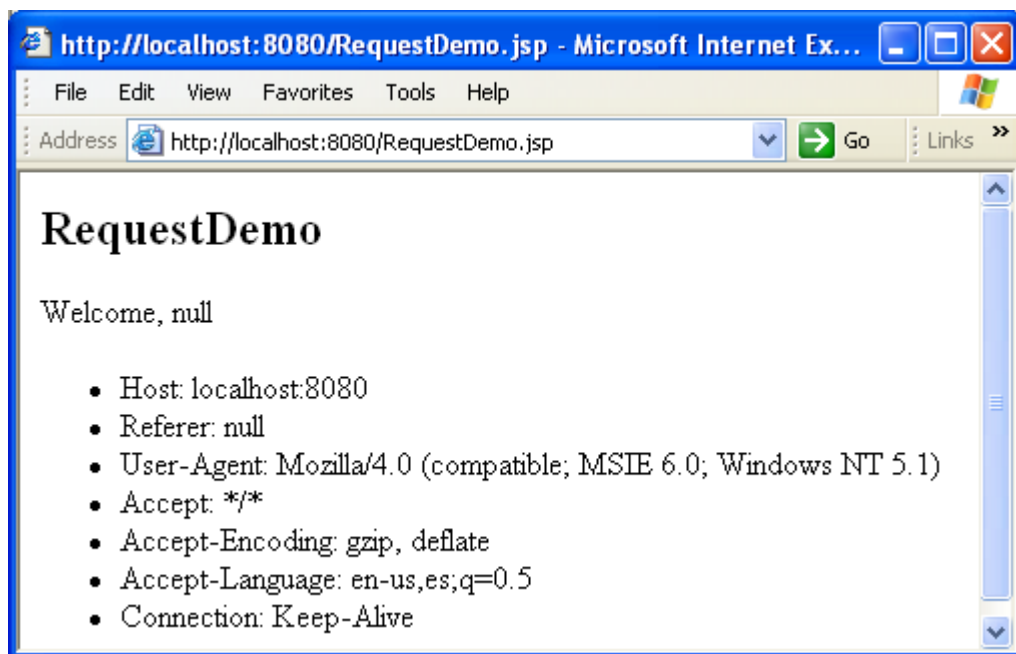


FIGURE 13. *The output of RequestDemo.jsp when "user" is not defined.*

Scriptlets

A scriptlet is a JSP tag that contains Java code. The code will end up in the `_jspService()` method of the corresponding servlet, which means that the scriptlet will be executed with each client request. Scriptlets appear in the `<% %>` tags.

For example, the following `DisplayHeading.jsp` page uses a scriptlet to determine if a parameter named “username” is defined in the request. If the parameter appears, then the client’s username is displayed.

```
1  <table width="75%" border="0">
2    <tr>
3      <td width="34%"></td>
5      <td width="66%">Welcome to BEST
6        <%! String username; %>
7        <% username = request.getParameter("username");
8          if(username != null) {
9            out.println(", " + username);
10           } %>
11      </td>
12    </tr>
13  </table>
```

FIGURE 14. *The DisplayHeading.jsp page.*

Figure 15 shows the code generated from Tomcat for the DisplayHeading.jsp page. Notice the scriptlet appears as Java code in the `_jspService()` method.

```
1      out = pageContext.getOut();
2
3      out.write("<table width=\"75%\" border=\"0\">\r\n  ");
4      out.write("<tr>\r\n    ");
5      out.write("<td width=\"34%\">");
6      out.write("<img src=\"best.gif\" width=\"190\" "
height=\"190\" align=\"middle\">");
7      out.write("</td>\r\n  ");
8      out.write("<td width=\"66%\">Welcome to BEST\r\n    \t");
9      out.write("\r\n    \t");
10     username = request.getParameter("username");
11     if(username != null)
12     {
13         out.println(", " + username);
14     }
15
16     out.write("\r\n  ");
17     out.write("</td>\r\n ");
18     out.write("</tr>\r\n");
19     out.write("</table>\r\n");
```

FIGURE 15. *Scriptlets appears as Java code in the `_jspService()` method of the implementing servlet class.*

See Figure 22 on page 179 for an example that uses the DisplayHeading.jsp page.

A JSP Using a Data Access Object

The following ScriptletDemo.jsp page demonstrates a JavaServer Page that uses scriptlets to communicate with a Data Access Object. The InventoryDB object is used to determine the items in the inventory table that match a given category. The category is denoted by a request parameter.

```
1  <!-- Filename: ScriptletDemo.jsp -->
2
3  <%@ page import="java.util.*, com.best.db.*" %>
4
5  <HTML>
6      <BODY>
7
8      <table>
9
10         <%
11             ResourceBundle text = (ResourceBundle)
session.getAttribute("text");
12             if(text == null)
13             {
14                 text = ResourceBundle.getBundle("text.Messages",
request.getLocale());
15             }
16             try
17             {
18                 InventoryDB invDB = new InventoryDB();
19                 Iterator items = null;
20                 String category = request.getParameter("category");
21                 if(category == null)
22                 {
23                     items = invDB.getAllItems().iterator();
24                 }else
25                 {
26                     items =
27                         invDB.getItemInCategory(category).iterator();
28                 }
29                 while(items.hasNext())
30                 {
31                     InventoryBean item = (InventoryBean)
items.next();
32                     out.println("<tr>");
33                     out.println("<td>" + item.getNumber() + "</
td>");
```

```
34             out.println("<td>" +
text.getString("Description") + ": " + item.getDescription() + "</
td>");
35             out.println("<td>" + text.getString("Price") +
": " + item.getPrice() + "</td>");
36             out.println("</tr>");
37         }
38     } catch (Exception e)
39     {
40         out.println("<p>No items found</p>");
41     } %>
42
43 </table>
44 </BODY>
45 </HTML>
```

FIGURE 16. *The ScripletDemo.jsp page displays items in the inventory database.*

Some comments about the ScripletDemo.jsp page:

- It uses a ResourceBundle object that either is an attribute of the implicit session object or the default locale. There is no need to invoke the getSession() method. The implicit session object is already initialized for you in a JSP page.
- If the category parameter is defined, then the items in the database that match the category are displayed.
- If the category parameter is not defined, then the page displays all items in the database.
- No SQL code appears in the JSP. All the database access is done via the InventoryDB object.

Figure 17 shows a sample output of the ScriptletDemo.jsp page when the category parameter equals “telephones”.

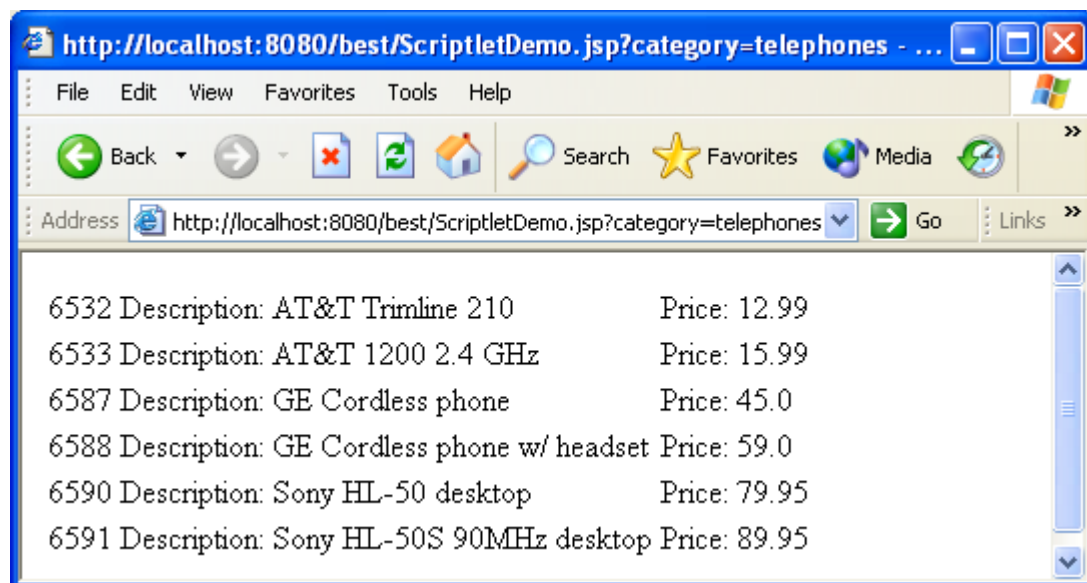


FIGURE 17. A sample output of the ScriptletDemo.jsp page.

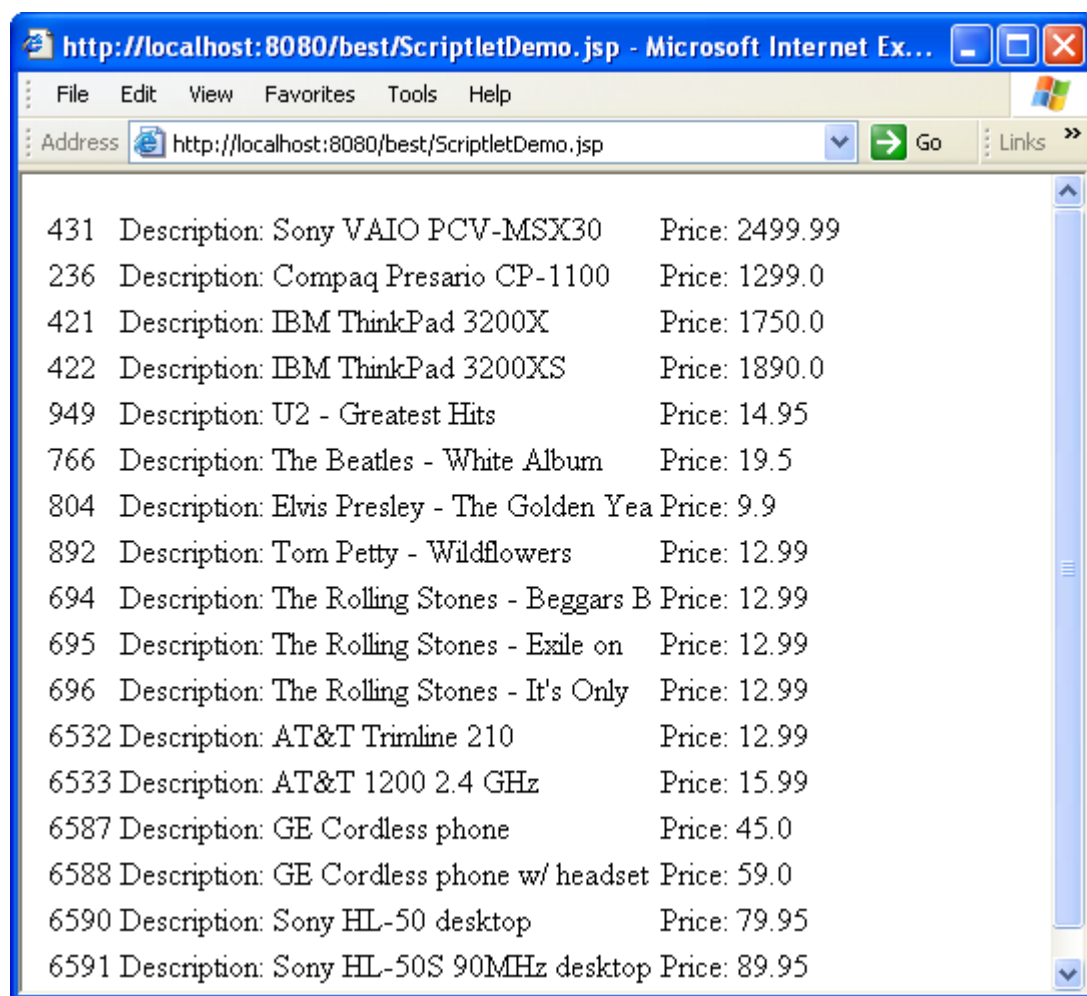


FIGURE 18. The output of *ScriptletDemo.jsp* when the category parameter is not defined.

The include Tag

A common technique in Web page development is the use of templates. For example, a Web site might want the same header on each page. Instead of repeating the same HTML in each file, the common HTML text is placed in a shared file that is included at the beginning of each page on the Web site. This has obvious maintenance and design benefits, since changing the header on every page involves only changing the HTML in one location.

JSP contains two ways to include another file or object:

- `<%@ include file="somefile.html" %>`: The denoted file is included at translation time into the output of the JSP page. This is done statically at translation time. Any changes made to "somefile.html" will not change the output of the JSP using this tag.
- `<jsp:include page="somefile.jsp" flush="true | false" />`: includes the output of the specified file in to the output of this JSP. This is done dynamically, so any changes to "somefile.jsp" will change accordingly the output of the JSP using this tag.

The first include directive above is an example of a page directive is and a common way to include output from other sources that do not change often. It is more efficient than the second directive, which actually causes the client request and response information to be sent to the other page. The second include directive above is meant for more dynamic handling of requests and responses that change with each client.

Notes

An Example Using include

The following JavaServer Page demonstrates the usage of the include directive.

```
1      <!-- Filename: IncludeDemo.jsp -->
2
3      <%@ page import="java.util.*, com.best.db.*" %>
4
5      <HTML>
6          <BODY>
7
8              <%@ include file="/DisplayHeading.html" %>
9
10             <table>
11
12                 <% (same scriptlet from ScriptletDemo.jsp) %>
13
14             </table>
15         </BODY>
16     </HTML>
```

FIGURE 19. *Using the include directive.*

The DisplayHeading.html file looks like:

```
1      <table width="75%" border="0">
2          <tr>
3              <td width="34%"></td>
4              <td width="66%">Welcome to BEST
5          </td>
6      </tr>
7  </table>
8
```

FIGURE 20. *The DisplayHeading.html file.*

The resulting Web page will look something like Figure 21.

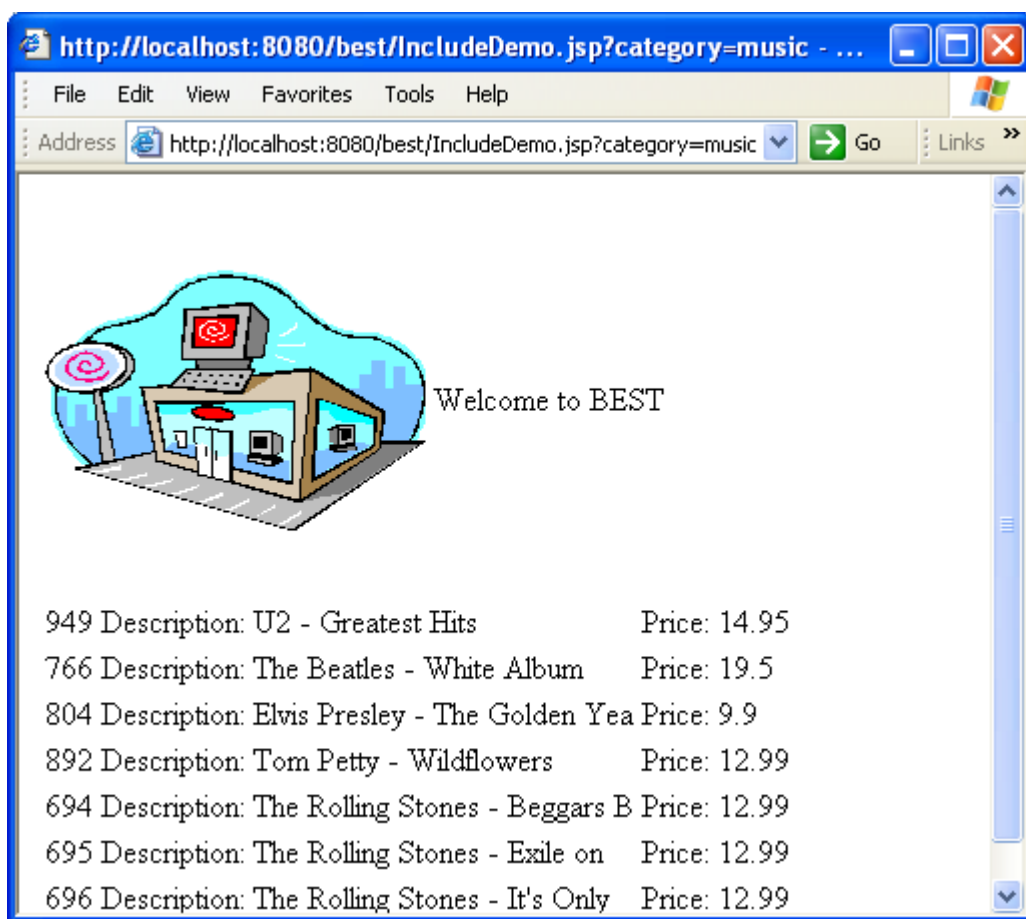


FIGURE 21. *The output of includedemo.jsp.*

Directives

A JSP directive is a tag that specifies a task to be performed for the page. A directive is created using the `<%@ %>` tags. There are three types of directives in JSP that use this syntax:

- **The include directive:** used for including the text of another JSP or file at translation time. The include directive looks like:

```
<%@ include file="filename" %>
```

where the file parameter is the name of the file to be included.

NOTE

Figure 19 on page 174 demonstrated using the include directive.

- **The taglib directive:** used for loading a set of custom JSP tags. The taglib directive looks like:

```
<%@ taglib uri="library" prefix="tagPrefix" %>
```

where library is the location of the library and tagPrefix is the prefix of the custom tag.

- **The page directive:** a collection of directives that specify options for the page. For example, importing a package, extending a class, or denoting the thread model.

The Page Directive

The page directive appears at the top of JSP page and has several parameters. The syntax for a page directive looks like:

```
<%@ page param=value %>
```

where param is one of the following options.

- **import:** imports a package.
- **extends:** denotes a parent class of the resulting servlet.
- **contentType:** sets the content type of the response. The default value is "text/html"
- **errorPage:** specifies the name of the page that will handle any errors that may occur.
- **isErrorPage:** a boolean that must be set to true for pages designed to handle errors.
- **session:** a boolean that specifies if a session object should be created, or an existing session object located, when a client accesses the page. The default value is true.
- **isThreadSafe:** a boolean that specifies whether the page is thread safe or not. The default value is true. If set to false, the corresponding servlet will implement the SingleThreadModel.
- **language:** denotes the language that appears in the scriptlets of this page. The only currently supported value is "java".

We have used the page directive already in many of the examples in this chapter to include packages. The following page directive demonstrates using some of the other parameters:

```
<%@ page import="java.util.*, java.sql.*" isThreadSafe="false"
errorPage="err.jsp" session="false" %>
```

JSP Actions

JavaServer Pages can use another type of tag referred to as an action tag. Action tags specify a specific task to be performed, like instantiating a JavaBean or forwarding the current request to another resource.

The action tags use the following syntax:

```
<jsp:action attribute="value" />
```

where action is the task to be performed. The tag can also contain one or more attributes that specify various options of the action.

JavaServer Pages contain seven standard action tags:

- `<jsp:forward page="resource" />`: forwards the request and response to the denoted resource specified by the page attribute.
- `<jsp:include page="resource" flush="true | false" />`: includes the resource in the response. The flush parameter denotes whether the buffer should be immediately flushed or not once the content is included.
- `<jsp:param name="name" value="value" />`: add the corresponding parameter and value to the current request object. This action is used within a forward, include or plugin action tag.
- `<jsp:useBean id="id" class="beanClass" scope="scope" beanName="name" type="classType" />`: instantiates a bean object of the given class type and assigns it the given id.
- `<jsp:setProperty />` and `<jsp:getProperty />`: discussed in detail later in this course, these actions are used to set and get the properties of a JavaBean.
- `<jsp:plugin />`: the plug-in action allows for download of the Java Plugin software. The HTML generated from this tag will contain either `<EMBED>` or `<OBJECT>` tags, whichever is appropriate for the client who initiated the request.

The following DirectiveDemo.jsp page demonstrates usage of the action tags.

```
1  <!-- Filename: DirectiveDemo.jsp -->
2
3  <%@ page import="java.util.*, com.best.db.*" %>
4
5  <HTML>
6      <BODY>
7
8          <% if(session.isNew()) { %>
9              <jsp:forward page="/welcome" />
10         <% } %>
11
12         <jsp:include page="/DisplayHeading.jsp" flush="false" />
13
14         <table>
15             <% (same scriptlet as ScriptletDemo.jsp) %>
16         </table>
17     </BODY>
18 </HTML>
```

FIGURE 22. *A JSP that uses an include and forward action tag.*

- The result of `session.isNew()` will be true the first time the client visits the Web site, since this JSP will have to create a new session object. In that situation, control is forwarded to the “/welcome” URL, which in our example is the `BestController` servlet.
- The JSP page `DisplayHeading.jsp` is included using the include action. The `DisplayHeading.jsp` page is defined in Figure 14 on page 166.

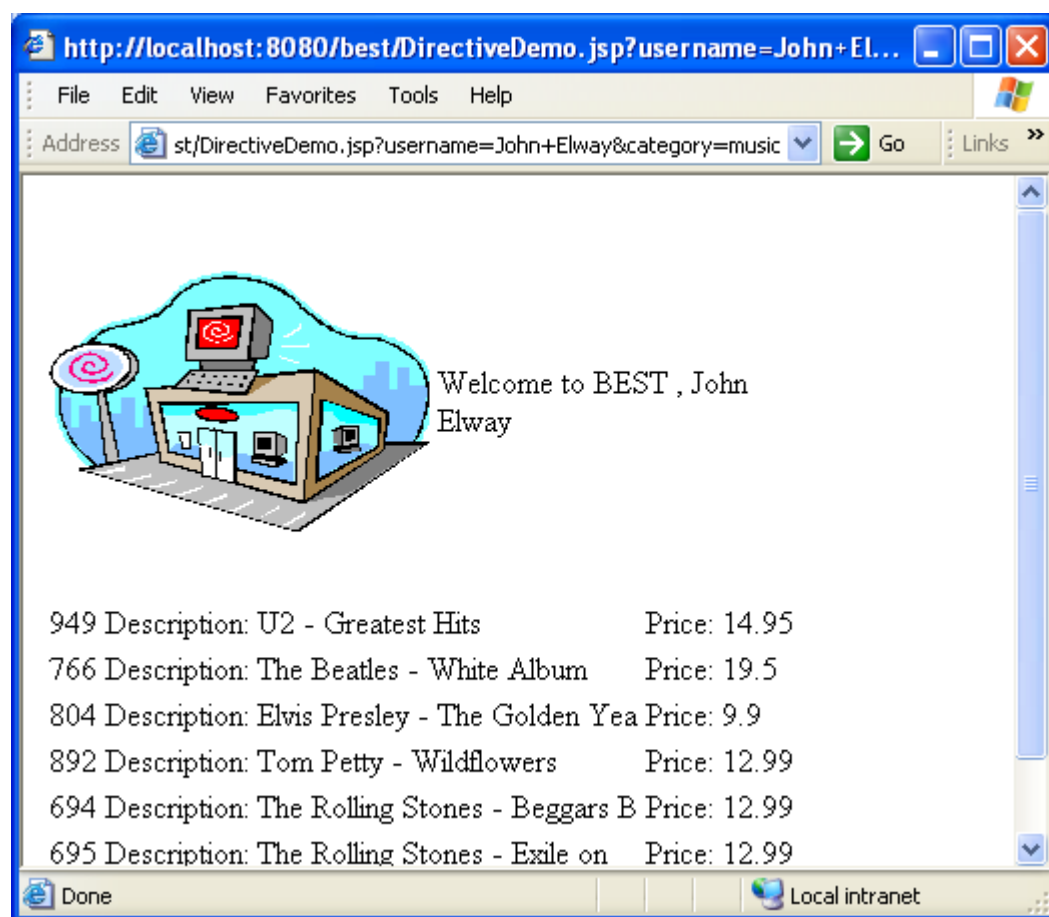


FIGURE 23. The output of the *DirectiveDemo.jsp* page when the username and category parameters are defined.

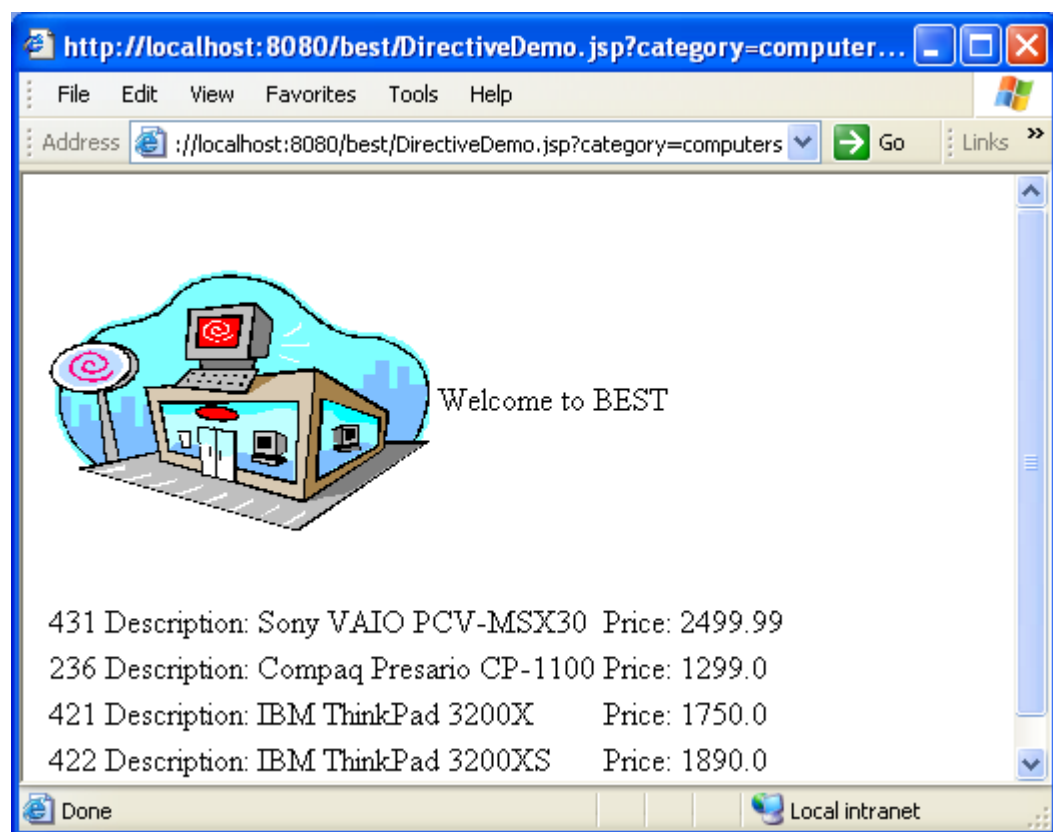


FIGURE 24. The *DirectiveDemo.jsp* when the *username* parameter is not defined.

Lab 1: Working with JSPs

Objective: To become familiar with writing JSP pages and using the various JSP elements. Your BrowseServlet currently displays the items in the database. This design does not follow the MVC model because the servlet is determining the view. In this lab, you are going to write a JSP that creates the view of all the items in the database, and your BrowseServlet is only going to control that data.

Perform the following tasks:

1. Start by modifying your BrowseServlet class. Open your BrowseServlet class and comment out any and all statements that involve generating a response. (This servlet is not going to generate any response.)
2. Your BrowseServlet is actually going to be shorter now. First, use `getParameter()` to obtain the value of a "category" parameter.
3. If category is null, then invoke `getAllItems()` using the InventoryDB field. The return value is a Collection object. Invoke the `iterator()` method to convert this Collection into an Iterator and attach the Iterator object to the request, giving it the attribute name "items".
4. If category is not null, repeat the previous step, invoking `getItemInCategory()` instead of `getAllItems()`.
5. You should now have an attribute named "items" of type Iterator attached to the request containing the items that the client wants to browse. Forward the request to `/browse.jsp`.
6. Save and compile your BrowseServlet class.
7. Create a new JSP page named `"browse.jsp"`. Save the file in the `%CATALINA_HOME%\webapps\best` directory.
8. Within `browse.jsp`, import the `java.util`, `text`, and `com.best.db` packages.

9. Declare a variable of type `ResourceBundle`. Use a scriptlet to assign this variable to the session attribute named `"text"`. If no attribute appears in the session object, use the `"text.Messages"` bundle for the locale of the request.
10. Within the `<body>` tag, include the `"/DisplayHeading.jsp"` page.
11. Create a form using the `<form>` tag whose action equals `"controller"` and method equals `"get"`. We want all requests to go through the controller servlet. In this case, when the form is submitted, the client is going to be adding items to their cart. Recall that the controller servlet uses an attribute named `"action"` to determine what to do with a request. Add the following tag within your `<form>` tag to define a hidden parameter named `"action"` whose value is `"addToCart"`:

```
<input type="hidden" name="action" value="addToCart">
```

12. Create a `<table>` within the form that displays the description and price of each item in the `"items"` request attribute, which is of type `Iterator`. Use a scriptlet and a `while()` loop to traverse the `Iterator`. Begin each row of the table with a text box so that the client can enter how many items they wish to purchase, similar to Figure 25. Use the tag:

```
<td><input type="text" name="<%= item.getNumber() %>" size="3"></td>
```

NOTE

The `"name"` attribute creates a request parameter that is going to match the item number of the product in the database.

13. Add a submit button to the form whose label is the `"AddToCart"` string of the `ResourceBundle`.
14. When you are finished with your `browse.jsp` page, save it.
15. Open your browser and visit your welcome page at `http://localhost:8080/best/welcome`. Click on the `"Browse"` link to visit the `BrowseServlet`, which should forward you to the `browse.jsp` page.

What you should see: You should see the items in the database, similar to Figure 25. If you add a `"category"` parameter, you should only see the items in that particular category. We have not written the servlet to manage the shopping cart yet, so clicking the `"Add to Cart"` button should display a page-not-found error.

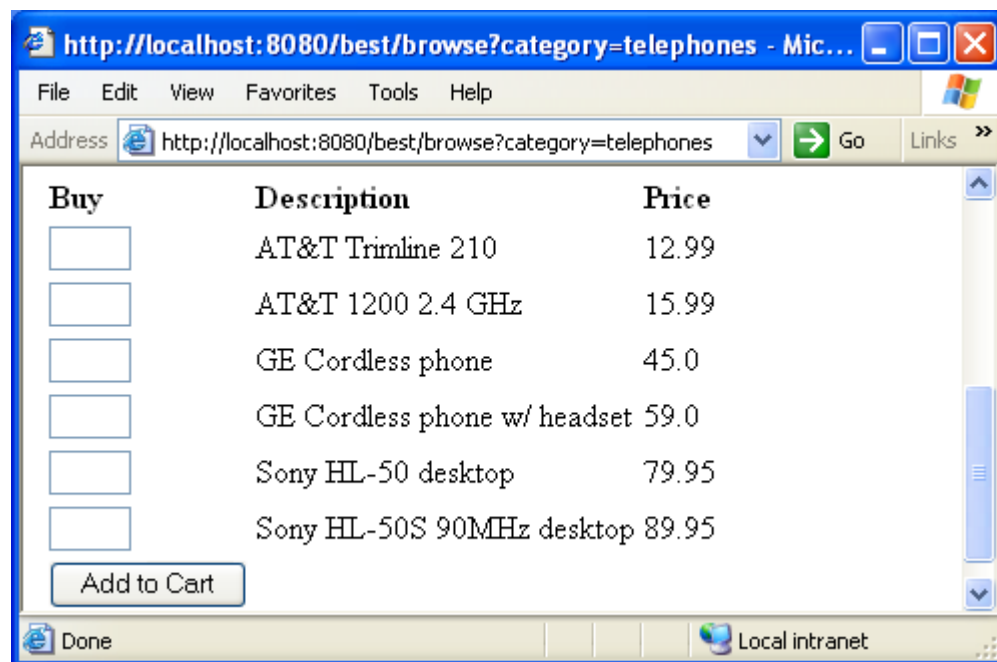


FIGURE 25. The *browse.jsp* page.



Using JavaBeans within JSP

JavaBeans are an integral part of developing effective JavaServer Pages. In this chapter, we will discuss how JavaServer Pages use the JavaBeans technology. Topics discussed in this chapter include:

- JavaBeans in JSP
- Setting and getting bean properties
- Scope of a bean
- Creating sessions in a JSP
- Error pages

The MVC Model Revisited

Keep in mind that it is best to separate the model of the data from its view. The examples discussed in Chapter 5, "An Introduction to JavaServer Pages", had a substantial amount of Java code in them. (For example, the `DeclareDemo.jsp` in Figure 5 on page 153.) These were written to demonstrate the use of JavaServer Pages. However, in a good Web application design, your JSP pages will likely contain only small amounts of Java code.

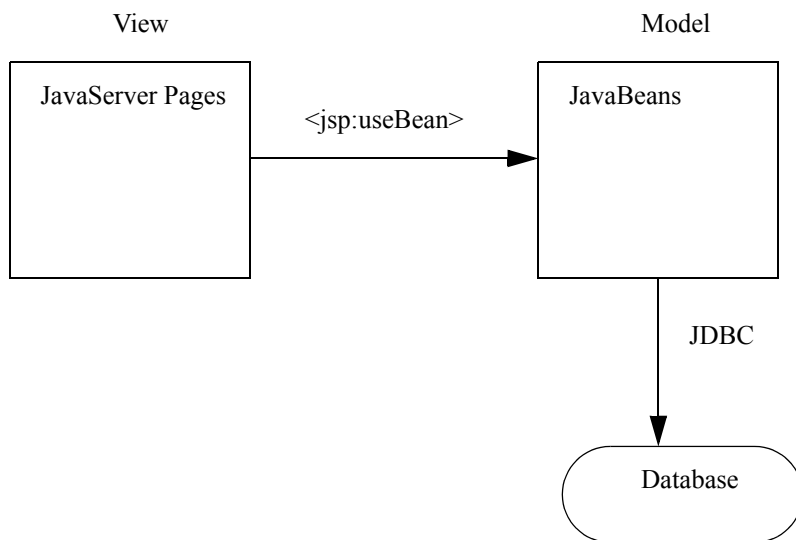


FIGURE 1. *JavaServer Pages use JavaBeans to determine their content.*

JavaBeans in JSPs

JavaServer Pages contains an action tag for using a JavaBean within a page. The `<jsp:useBean>` tag causes a JavaBean to be instantiated and makes it available to the page. The syntax for the useBean tag looks like:

```
<jsp:useBean id="name"
             class="beanClass"
             scope="scope"
             beanName="name"
             type="classType" />
```

The following list describes the attributes of the useBean tag:

- **id:** represents the name of the bean. The name must be unique to the page (naming conflicts cause an exception to occur), and is used later on the page to access the bean.
- **class:** the name of the class to be instantiated.
- **scope:** can be either "page", "request", "session" or "application". The default value is page scope.
- **type:** an optional attribute that represents the data type of the reference.
- **beanName:** represents the argument passed in to the Beans.instantiate() method.

Scope Parameter of `<jsp:useBean>`

Depending on the value of “scope” and any attributes of the particular scope object, the `<jsp:useBean>` tag either instantiates a new bean of the given class type or uses an existing bean that is an attribute of the given scope.

For example, suppose a page contains the action:

```
<jsp:useBean id="home" class="demos.Address" scope="request" />
```

The following sequence of events occurs with this tag:

1. A “home” attribute is looked for on the request object.
2. If there is no “home” attribute in the request, then a new bean of type `demos.Address` is instantiated and assigned the id “home”. This new bean is also added to the request object as an attribute named “home”.
3. If the request object contains a “home” attribute of type `demos.Address`, then a new bean is not instantiated. Instead, the existing bean is retrieved from the request object and assigned the id “home” for use on this page.

NOTE

This sequence of events occurs for “session” scope and “application” scope as well. The JSP Engine always searches the attributes of the specified scope object for an existing bean before attempting to instantiate a new one.

If no “scope” attribute is defined, then the default value of “page” is used, meaning the bean will have page scope.

Compare the following Java servlet or scriptlet code:

```
1   ResourceBundle text =
2       (ResourceBundle) session.getAttribute("text");
3   if(text == null) {
4       text = ResourceBundle.getBundle("text.Messages",
5                                       request.getLocale());
6       session.setAttribute("text", text);
7   }
```

to the following JSP action tag:

```
<jsp:useBean id="text" class="text.ResourceBean" scope="session" />
```

THE RESULT...

...is that the seven lines of Java code have the same outcome as the single `<jsp:useBean>` element.

Setting Bean Properties

JavaBeans have properties that can be accessed and changed using the `setProperty` and `getProperty` action tags. The `setProperty` tag can be used in three ways:

```
<jsp:setProperty name="beanID"
                 property="propertyName"
                 value="newValue" />
```

NOTE

The “name” attribute matches the “id” attribute of the bean. The “property” attribute is the name of the property to be set, and “value” is the value to set it to.

A common occurrence with JavaServer Pages is to have the data come from request parameters. If the new value is from a parameter, you can use the `param` tag instead of the `value` tag:

```
<jsp:setProperty name="beanID"
                 property="propertyName"
                 param="paramName" />
```

If a `JavaBean` has more than one property that maps to multiple parameters, and the parameter names match the property names, then the following tag sets each property of the bean to each matching parameter of the request:

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

For example, the following two `setProperty` tags:

```
<jsp:setProperty name="myBean" property="user" param="user" />
<jsp:setProperty name="myBean" property="password" param="password" />
```

can be replaced with:

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

Getting Bean Properties

A JSP uses the `getProperty` tag to get the value of a bean property. The `getProperty` tag obtains the value of the desired property, converts it to a `String`, and sends it back to the client using the `out` object.

The `getProperty` tag has the following syntax:

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

The `name` attribute must match the `id` attribute of the desired bean. The `property` attribute represents the name of the property to be obtained.

JavaBeans and Form Properties

Suppose we have the following `demos.Address` bean defined in Figure 2.

```
1  package demos;
2
3  public class Address implements java.io.Serializable
4  {
5      private String recipient;
6      private String streetName;
7      private int houseNum;
8      private boolean localAddress;
9      public Address()
10     {}
11     public void setRecipient(String value)
12     {
13         recipient = value;
14     }
15     public String getRecipient()
16     {
17         return recipient;
18     }
19     public void setStreet(String value)
20     {
21         streetName = value;
22     }
23     public String getStreet()
24     {
25         return streetName;
26     }
27     public void setHouseNumber(int value)
28     {
29         houseNum = value;
30     }
31     public void setLocal(boolean value)
32     {
33         localAddress = value;
34     }
35     public boolean isLocal()
36     {
37         return localAddress;
38     }
39 }
```

FIGURE 2. *The `demos.Address` bean.*

The input.jsp page in Figure 3 displays a form for the user to enter their address, The entries in the form match the properties of the Address bean.

```
1  <html>
2      <body>
3
4      <%
5      if (request.getMethod().equals("POST")) {
6      %>
7
8      <jsp:useBean id="address" class="demos.Address">
9          <jsp:setProperty name="address" property="*" />
10     </jsp:useBean>
11
12     <p>You entered:</p>
13     Recipient: <jsp:getProperty name="address"
14                 property="recipient"/><br>
15     Street name: <jsp:getProperty name="address"
16                  property="street"/><br>
17     Local?: <jsp:getProperty name="address"
18              property="local"/><br>
19     <% } else { %>
20
21     <p>Please enter your address here </p>
22     <form method="post" action="input.jsp">
23     <table>
24         <tr><td>Recipient: </td><td><input type="text"
25                                     name="recipient"></td></tr>
26         <tr><td>House number: </td><td><input type="text"
27                                     name="houseNumber"></td></tr>
28         <tr><td>Street Name: </td><td><input type="text"
29                                     name="street"></td></tr>
30         <tr><td>Local: </td><td><input type="checkbox"
31                                     name="local"></td></tr>
32         <tr><td><input type="submit" value="Submit"></td></tr>
33     </table>
34     </form>
35
36     <% } %>
37 </body>
38 </html>
```

FIGURE 3. *The input.jsp page.*

Some important comments about the input.jsp page in Figure 3:

- The form posts to itself. This JSP page both displays the form and handles the form when a client submits it.
- If the page is accessed from a GET request, the <form> is displayed, as shown in .

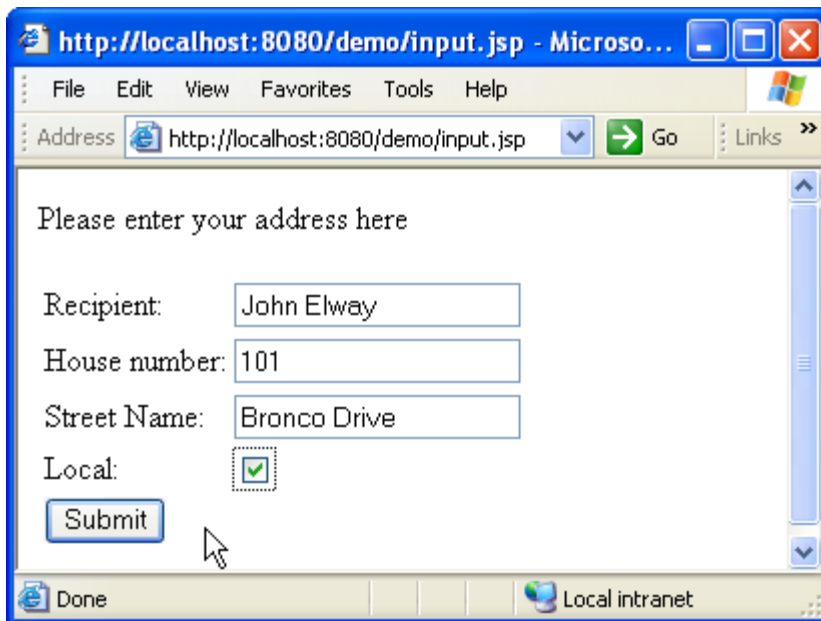


FIGURE 4. A GET request displays the <form>.

- When the form is submitted, notice the action is POST.
- When the page is accessed via a POST, a new demos.Address bean is instantiated with the properties of the form using the statements:

```
<jsp:useBean id="address" class="demos.Address">  
  <jsp:setProperty name="address" property="*" />  
</jsp:useBean>
```

- The `<jsp:getProperty>` tag is used to retrieve the readable properties from the Address bean and displayed to the user, similar to .

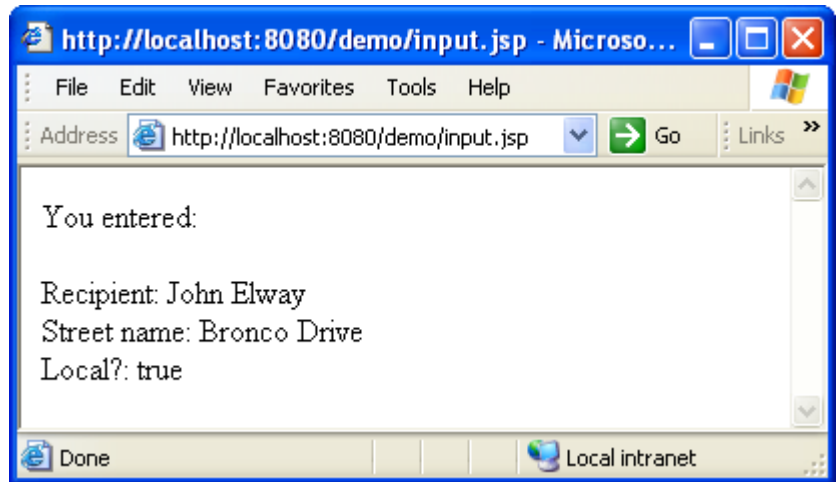


FIGURE 5. *The output of a POST made to input.jsp.*

Validating Form Data

One advantage of posting a form to itself is that it makes it easier for a form to be validated. This is typically done by the JavaBean that represents the data in the form. For example, we could add a `validate()` method to the `demos.Address` bean similar to Figure 6.

```
1  package demos;
2
3  public class Address implements java.io.Serializable
4  {
5      private String recipient;
6      private String streetName;
7      private int houseNum;
8      private boolean localAddress;
9
10     public boolean validate() throws Exception
11     {
12         if(recipient == null)
13             throw new Exception("Recipient can not be null");
14         else if(streetName == null)
15             throw new Exception("Street can not be null");
16         else if(houseNum <= 0)
17             throw new Exception("Inappropriate value for house
number");
18         else
19             return true;
20     }
21
22     //set and get methods...
23 }
```

FIGURE 6. *The Address bean can be used to validate the input of the form.*

The `input_validate.jsp` page in Figure 7 is similar to the `input.jsp` page in Figure 3 on page 195 except that the `input_validate.jsp` page invokes the `validate()` method on the `Address` bean after the data is posted.

If valid data is input, the user is forwarded to the `success.jsp` page defined in Figure 8. If the data is not correct, the user sees the same form again along with an error message.

```
1  <html>
2    <body>
3
4    <%
5      if (request.getMethod().equals("POST")) {
6        %>
7
8      <jsp:useBean id="address" class="demos.Address"
scope="request">
9        <jsp:setProperty name="address" property="*" />
10     </jsp:useBean>
11
12     <% try
13       {
14         if (address.validate())
15         {
16           %>
17           <jsp:forward page="success.jsp" />
18         <%
19           } catch (Exception e)
20           {
21             out.println("<h2>Error: " + e.getMessage() + "</h2>");
22           } %>
23       <% } %>
24
25       <p>Please enter your address here </p>
26       <form method="post" action="input_validate.jsp">
27         <table>
28           <tr><td>Recipient: </td><td><input type="text"
name="recipient"></td></tr>
29           <tr><td>House number: </td><td><input type="text"
name="houseNumber"></td></tr>
30           <tr><td>Street Name: </td><td><input type="text"
name="street"></td></tr>
31           <tr><td>Local: </td><td><input type="checkbox"
name="local"></td></tr>
32           <tr><td><input type="submit" value="Submit"></td></tr>
33         </table>
34       </form>
35     </body>
36  </html>
```

FIGURE 7. *The input_validate.jsp page forwards valid data to another page.*

```
1  <html>
2  <body>
3      <h2>Success!</h2>
4      <jsp:useBean id="address" class="demos.Address"
5                  scope="request"/>
6
7      <p>You entered:</p>
8      Recipient: <jsp:getProperty name="address"
9                  property="recipient"/><br>
10     Street name: <jsp:getProperty name="address"
11                  property="street"/><br>
12     Local?: <jsp:getProperty name="address" property="local"/
13 ><br>
14 </body>
15 </html>
```

FIGURE 8. *The success.jsp page.*

Figure 9 shows the success.jsp displayed when the user enters valid data.

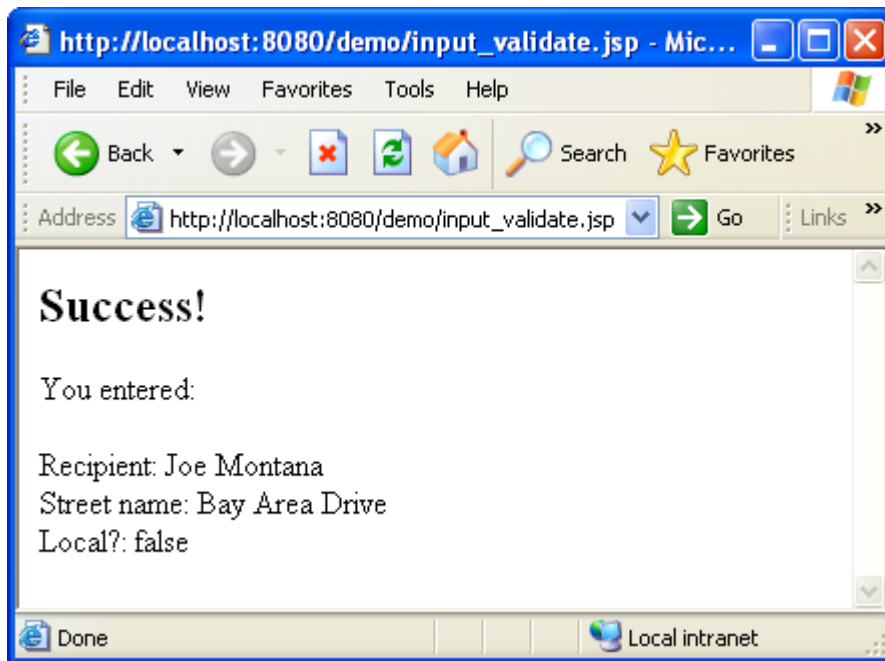


FIGURE 9. *Successfully entering data in the Web page.*

QUESTION

How did the address information in the form make its way over to the success.jsp page?

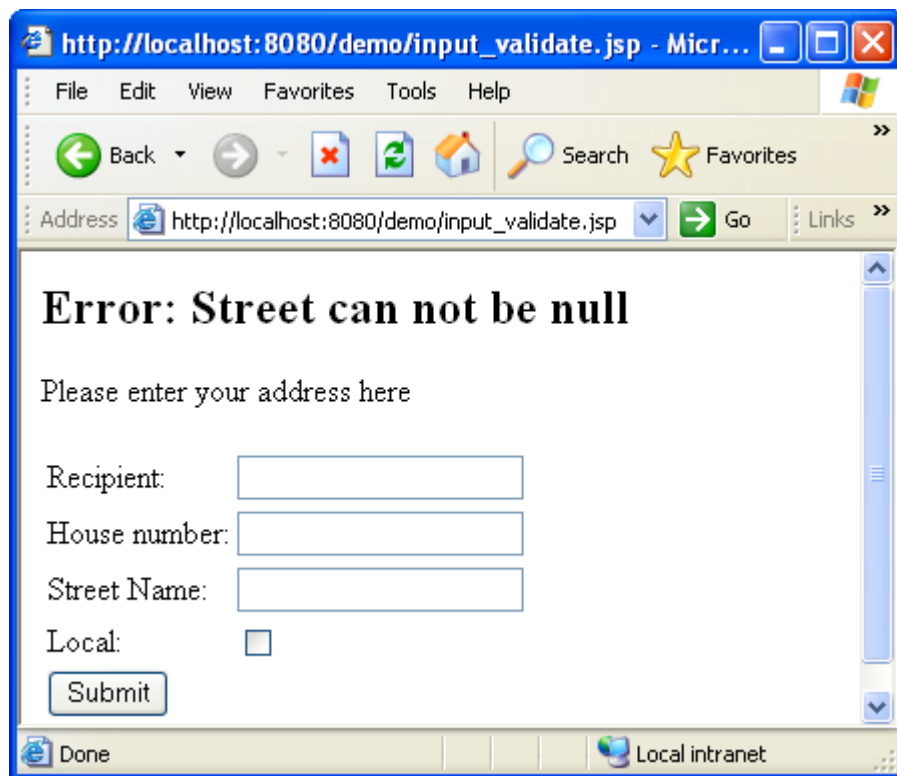


FIGURE 10. *Invalid data was entered.*

Lab 1: The ResourceBean

Objective: The purpose of this lab is to become familiar with using the `<jsp:useBean>` tag along with `<jsp:setProperty>` and `<jsp:getProperty>`.

Perform the following tasks:

1. Copy the `ResourceBean.java` file that you wrote in Lab 1 of Chapter 4, "The Model: JavaBeans and JDBC" into the directory `%CATALINA_HOME%\webapps\best\WEB-INF\classes` and compile it.
2. Write a new JavaServer Page named `"welcome.jsp"`. Save it in the `%CATALINA_HOME%\webapps\best` directory.
3. Create a bean on the page named `"resource"` of type `text.ResourceBean`. Give the bean session scope.
4. Set the `"bundle"` property of the bean to be `"text.Messages"`.
5. Set the `"locale"` property of the bean to be the locale of the request.
6. Set the `"text"` property to be `"Welcome"`, then get the `"text"` property and display it in an `<h2>` heading.
7. Similarly, set and get the `"text"` property for the `"Products"` resource bundle string. Display this string in a hyperlink using `...`.
8. Save your `welcome.jsp` page and open it in a Web browser.

What you should see: You will see the welcome page in the locale of your Web browser. For example, you should be able to change the language to Spanish and have the Spanish versions of `"Welcome"` and `"Products"` appear, as shown in Figure 11.

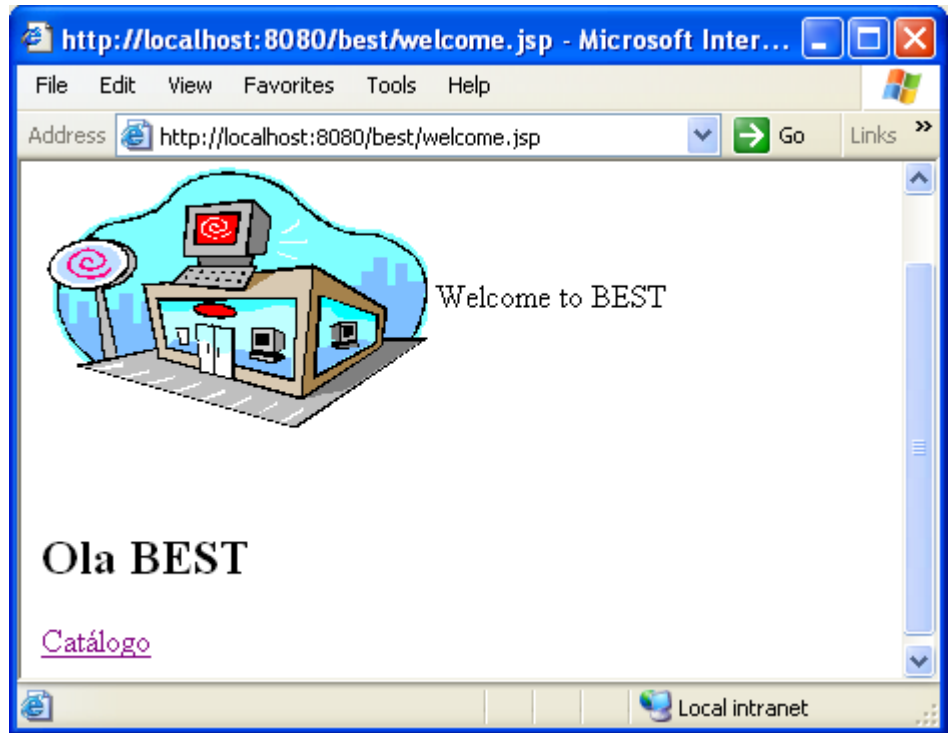


FIGURE 11. *The welcome.jsp page with locale set to "es".*

Scope

Objects within a JavaServer Page have four different types of scope:

- **Page scope:** An object that has page scope is a field in the resulting servlet class. The term "page scope" is used because the object is only available to that particular page.
- **Request scope:** An object can be associated with a client request. Objects with request scope can be accessed by another JavaServer Page that is either forwarded the request or included in this request.
- **Application scope:** An object with application scope is accessible to any JavaServer Page in the JSP engine.
- **Session scope:** A session is the term used to describe the time a client spends at a particular Web site. Objects can be associated with a client's session, making the objects available to any JSP accessed during the session.

Page scope is the default scope of declared variables and also the default scope of beans instantiated with the `<jsp:useBean>` tag. A variable with page scope is accessible anywhere within the page that it is declared.

Notes

Request Scope

An object with request scope is attached to the implicit request object of the `HttpServletRequest` object. The following methods invoked on the implicit request object can be used to create and retrieve objects with request scope:

```
public void setAttribute(String name, Object x)
```

adds the given object to the request. The name is used to retrieve the object later.

```
public Object getAttribute(String name)
```

returns a reference to the object with request scope whose name is `name`. If no object exists in the request with the given name, the method returns `null`.

```
public java.util.Enumeration getAttributeNames()
```

returns a collection of strings containing the names of all attributes associated with the request.

```
public void removeAttribute(String name)
```

removes the object with the given name from the request.

NOTE

Attributes are reset between each request. If an object needs to exist beyond the life of the request, then application or session scope needs to be used.

You can also create request scope with JavaBeans by using the scope attribute of the `<jsp:useBean>` tag:

```
<jsp:useBean id="beanID" class="className" scope="request" />
```

The example of validating forms discussed earlier used session scope to send the Address bean from the `input_validate.jsp` page to the `success.jsp` page.

Within `input_validate.jsp`, the tags used are:

```
<jsp:useBean id="address" class="demos.Address" scope="request">  
  <jsp:setProperty name="address" property="*" />  
</jsp:useBean>
```

This created a new Address bean and added it as an attribute named “address” to the request object. When the request was forwarded to `success.jsp`, the following tag was used to retrieve this Address bean from the request object:

```
<jsp:useBean id="address" class="demos.Address" scope="request"/>
```

Application Scope

An object can be given application scope by assigning it as an attribute of the JSP engine. Methods for communicating with the JSP engine are found in the implicit application object.

The following methods of the `ServletContext` interface can be invoked on the application object.

```
public void setAttribute(String name, Object x)
```

adds the given object to the application object. The name is used to retrieve the object later.

```
public Object getAttribute(String name)
```

returns a reference to the object with application scope whose name is name. If no object exists in the application object with the given name, the method returns null.

```
public java.util.Enumeration getAttributeNames()
```

returns a collection of strings containing the names of all attributes associated with the application object.

```
public void removeAttribute(String name)
```

removes the object with the given name from the application object.

As with request scope, application scope is best obtained using JavaBeans instead of the methods listed above. The process for creating application scope with JavaBeans is to denote the scope attribute of the `<jsp:useBean>` tag as "application".

For example, the following JSP page attaches an InventoryDB object to the application scope.

```
<jsp:useBean id="invDB" class="com.best.db.InventoryDB"
scope="application"/>
```

To obtain a reference to the InventoryDB JavaBean with application scope, you use the `<jsp:useBean>` tag. If the id and class type match a bean found in the application object, then that will be the bean used.

Keep in mind that the application scope is available to all Web components in the container. Assuming the above JSP added the "invDB" attribute to the application scope, the following servlet could retrieve that attribute:

```
1  public class BrowseServlet extends HttpServlet
2  {
3      private InventoryDB invDB;
4
5      public void init() throws ServletException
6      {
7          invDB = (InventoryDB)
8              getServletContext().getAttribute("invDB");
9      }
10 }
```

RECALL

Application scope in a servlet is obtained by using the servlet context object. The implicit "application" object available to all JavaServer Pages is the same servlet context object.

Session Scope

A session is the term used to describe the time a client spends at a Web site, assuming that subsequent requests are related to previous requests. Sessions are commonplace in today's Web sites:

Having users log on to access private data without having to verify their username and password on each page visited.

- Shopping carts.
- Online banking.
- Tracking the pages that a client visits to determine their personal likes and dislikes.

These are only a few of the practical uses of client session tracking. There are two common techniques to manage sessions within JavaServer Pages:

- **JavaBeans:** a JavaBean can be assigned session scope.
- **The session Object:** an interface in the Servlet API that simplifies the creation and tracking of sessions. A JavaServer Page uses the implicit session object for managing a client session using the HttpSession interface.

Notes

The Implicit Session Object

The `javax.servlet.http.HttpSession` interface is used for encapsulating the task of session tracking within servlets. An `HttpSession` object is created by the JSP engine, and a JSP uses its implicit session object to invoke the methods of the interface.

JavaServer Pages can specify the session attribute of the page directive to be true or false. If the session attribute is false, then the implicit session object will be unavailable for this page. For example,

```
<%@ page session="false" %>
```

Note the session attribute defaults to true, which means the session object will be available for the page.

The session object contains methods for storing objects within the session object, thereby giving the stored objects session scope:

```
public void setAttribute(String name, Object x)
```

adds the given object to the session object. The name is used to retrieve the object later.

```
public Object getAttribute(String name)
```

returns a reference to the object within the session scope whose name is name. If no object exists with the given name in the session object, the method returns null.

The following JSP demonstrates the usage of the implicit session object. The first time a client access this page, a session object will not be created since the session attribute is false. If a session object exists for this client, then the reference will not be null and a new ShoppingCart object is added to the session.

```
1 <!-- Filename: SessionDemo.jsp -->
2 <%@ page session="true" %>
3
4 <% if(!session.isNew()) { %>
5     <jsp:forward page="KeepShopping.jsp" />
6 <% } %>
7
8 <% ShoppingCart cart = new ShoppingCart(); %>
9 <% session.setAttribute("cart", cart); %>
10 <jsp:forward page="BeginShopping.jsp" />
```

FIGURE 12. *The implicit session object is used to maintain a shopping cart.*

Beans with Session Scope

A JavaBean can be given session scope by assigning the scope attribute of the useBean tag to equal "session". For example,

```
<jsp:useBean id="bank" class="BankSession" scope="session" />
```

There are two possible scenarios that can occur when using the useBean tag with "session" scope:

- If the bean is not a part of the particular client's session object, then a new BankSession will be instantiated and attached to the session object.
- If this client's session object contains a BankSession bean with an id equal to "bank", then the existing BankSession bean will be used on this page.

Notes

Lab 2: Create the ShoppingCart View

Objective: To become familiar with creating a client session using JavaServer Pages. You will create a shopping cart for clients who access the BEST site.

Perform the following tasks:

1. Three classes have been written for you for: `ManageCartServlet.java`, `ShoppingCart.java` and `ShoppingCartItem.java`. They can be found in the directory `.\WebDevelopment\Solutions\Chapter_07\Lab2\best\WEB-INF\classes\`. Copy these three .java files into the directory `%CATALINA_HOME%\webapps\best\WEB-INF\classes` and compile them (using the `-d .` flag). (If you already have a `ShoppingCart.java` file in the `\classes` directory, you can simply delete it.)
2. Modify your `web.xml` file for the "best" application, adding the `com.best.ManageCartServlet`. Map its URL to `/managecart`.

NOTE

`ManageCartServlet` is invoked whenever the user wants to add, remove or view the items in the cart, which is when the request parameter "action" is "addToCart", "removeFromCart" or "showCart". It uses the methods of the `ShoppingCart` class to appropriately add or remove the specified items. The item numbers and quantities are parameters in the request object.

3. Write a new JSP page named `managecart.jsp`. Save this file in the `%CATALINA_HOME%\webapps\best` directory.
4. Use the `<jsp:useBean>` tag to use a bean whose id is "resource" with "session" scope of type "text.ResourceBean".
5. Use the `<jsp:useBean>` tag to use on this page a bean of type "com.best.ShoppingCart" whose id is "cart" and scope is "session".
6. Add the `<html>` and `<body>` elements, then use the `<jsp:include>` tag to include the `/DisplayHeading.jsp` page.
7. Use the `getItems()` method of the `ShoppingCart` bean to retrieve a `java.util.Iterator` object of `ShoppingCartItem` objects.

8. Write a while loop to display the description and quantity of each ShoppingCartItem in the Iterator using a <table>. Within each row of the table, add a link that allows the user to remove the selected item from their shopping cart, similar to Figure 13. This "remove" link should have two parameters: "action" equal to "removeFromCart", and the item number equal to the quantity, which can be obtained from the "number" and "quantity" fields of the ShoppingCartItem object.
9. Open your web browser and enter the /welcome page. Click on the link to browse the database, and add some items to your cart. Then remove some of the items from your cart, ensuring that your managecart.jsp page is working properly.

What you should see: You should see the items in your shopping cart and be able to add and remove items from the cart.

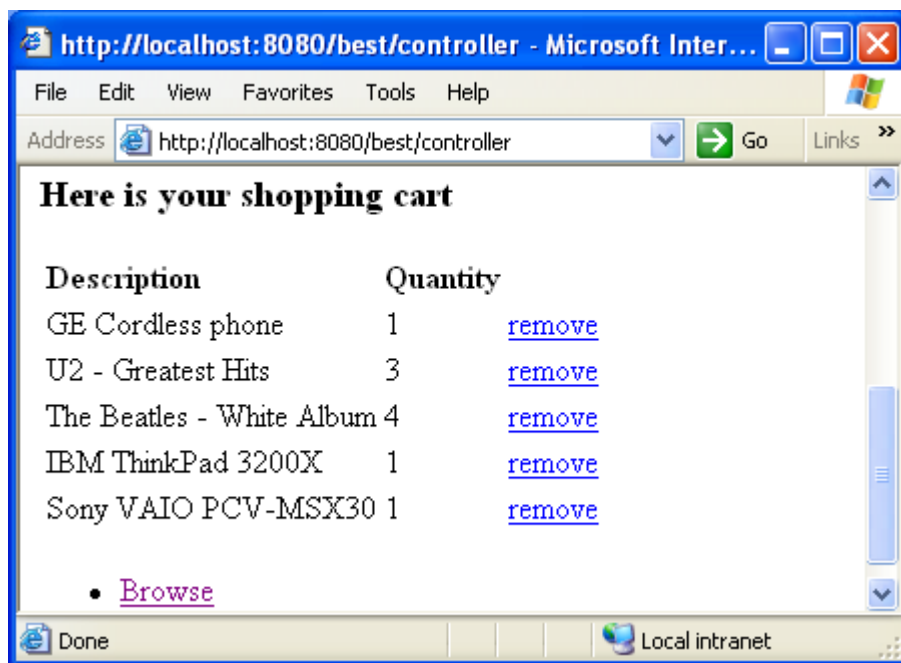


FIGURE 13. *The managecart.jsp page.*

Error Pages

JavaServer Pages are designed to be written by Web developers, and the intent is to minimize the amount of Java code. With this in mind, notice that exception handling is often an afterthought of a JSP. You are not required to use try/catch blocks since the container handles all exceptions.

However, the JavaServer Pages specification allows for a JSP to denote another JSP that handles any exceptions that may occur. The error page can be used to catch an exception before it is passed on to the client, giving you an opportunity to display the exception in a user-friendly format or suppress the exception.

The following lists the steps involved in creating an error page:

- Write a JavaServer Page to handle exceptions. Use the page directive to set the `isErrorPage` attribute to `true`.
- Use the implicit exception object to determine information about the exception.
- In your other JavaServer Pages, use the page directive to assign the `errorPage` attribute to your JSP error page.

The following JavaServer Page is designed to handle exceptions occurring from other pages. Notice the `isErrorPage` attribute is set equal to `true`.

```
1  <!-- Filename: err.jsp -->
2  <%@ page isErrorPage="true" %>
3
4  <html>
5      <body>
6          <h2>An error occurred!</h2>
7          The exception is <%= exception.toString() %>
8      </body>
9  </html>
```

FIGURE 14. *The err.jsp page.*

The following version of ScriptletDemo in Figure 15 has a typo in it. Instead of accessing the ResourceBundle “text.Messages”, it attempts to access “text.Message”, which is a class that does not exist.

```
1  <%@ page import="java.util.*" errorPage="err.jsp" %>
2
3  <%
4  ResourceBundle text = (ResourceBundle)
5                      session.getAttribute("text");
6  if(text == null)
7  {
8      text = ResourceBundle.getBundle("text.Message",
9                                     request.getLocale());
10 } %>
```

FIGURE 15. A JSP that uses the *err.jsp* page as its error page.

Figure 16 shows the resulting Web page when trying to execute the code in Figure 15.

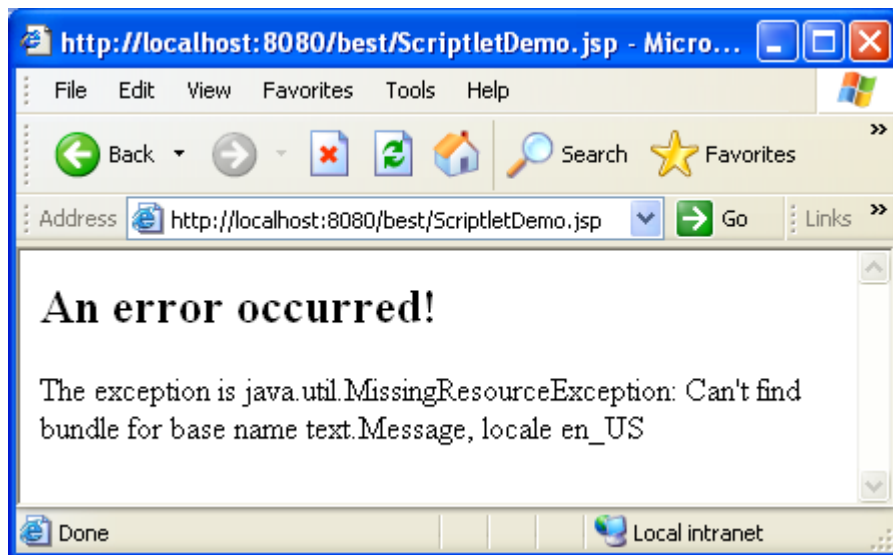


FIGURE 16. The error page is displayed when an exception occurs.

Lab 3: An Error Page for BEST

Objective: To become familiar with creating an error page to handle any exceptions generated by the JavaServer Pages on the BEST site.

Perform the following tasks:

1. Create a JavaServer Page called "error.jsp" and save it in the %CATALINA_HOME%\webapps\best directory with your other JSPs.
2. Use the page directive to denote that "error.jsp" is an error page.
3. Add the <HTML> and <BODY> tags.
4. Display a message in the body of the page stating that the user has reached the error page and that the page they were trying to access is not available.
5. Invoke toString() and getMessage() on the exception object to display information about the exception that has occurred, similar to Figure 17.
6. Add a link back to the "welcome" page so the user can easily leave the error page and start over again shopping.
7. Add the ending </BODY> and </HTML> tags and save the file.
8. Using the page directive, denote "error.jsp" as the error page for all JSP pages you have created so far in this project.
9. See if you can purposely cause an exception to occur on one of your pages and verify that your error page is working properly.

What you should see: The error page should be displayed anytime an unhandled exception occurs on any of JSP pages.

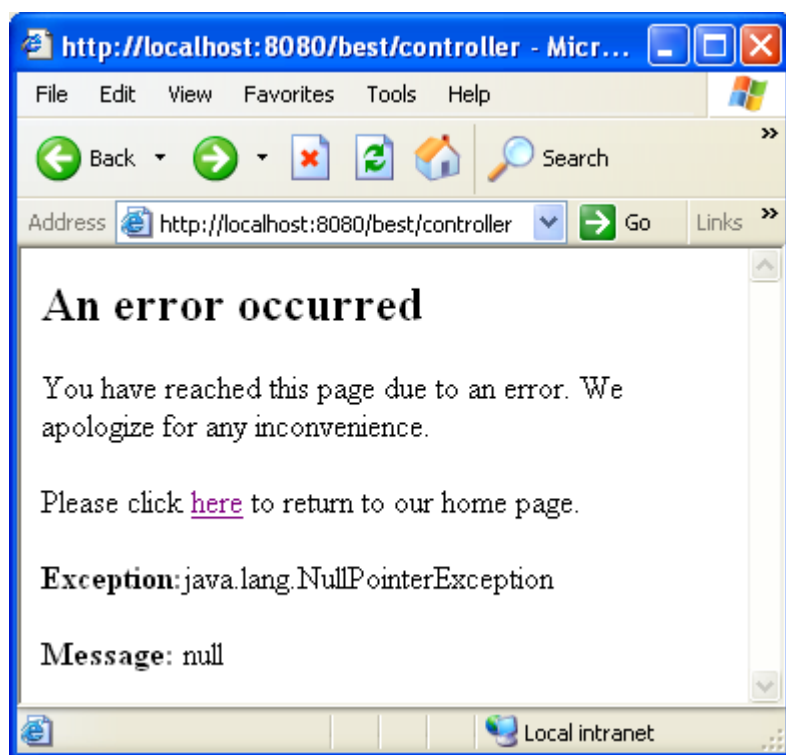


FIGURE 17. A sample output of the *error.jsp* page.



Custom JSP Tags

In this chapter, we will discuss the details of creating and using custom tags. Topics discussed in this chapter include:

- Steps to Create Your Own Tags
- Step 1: Write the Java code
- Step 2: Create a TLD File
- Step 3: Use the tags in a JSP
- Step 4: Deploy the files
- JSP Standard Tag Library (JSTL)
- The Core JSTL Tags
- The JSP Expression Language

Overview of Custom Tags

One of the more important features of JSP development is working with custom tags, either ones that you write yourself or tags that someone else provides you.

Custom tags have the following features:

- They map to Java code that executes when the tag is reached on a JSP page.
- Data from the JSP page can be passed to the custom tag using attributes.
- The custom tag can modify the response sent back to the client.
- They allow you to easily reuse code.
- Non-programmers can write JSP pages using custom tags, avoiding any Java code in your JavaServer Pages. This is nice feature for HTML developers who are not Java programmers.
- You can modify the effect of a tag without modifying the JSP page.

NOTE

Tags are described in an XML file known as a Tag Library Descriptor (TLD). The TLD maps the custom tags to the Java class that represents the code of the tags. You use the `<%@ taglib %>` directive to denote the location of the TLD file. For example:

```
<%@ taglib uri="/WEB-INF/tlds/best.tld" prefix="best" %>
```

The syntax for custom tags is similar to XML. The tags must be well-formed, meaning each element must have an ending element and they must be nested properly. They also consist of a prefix that denotes the library where the tag is defined.

There are several types of custom tags, including:

- A simple tag - which has no body or attributes. For example:

```
<best:checkout />
```

- A body tag - which has a body and possibly attributes. For example:

```
<best:showItems option="all">
    <a href="/checkout">Click here to check out</a>
</best:showItems>
```

- A tag with an attribute - which looks like:

```
<best:addToCart number="101" />
```

Steps to Create Your Own Tags

Let's take a look at the steps involved in writing your own custom JSP tags:

1. First, you write a Tag Handler class that implements one of the JspTag interfaces in the `javax.servlet.jsp.tagext` package. The actual interface that you implement depends on the type of tag you have (whether or not it has a body and/or attributes.)
2. Write a Tag Library Descriptor (TLD).
3. Use the tags in a JSP page.
4. Deploy the classes, TLD and JSP pages.

Let's go through each of these steps in detail. We will create a custom tag that displays the items in a particular category of the inventory items in the "best" database. The custom tag will use an attribute to denote which category the client wants to browse:

```
<best:showItems category="computers" />
```


The javax.servlet.jsp.tagext Package

A tag handler is a Java class that contains methods that get invoked when the corresponding custom tags appear within a JSP page. A tag handler has the following properties:

- It must implement one of the JspTag interfaces, which are shown in Figure 1.
- Optionally, a tag handler class can extend either TagSupport, SimpleTagSupport or BodyTagSupport, helper class that implement the methods of their respective interfaces for you.

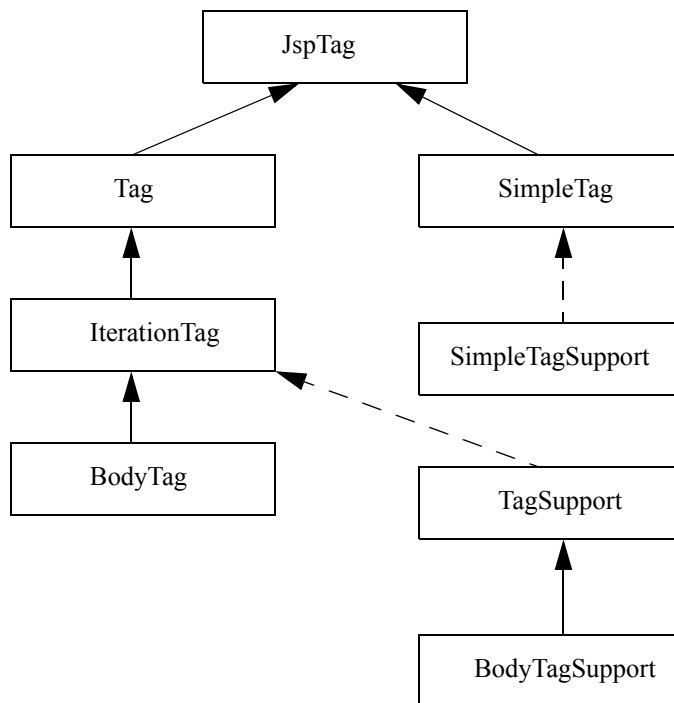


FIGURE 1. *The hierarchy of the JspTag interfaces.*

- The tag handler has access to all of the implicit objects that a JSP page has access to, like request, response, out, page, application and so on.

The Lifecycle of a Tag Handler

The different interfaces have Figure 1 create handlers that each have a unique lifecycle. For example, the lifecycle of an `IterationTag` handler is rather complex compared the lifecycle of a `SimpleTag` handler.

In our example, our custom tag is not going to have a body, so we do not need to the behavior of a `BodyTag` handler. We will write a class that extends `TagSupport` instead.

The `TagSupport` handler we write will have the behavior of a `Tag` handler, whose lifecycle is a sequence of method calls made by the container. Here is the order in which methods are invoked on a `Tag` Handler when its tag is reached in a JSP page:

- `setPageContext()` - use the page context to access the implicit JSP objects.
- `setParent()` - which passes in the parent tag handler, if any.
- For each attribute, its corresponding “set” method is invoked. For example, if a custom tag has an attribute named “category”, then the “`setCategory()`” method of the tag handler is invoked.
- `doStartTag()` - this is where your logic appears for the tag. This method returns either `EVAL_BODY_INCLUDE`, which tells the container that the body of this tag is to be processed, or `SKIP_BODY` to not process the body of the custom tag.
- `doEndTag()` - to denote that the end of the tag has been reached. This method returns either `EVAL_PAGE`, which means the remainder of the JSP page continues to be evaluated, or `SKIP_PAGE`, meaning the request completed and the remainder of the current page is skipped.
- `release()` - the process is complete and any resources can be freed.

Notes

Step 1: Write the Tag Handler Class

Our custom tag is going to look like:

```
<best:showItems category="computers" />
```

Figure 2 defines a class named `ShowItemsTag` that will be used to define the `<best:showItems>` tag. Notice the “set” and “get” methods for the “category” attribute.

```
1  package com.best.tags;
2
3  import com.best.db.*;
4  import javax.servlet.jsp.tagext.*;
5  import javax.servlet.jsp.*;
6  import java.util.*;
7
8  public class ShowItemsTag extends TagSupport
9  {
10     private PageContext pageContext;
11     private String category;
12
13     public void setPageContext(PageContext p)
14     {
15         pageContext = p;
16     }
17
18     public void setCategory(String c)
19     {
20         category = c;
21     }
22
23     public String getCategory()
24     {
25         return category;
26     }
27
28     public int doStartTag() throws JspException
29     {
30         System.out.println("Inside doStartTag()");
31         try
32         {
33             JspWriter out = pageContext.getOut();
34
35             ResourceBundle text = (ResourceBundle)
36                 pageContext.getSession().getAttribute("text");
37             if(text == null)
38             {
```

```
39             text = ResourceBundle.getBundle("text.Message",
40                 pageContext.getRequest().getLocale());
41         }
42
43         InventoryDB invDB = new InventoryDB();
44         Iterator items = null;
45         if(category == null)
46         {
47             items = invDB.getAllItems().iterator();
48         }else
49         {
50             items =
51 invDB.getItemInCategory(category).iterator();
52         }
53
54         out.println("<table>");
55         while(items.hasNext())
56         {
57             InventoryBean item = (InventoryBean)
58                 items.next();
59             out.println("<tr>");
60             out.println("<td>" + item.getNumber() + "</
td>");
61             out.println("<td>" +
62                 text.getString("Description") + ": " +
63                 item.getDescription() + "</td>");
64             out.println("<td>" + text.getString("Price") +
65                 ": " + item.getPrice() + "</td>");
66             out.println("</tr>");
67         }
68         out.println("</table>");
69     }catch(Exception e)
70     {
71         throw new JspException(e.getMessage());
72     }
73     return SKIP_BODY;
74 }
75
76 public int doEndTag() throws JspException
77 {
78     System.out.println("Inside doEndTag()");
79     return EVAL_PAGE;
80 }
81 }
```

FIGURE 2. *The ShowItemsTag class.*

The Tag Library Descriptor

Once you have the tag handler class written, the next step is to write a tag library descriptor (TLD) file, an XML document that maps custom tags in a JSP page to the code behind the scenes.

A TLD must be a valid XML document, and therefore must contain an appropriate DTD. The beginning of a TLD looks like:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag
Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
```

This assumes you are using the 1.2 version of the tag library DTD.

The root element is `<taglib>`, followed by:

- `tlib-version` - The tag library's version
- `jsp-version` - The JSP specification version that the tag library requires
- `short-name` - Optional name used by a JSP page authoring tools
- `uri` - A URI that uniquely identifies the tag library
- `display-name` - Optional name used by authoring tools
- `small-icon` - Optional small-icon used by authoring tools
- `large-icon` - Optional large-icon that can be used by authoring tools
- `description` - Optional description of the tag library
- `listener` - a tag library can have a listener, which is defined in the `<listener-class>` subelement.
- `tag` - contains information about a custom tag.

The <tag> element has a list of subelements:

- name - the tag name
- tag-class - the fully-qualified name of the tag handler class
- tei-class - a TagExtraInfo class that contains information about the tag.
- body-content - the body content type, which is either “empty”, “JSP”, or “tagdependent”. Use “JSP” for HTML, scripting elements and other custom tags. Use “tagdependent” for any other types of content.
- description - optional description of the tag
- variable - Optional scripting variable information
- attribute - information about attributes.

The attribute element looks like:

```
<attribute>
  <name>...</name>
  <required>true|false|yes|no</required>
  <rtexprvalue>true|false|yes|no</rtexprvalue>
  <type>...</type>
</attribute>
```

NOTE

Set the <rtexprvalue> attribute to true or yes if the attribute is to be dynamically set by the container, which is the typical desired behavior. Note, however, that this value defaults to “false”.

Step 2: Create a TLD File

Let's look at an example.

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE taglib
3      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.2//EN"
4      "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
5  <taglib>
6      <tlib-version>1.0</tlib-version>
7      <jsp-version>1.2</jsp-version>
8      <short-name></short-name>
9      <tag>
10         <name>showItems</name>
11         <tag-class>com.best.tags.ShowItemsTag</tag-class>
12         <body-content>empty</body-content>
13         <attribute>
14             <name>category</name>
15             <required>false</required>
16             <rtexprvalue>true</rtexprvalue>
17         </attribute>
18     </tag>
19 </taglib>
```

FIGURE 3. *The tag library descriptor for the `<best:showItems>` tag.*

Some comments about the TLD in Figure 3:

- This library defines one custom tag: `showItems`.
- The `showItems` tag has one attribute: `category`.
- The `category` attribute is not required.
- The class that handles the `showItems` tag is `com.best.tags.ShowItemsTag`, which is the class defined in Figure 2.
- TLD files are typically saved with a `.tld` extension. This example is saved as “`best.tld`”.

Notes

Step 3: Use the tags in a JSP

We are now ready to use the `<showItems>` tag in a JSP page. To use a tag, you need to declare the TLD library location using the syntax:

```
<%@ taglib uri="tld_file" prefix="prefix" %>
```

The uri is where the TLD file can be found, and each tag library uses a prefix to denote which TLD the tag is defined in.

In our example, we will use the directive:

```
<%@ taglib uri="/WEB-INF/tlds/best.tld" prefix="best" %>
```

```
1  <%@ taglib uri="/WEB-INF/tlds/best.tld" prefix="best" %>
2
3  <html>
4  <body>
5
6      <%@ include file="/DisplayHeading.html" %>
7
8      <p>Here is our first custom tag!</p>
9      <best:showItems category="telephones" />
10 </html>
11 </body>
```

FIGURE 4. *The TagDemo.jsp page uses a custom tag.*

That's it! Once you have denoted the tag library with the taglib directive, you can use the tags on the page.

NOTE

The TagDemo.jsp page has no Java code in it. It contains only HTML and JSP tags, but no scriptlets.

Notes

Step 4: Deploy the files

The next step is to deploy the files and make sure everything is working properly.

- Place the bytecode file `\com\best\tags\ShowItemsTag.class` in the `%CATALINA_HOME%\webapps\best\WEB-INF\classes` directory.
- Place the file “best.tld” from Figure 3 in the `%CATALINA_HOME%\webapps\best\WEB-INF\tlds` directory, which you will need to create.
- Place the JSP page `TagDemo.jsp` in the `%CATALINA_HOME%\webapps\best` directory.

You are now ready to view the `TagDemo.jsp` page and see if the custom tag worked. Figure 5 shows what the output should look like.

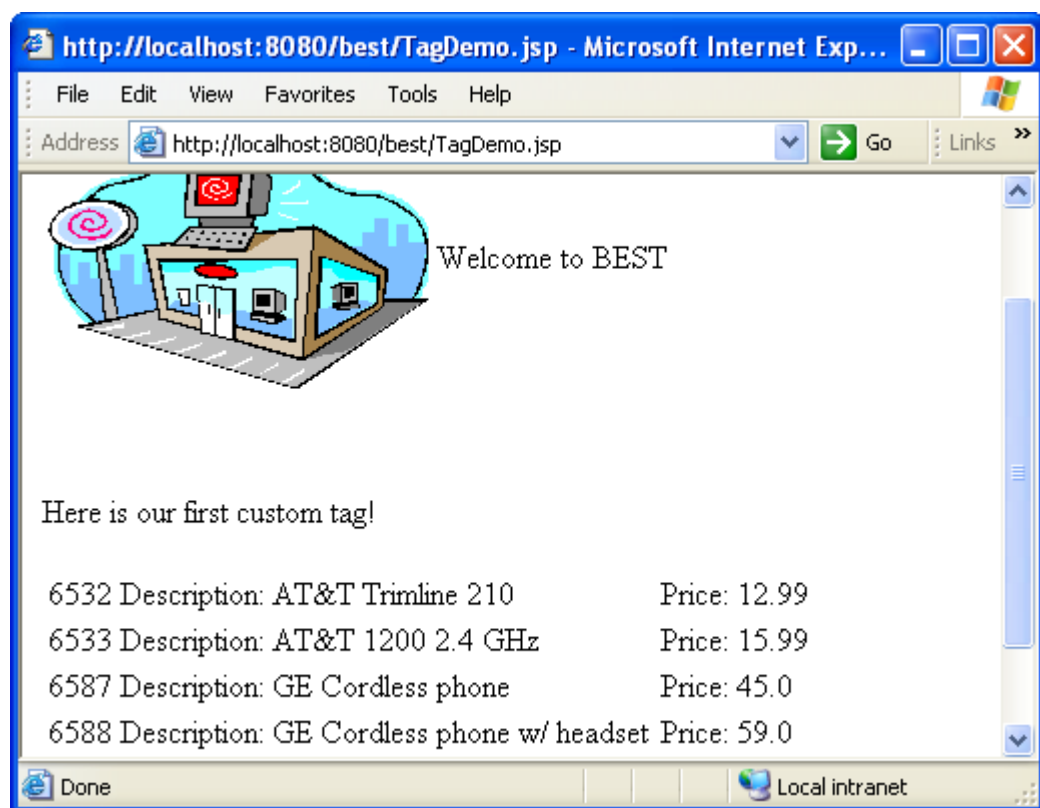


FIGURE 5. The output of the TagDemo.jsp page.

Body Tags

A custom JSP tag can contain a body that formats the content generated by the tag handler. Body tags also allow you to iterate through the tag like a loop. This is useful for displaying queries, result sets, HTML tables, HTML lists, and all types of other repetitive tasks.

Here is an example of a custom tag with a body:

```
<ul>
<best:iterateItem category="computers">
  <li><%= description %></li>
</best:iterateItem>
</ul>
```

NOTE

Notice in the ShowItemsTag class that the items were displayed, but the tag handler class contained all of the HTML formatting. Using a BodyTag allows the tag handler to produce the content and the JSP developer to format the content.

The implementing class of a body tag needs to implement the BodyTag or IterationTag interface. Typically, you will write a class that extends BodyTagSupport.

The Lifecycle of a Body Tag

The following methods are invoked, in the order listed, on a custom tag class that extends the `BodyTagSupport` class:

- `setPageContext()`
- `setParent()`
- The “set” methods of any attributes are invoked
- `doStartTag()` - this method is invoked before any of the body elements have been evaluated yet. This method returns either `EVAL_BODY_INCLUDE` (to output to body in the response) or `EVAL_BODY_BUFFERED` (for `BodyTag` handlers) or `SKIP_BODY` (to skip the body).
- `setBodyContent(BodyContent)` - this method is invoked once if `doStartTag()` returns `EVAL_BODY_BUFFERED`. A tag handler should save the `BodyContent` object in a field for use within subsequent method calls. `BodyContent` is a child of `JspWriter` and provides supports for adding content to the response.
- `doInitBody()` - an initialization method that allows you to perform any tasks that need to occur before processing begins. This method is not invoked if the tag does not contain a body or `doStartTag()` returns `SKIP_BODY` or `EVAL_BODY_INCLUDE`.
- `doAfterBody()` - here is where you decide whether to repeat the iteration or not. If this method returns `EVAL_BODY_AGAIN`, the body of the tag is evaluated again and `doAfterBody()` is invoked again. If this method returns `SKIP_BODY`, then no more body evaluations occur.
- `doEndTag()` - invoked at the end of the body evaluations, this method returns `EVAL_PAGE` to have the rest of the page evaluated and `SKIP_PAGE` if the response for the client is complete. Typically this is the method to output the `BodyContent` the response.
- `release()`

An Example of a Body Tag

Suppose we have the following page named `BodyTagDemo.jsp` that uses the custom tag `<best:iterateItem>`. Notice this custom tag has a body to it:

```
1  <%@ taglib uri="/WEB-INF/tlds/best.tld" prefix="best" %>
2
3  <html>
4  <body>
5
6      <%@ include file="/DisplayHeading.html" %>
7
8      <p>This page uses a custom tag with a body</p>
9
10     <jsp:useBean id="itemBean" class="com.best.db.InventoryBean"
11                 scope="page" />
12
13     <table border="1">
14     <tr>
15         <td><b>Description</b></td>
16         <td><b>Price</b></td>
17         <td><b>Quantity</b></td>
18     </tr>
19     <best:iterateItem
20         category="<%= request.getParameter("category") %>" >
21         <tr>
22             <td><jsp:getProperty name="itemBean"
23                             property="description"/></td>
24             <td><jsp:getProperty name="itemBean"
25                             property="price"/></td>
26             <td><jsp:getProperty name="itemBean"
27                             property="quantity"/></td>
28         </tr>
29     </best:iterateItem>
30     </table>
31 </html>
32 </body>
```

FIGURE 6. *The `BodyTagDemo.jsp` page.*

Within the body of this tag, properties of a bean named “itemBean” (which has page scope) are displayed as table data in a table row.

- Notice the attribute named “category” is set to the value of a request parameter named “category”.

The tag handler class for the `<best:iterateItem>` tag is defined here in Figure 7.

```
1  package com.best.tags;
2
3  import com.best.db.*;
4  import javax.servlet.jsp.tagext.*;
5  import javax.servlet.jsp.*;
6  import java.util.*;
7
8  public class IterateItemTag extends BodyTagSupport
9  {
10     private PageContext pageContext;
11     private BodyContent bodyContent;
12     private String category;
13     private InventoryDB invDB;
14     private Iterator iterator;
15
16     public void setPageContext(PageContext p)
17     {
18         pageContext = p;
19     }
20
21     public void setBodyContent(BodyContent b)
22     {
23         bodyContent = b;
24     }
25
26     public void setCategory(String c)
27     {
28         category = c;
29     }
30
31     public String getCategory()
32     {
33         return category;
34     }
35
36
37     public void doInitBody() throws JspException
38     {
39         System.out.println("Inside doInitBody()");
40     }
41
42     public int doStartTag() throws JspException
43     {
44         System.out.println("Inside doStartTag()");
45         try
46         {
47             invDB = new InventoryDB();
```

```
48         if(category == null)
49         {
50             iterator = invDB.getAllItems().iterator();
51         }
52         else
53         {
54             iterator =
55                 invDB.getItemInCategory(category).iterator();
56         }
57     }catch(Exception e)
58     {
59         throw new JspException(e.getMessage());
60     }
61
62     if(iterator != null && iterator.hasNext())
63     {
64         pageContext.setAttribute("itemBean",
65                                 iterator.next());
66         return EVAL_BODY_BUFFERED;
67     }
68     else
69     {
70         return SKIP_BODY;
71     }
72 }
73
74 public int doAfterBody() throws JspException
75 {
76     System.out.println("Inside doAfterBody()");
77     if(iterator.hasNext())
78     {
79         pageContext.setAttribute("itemBean",
80                                 iterator.next());
81         return EVAL_BODY_AGAIN;
82     } else
83     {
84         return SKIP_BODY;
85     }
86 }
87
88 public int doEndTag() throws JspException
89 {
90     System.out.println("Inside doEndTag()");
91     try
92     {
93         if(bodyContent != null)
94             bodyContent.writeOut(
95                 bodyContent.getEnclosingWriter());
96     }catch(java.io.IOException e)
```

```
97         {  
98             throw new JspTagException(e.getMessage());  
99         }  
100         return EVAL_PAGE;  
101     }  
102 }
```

FIGURE 7. *The IterateItemTag class.*

Some comments about the IterateItemTag class:

- The class extends BodyTagHandler. This tells the container which type of tag handler this class is so the container knows which methods to invoke.
- The PageContext and BodyContent are stored as fields in the class, which is typical of a BodyTagHandler.
- The items to be displayed are stored in a java.util.Iterator field, which is initialized in the doStartTag() method.
- If the Iterator is empty, doStartTag() returns SKIP_BODY.
- The information is shared between the JSP page and tag handler using the PageContext object. The tag handler adds an InventoryBean object named “itemBean” to the “page” scope object.
- The JSP page retrieves data in the “itemBean” object using the `<jsp:getProperty>` tag.
- No HTML formatting appears in the tag handler. The formatting is all done within the JSP page.
- Output is buffered in the BodyContent object. When the tag is done being evaluated, this output is sent to the response using the writeOut() method of the BodyContent object. This is typically done in the doEndTag() method, as demonstrated here.

Modifying the TLD

Be sure to add the `<tag>` entry for the `<itemIterator>` tag in the tag library descriptor. Figure 8 shows the “best.tld” descriptor with an entry for our new tag. This file was deployed to the `%CATALINA_HOME%\webapps\best\WEB-INF\tlds` directory.

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE taglib
3      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
4      1.2//EN"
5      "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
6  <taglib>
7      <tlib-version>1.0</tlib-version>
8      <jsp-version>1.2</jsp-version>
9      <short-name></short-name>
10     <tag>
11         <name>showItems</name>
12         <tag-class>com.best.tags.ShowItemsTag</tag-class>
13         <body-content>empty</body-content>
14         <attribute>
15             <name>category</name>
16             <required>>false</required>
17             <rtexprvalue>>true</rtexprvalue>
18         </attribute>
19     </tag>
20     <tag>
21         <name>iterateItem</name>
22         <tag-class>com.best.tags.IterateItemTag</tag-class>
23         <body-content>JSP</body-content>
24         <attribute>
25             <name>category</name>
26             <required>>false</required>
27             <rtexprvalue>>true</rtexprvalue>
28         </attribute>
29     </tag>
30 </taglib>
```

FIGURE 8. *The best.tld tag library descriptor.*

NOTE

For a tag with a body, the `<body-content>` element must appear within the `<tag>` element and be set to “JSP”.

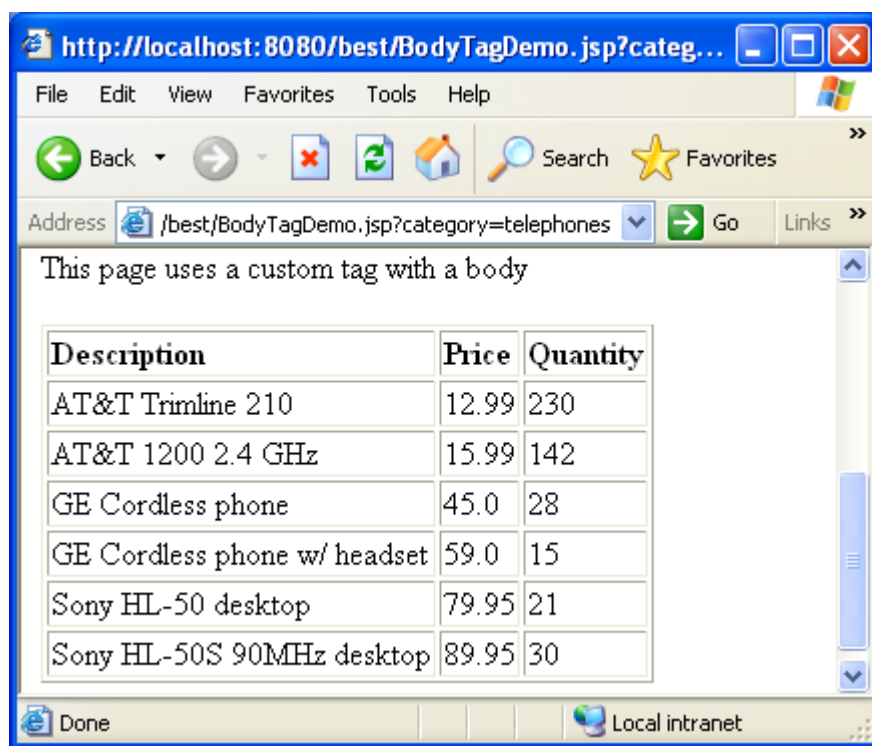


FIGURE 9. The output of the *BodyTagDemo.jsp* page.

The TryCatchFinally Interface

A tag handler that implements Tag, IteratorTag or BodyTag has the option of also implementing the TryCatchFinally interface. This interface contains two methods:

```
public void doCatch(Throwable t) throws Throwable
```

This method is invoked if a Throwable occurs while evaluating the body of a tag, or if a Throwable occurs within doStartTag(), doInitBody(), doAfterBody(), and doEndTag().

```
public void doFinally()
```

This method is always invoked after doEndTag() and much like a finally block of code. This is a useful method for freeing resources and similar finally tasks.

- The doCatch() method is useful for handling exceptions in the body of a tag, because you probably don't want a try/catch block in your JSP page. You can also use doCatch() as a "catch all" block for exceptions that might otherwise have fallen through the call stack uncaught.
- The doFinally() method is basically used for freeing resource or maintaining data integrity.

```
1  public class IterateItemTag2 extends BodyTagSupport implements
TryCatchFinally
2  {
3      private PageContext pageContext;
4      private BodyContent bodyContent;
5      private String category;
6      private InventoryDB invDB;
7      private Iterator iterator;
8
9      public void doCatch(Throwable t) throws Throwable
10     {
11         System.out.println("An exception occurred: " +
t.getMessage());
12     }
```

```
13
14     public void doFinally()
15     {
16         System.out.println("Inside doFinally()");
17         iterator = null;
18         invDB = null;
19     }
20     //remainder of the class stays the same
21 }
```

FIGURE 10. *The IterateItemTag2 handler implements TryCatchFinally.*

To generate an exception, the following code was added to a JSP page named CatchFinallyDemo.jsp:

```
<best:iterateItem category="<%= request.getParameter("category")%>" >
<tr>
<td><jsp:getProperty name="itemBean" property="description"/></td>
<td><jsp:getProperty name="itemBean" property="price"/></td>
<td><jsp:getProperty name="itemBean" property="quantity"/></td>
<td><jsp:getProperty name="itemBean" property="notThere"/></td>
</tr>
</best:iterateItem>
```

The “notThere” property is not a property of the bean, so the body of this tag will generate an Exception that will be “caught” by the doCatch() method.

The following output is from System.out when the CatchFinallyDemo.jsp page is viewed in a browser:

```
Inside doStartTag()
Inside doInitBody()
An exception occurred: Cannot find any information on property
'notThere' in a bean of type 'com.best.db.InventoryBean'
Inside doFinally()
```

Lab 1: Writing a Custom JSP Tag

Objective: To become familiar with the process of writing a custom JSP tag. You will write the class and TLD file to create a custom tag for viewing the contents of the a shopping cart.

Perform the following tasks:

1. Write a class named ShowCartTag that extends TagSupport. Declare it in the com.best.tags packages. This class is going to be the tag handler for a tag that looks like <best:showCart />.
2. Add a field of PageContext and BodyContent, and initialize them in the setPageContext() and setBodyContent() methods, respectively.
3. Add a field of type java.util.Iterator. In the doStartTag() method, initialize this field to be the Iterator returned from the ShoppingCart of the customer. The ShoppingCart object should be a "cart" attribute of the session object.
4. Within doStartTag(), if the iterator isn't empty, then set a page attribute named "item" to be the next() element in the Iterator and return EVAL_BODY_BUFFERED. If the iterator is empty or null, return SKIP_BODY.
5. Within the doAfterBody() method, if the Iterator has a next() element, then set the "item" page attribute to be the next element and evaluate the body again. Otherwise, skip the body.
6. In the doEndTag() method, write out the body content to the response and return EVAL_PAGE.
7. Save and compile your ShowCartTag class.
8. Write a TLD file for your custom tag.
9. Modify your managecart.jsp page so that it uses the <best:showCart/> tag instead of using a scriptlet.

10. On your managecart.jsp page, add a link at the bottom of the page for checking out. The URL should be "controller?action=checkout". (In this lab, the link will not actually work. In the next chapter, you will write a servlet and JSP page for checking out items in the shopping cart.)
11. Open the pages in a browser and verify that everything is working properly.

What you should see: The output will be similar to what you have seen already from your managecart.jsp page.

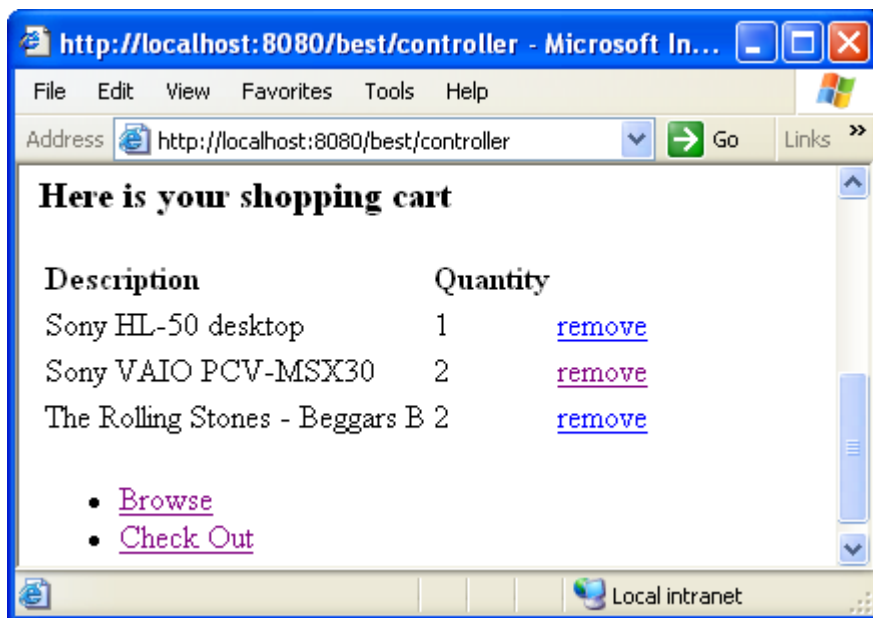


FIGURE 11. The output should look similar to the previous managecart.jsp page.



This chapter discusses the details of developing Java applications that use CORBA. Topics discussed in this chapter include:

- An overview of CORBA
- The Interface Definition Language
- CORBA Services
- The Portable Object Adapter (POA)
- Writing an IDL interface
- Implementing CORBA in Java
- Writing a CORBA client in Java
- Using a CORBA naming service

The Object Management Group

CORBA stands for *Common Object Request Broker Architecture*. The term describes the distributed object architecture developed by the *Object Management Group* (OMG). The OMG is a consortium of over 1,000 companies, created in 1989, whose goal is to establish a specification for the standards of distributed object computing.

NOTE

One of the main tasks of the OMG is to maintain the standards of the Interface Definition Language (IDL). IDL is the language used to define the interface of a CORBA object.

CORBA is not a programming language. It is a specification that describes the services and attributes of a CORBA implementation. The OMG defines the standards and guidelines of the CORBA architecture.

The OMG does not implement the standards that they define. Instead, third-party vendors provide the implementation for CORBA in software products like ORBs, application servers, Web browsers, middleware applications, and so on.

Overview of CORBA

A CORBA object is a software component that provides various services. The advantages of a CORBA object include:

- CORBA objects can be accessed from any client.
- The implementation of the CORBA object can be in any object-oriented programming language.
- The client does not need to know the location of the server where the object is located.
- The client does not need to know the language that the CORBA object is implemented in.

The CORBA specification defines the set of services provided by an *Object Request Broker*, commonly referred to as an ORB. The term ORB is quite relevant in this situation:

- A client wants to locate an object that will provide a service that the client needs.
- An object wants to advertise its services to potential clients. This object will use an ORB to provide this advertising of services.
- The client will pass a request to the ORB, without knowing the location of the object that will fulfill the request.
- The ORB locates an object to handle the request, passes the request to the remote object, and sends back any return information to the client.

Java IDL

CORBA uses the *Interface Definition Language* (IDL) for creating language independent interfaces for distributed objects. The Java IDL is a mapping between IDL and Java. You use the `idltojava` compiler, freely downloadable from Sun's web site, to create the necessary stubs and skeletons required by the ORB.

NOTE

The Java IDL is not an API, rather a mapping of IDL data types to Java data types. The necessary classes and APIs for using CORBA are found in the `org.omg` packages of the JDK.

The IDL is what makes CORBA language-independent. The ORB only has to understand one language: IDL. Note that IDL is strictly a definition language. It does not contain any programming constructs. An ORB is not concerned about implementations, anyway. An ORB does not care how an interface is implemented. It only needs to be aware of the interface that a particular object implements.

The diagram in Figure 1 demonstrates how an ORB uses IDL to communicate with implementations in any language.

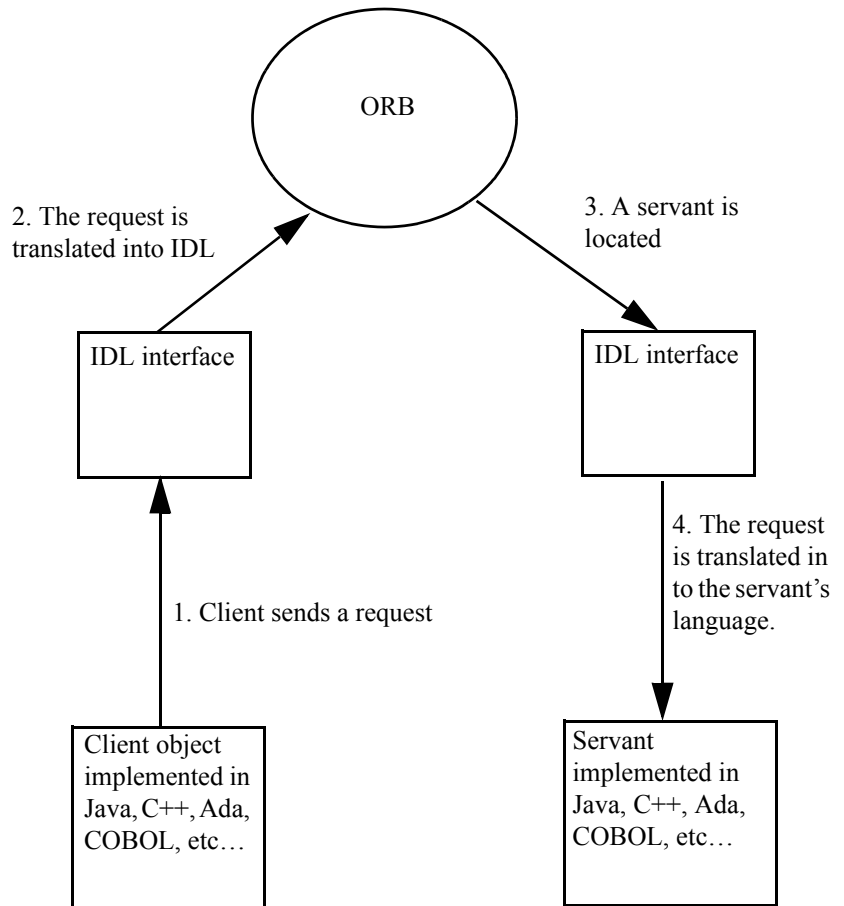


FIGURE 1. *An ORB understands one language: IDL.*

In Figure 1, if the servant method invoked by the client had a return value, then the value is passed back to the client via the ORB.

The ORB Architecture

Figure 2 shows the ORB architecture involved with a client passing a request to an ORB. The stub and skeleton provide a key role for the ORB, since this is where the marshalling and unmarshalling of parameters occurs.

The stub is responsible for taking the arguments of the function call, which are native data types of the client's programming language, and converting them to IDL data types that the ORB can understand. The stub then passes these IDL data types to the ORB. The ORB passes these arguments on to the skeleton.

The skeleton is responsible for unmarshalling the parameters, which involves mapping the IDL types to the corresponding native types on the server. The skeleton then invokes the appropriate method on the servant.

Any return values are passed back in a similar fashion, with the skeleton marshalling the return value and the stub unmarshalling it and returning it to the client.

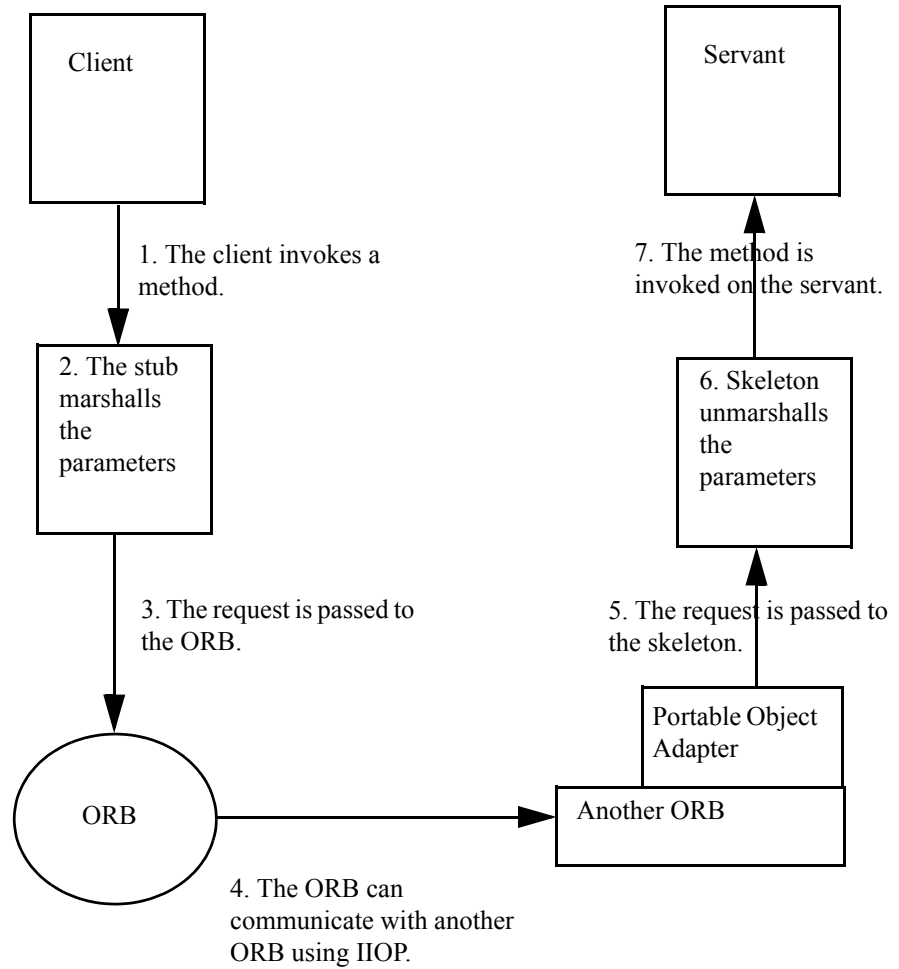


FIGURE 2. *The ORB architecture.*

Services of the ORB

An ORB provides various services that a client and server can use in assisting the brokering of the client request. These services are accessible by your Java code through an API either provided by the ORB vendor or third-party software.

The common services that an ORB provides include:

- **Naming Service:** ORBs use the Corba Object Services (COS) naming service. COS is a tree-like structure of object references. The references are stored on a context in the tree, and a client can locate the object reference by searching for various contexts. The COS naming service is accessed via the `org.omg.CosNaming` package of the JDK.
- **Transaction Service:** Transactions can be complex in a distributed environment. The ORB provides the client and server with the ability to start, commit and rollback transactions. The transaction services of an ORB are accessed via the Java Transaction API (JTA).
- **Event Service:** Events allow for a client request to be asynchronously, with an event sent back to the client instead of a return value from a method call. Event services consist of an event channel, with events either being pushed or pulled through the channel.

Notes

The Portable Object Adapter (POA)

Another feature of the ORB is what is called the *portable object adapter*, or POA. The portable object adapter is responsible for communicating directly with the servant, and it routes requests from the ORB to the servant. The POA can also be initialized to control various aspects about the servant, like multithreading or persistence.

The behavior of a POA is determined by a set of policies. For example, there is a lifespan policy that can be set to either TRANSIENT or PERSISTENT. A persistent servant typically is associated with data in a database and, therefore, must have a lifetime beyond that of the server application and represent stateful information. Transient servants typically do not have state and can therefore be removed by the POA without any concern.

There is also a thread policy that is either SINGLE_THREAD_MODEL or ORB_CTRL_MODEL. The single thread model tells the POA to ensure that no two client requests are invoked concurrently on the same servant. The other model tells the POA it can simply start a new thread for concurrent client requests on the servant.

An ORB can have multiple POAs, which allows you to create a POA with policies tailored specifically for a servant's needs. Every ORB has a Root POA from which child POAs can be created. The following diagram illustrates how different POAs can be used with different policies to handle servants differently.

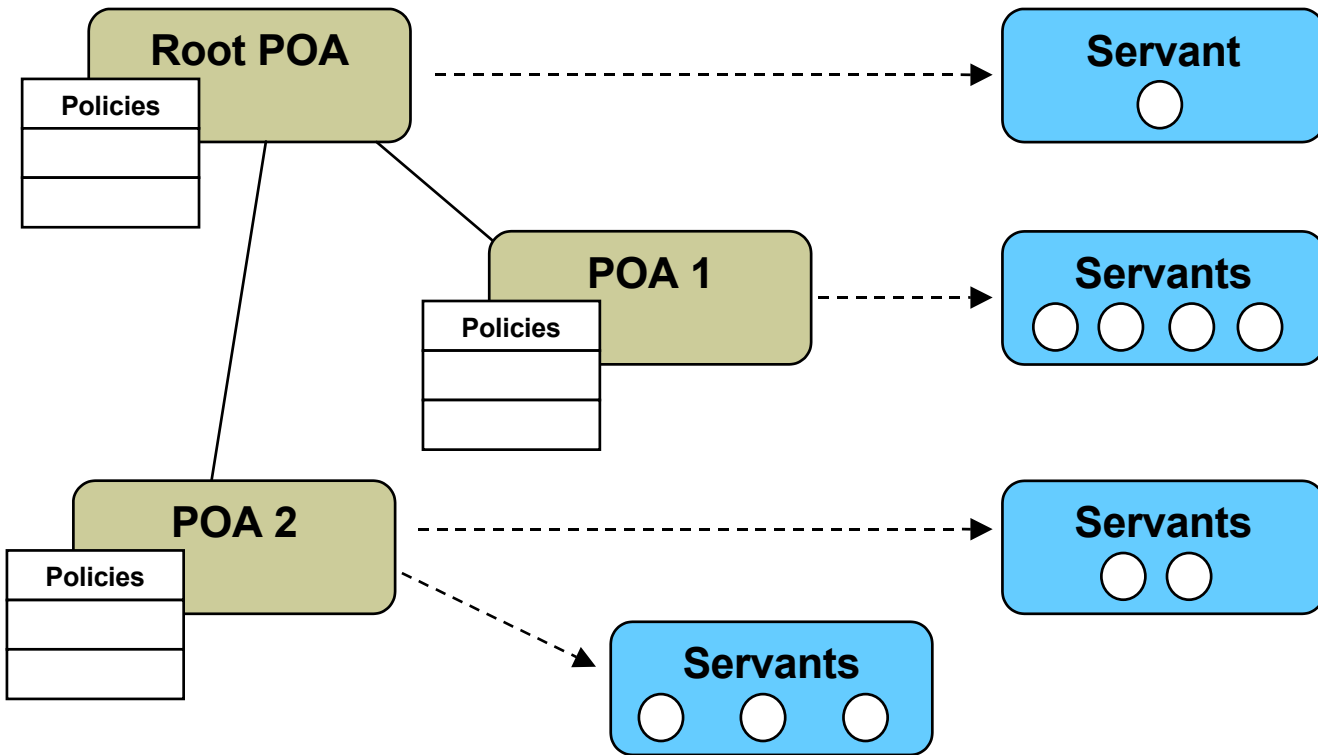


FIGURE 3. A POA consists of various policies.

The Interface Definition Language

IDL is the language of ORBs. All methods and attributes of a servant need to be exposed in an interface written in IDL. The IDL language is not a programming language. It is a definition language for defining types of objects and the methods and attributes they provide.

The following is a list of IDL keywords:

any	default	inout	out	switch
attributedouble		interface	raises	TRUE
boolean	enum	long	readonly	typedef
case	exception	module	sequence	unsigned
char	FALSE	Object	short	union
const	float	octet	string	void
context	in	oneway	struct	

Notice that IDL does not contain programming constructs like if, for and while. This is because IDL is not used for programming. The keywords above map to data types and other Java keywords used for defining classes. Notice that IDL maps closely to the C++ language, which was the most widely used object oriented language when the IDL was first established.

The IDL to Java Mapping

Each of the keywords in the IDL maps to a keyword in the Java programming language. For example, module maps to package:

```
//in IDL
module banking
{ };
```

would be the same as a package declaration

```
package banking;
```

Data Types

The primitive data types of IDL map to the following Java types:

TABLE 1. Mapping IDL data types to Java data types.

IDL Type	Java Type
boolean	boolean
char	char
wchar	char
octet	byte
string	java.lang.String
wstring	java.lang.String
short	short
unsigned short	short
long	int
unsigned long	int
long long	long
unsigned long long	long
float	float
double	double

- The IDL literal null maps to null in Java, and it's no surprise that TRUE and FALSE in IDL map to true and false in Java, respectively. Also, the IDL void keyword maps to void in Java.
- The keyword string maps to java.lang.String.
- An IDL interface maps to an interface in Java with the same name.
- The const field in an IDL interface maps to a public static final field of the corresponding Java interface. For example,

```
//IDL
module banking
{
    interface Teller
    {
        const string greeting = "Welcome";
    };
};

//Java
package banking;

public interface Teller
{
    public static final java.lang.String greeting = "Welcome";
}
```

- A const field in IDL for a global variable maps to an interface of the same name as the field, with a static final field named value within this interface. For example,

```
//IDL
module banking
{
    const string greeting = "Welcome";
};
```



```
//Java
package banking;

public interface greeting
{
    public static final java.lang.String value = "Welcome";
}
```

- An IDL struct maps to a final class in Java.
- An IDL union maps to a final class in Java that contains method named `discriminator()` to emulate the behavior of a union.
- An IDL sequence maps to an array in Java of the corresponding data type. A sequence is created in IDL by first using `typedef` to create an array data type. For example,

```
//IDL
typedef sequence<octet, 5> smallArray;
typedef sequence<long> anySizeArray;

smallArray scores;
anySizeArray playerIDs;

//Java
byte [] scores;
int [] playerIDs;
```

- The `typedef` IDL keyword does not map to Java. When using `typedef` for primitive data types, the Java type is used directly. For complex typedefs, the types are "unnested" using Java classes.

Writing an IDL Interface

When implementing CORBA, you need to write an IDL interface to expose the various methods of the servant that are to be available to a client. The methods of the servant must be defined in an IDL file, which is then compiled into the corresponding Java interface.

An IDL file is a text file and is typically saved with the *.idl extension. An idltojava compiler is used to verify the syntax of the IDL interface. The idltojava compiler also generates several Java interfaces and classes, including the stub and skeleton.

Adding Attributes

An interface can contain attributes, which map to accessor and mutator methods in the Java interface. An attribute is denoted using the IDL keyword `attribute`, and it can also be denoted as `readonly`, which results in only an accessor method being created.

Consider the following IDL interface, which was created in a file called "football.idl".

```
1  module sports
2  {
3      interface Football
4      {
5          attribute long homeScore;
6          attribute long visitorsScore;
7          readonly attribute string location;
8      };
9  };
```

The Java interface generated from the "football.idl" interface looks like:

```
10 package sports;
11
12 public interface Football
13 {
14
15     public int homeScore ();
16
17     public void homeScore (int homeScore);
18
19     public int visitorsScore ();
20
21     public void visitorsScore (int visitorsScore);
22
23     public java.lang.String location ();
24
25 }
```

Notice the names of the methods match the names of the attributes. The `location()` method does not contain a mutator method since it was a `readonly` attribute.

Adding Methods

An IDL interface can contain methods, which do not require any special IDL keyword. Methods are denoted by their signature, which will contain a parameter list in parentheses. The method signature also needs to declare a return value.

For example, the following IDL module contains an interface with an `addToScore()` method. The method has an `in` parameter that is added to the `score` attribute:

```
module sports
{
    interface Basketball
    {
        attribute long score;
        void addToScore(in octet points);
    };
};
```

The corresponding Java interface looks like:

```
package sports;

public interface Basketball {

    public int score ();

    public void score (int score);

    public void addToScore (byte points);

}
```

Out Parameters

Keep in mind that CORBA maps to different programming languages, and many languages support the concept of an out parameter. However, there is no analogous concept of an out parameter in Java.

In those situations, a class object is needed to simulate the effect of an out parameter. Since this can be a common occurrence, the `idltojava` compiler always generates a Holder class for every data type defined in an IDL file.

The following interface contains a method with an out parameter:

```
module sports
{
    interface Basketball
    {
        attribute long score;
        void addToScore(in octet points, out long newScore);
    };
};
```

The generated Java interface looks something like:

```
package sports;

public interface Basketball {

    public int score ();

    public void score (int score);

    public void addToScore (byte points,
                           org.omg.CORBA.IntHolder newScore);

}
```

NOTE

Notice the out parameter is replaced with an `IntHolder` object. There is a Holder class for each of the Java primitive data types, the mapped data types like `String`, and any new data type that is defined in an IDL. (The `idltojava` compiler generates the Holder classes for you.)

Compiling the IDL

The `idltojava` is used to generate the necessary stubs and skeletons for the Java implementation of the IDL interface. The `idltojava` compiler can be downloaded freely from Sun's Javasoft web site. More commonly, you will use the `idltojava` compiler provided by the vendor of the particular ORB you are using.

The following is a list of the files generated from the `idl2java` compiler that comes with Inprise's Visibroker product. (The IDL interface was the "football.idl" file defined above).

- `Football.java`: The declaration of the Football interface in the sports package.
- `FootballOperations.java`: A subinterface of Football, it contains the method signatures and any constant attributes defined in the IDL interface.
- `FootballHelper.java`: A class that contains various static "helper" methods. You will use the helper class to narrow references and to bind references with the ORB.
- `FootballHolder.java`: The holder class for the Football interface. This class will be used for any methods that declare an out parameter of type Football.
- `_FootballStub.java`: The stub for the client side.
- `FootballPOA.java`: Used by the POA to communicate with the servant and invoke the methods of the Football interface.
- `FootballPOAtie.java`: This class "ties" the IDL interface method signatures to those in the Football interface. This class will contain the identical methods as the Football interface.

The following class is the FootballHolder class. The holder class look similar for any data type defined in an IDL file.

```
1  package sports;
2
3  public final class FootballHolder implements
org.omg.CORBA.portable.Streamable {
4      public sports.Football value;
5
6      public FootballHolder () {
7          }
8
9      public FootballHolder (final sports.Football _vis_value) {
10         this.value = _vis_value;
11     }
12
13     public void _read (final org.omg.CORBA.portable.InputStream
input) {
14         value = sports.FootballHelper.read(input);
15     }
16
17     public void _write (final org.omg.CORBA.portable.OutputStream
output) {
18         sports.FootballHelper.write(output, value);
19     }
20
21     public org.omg.CORBA.TypeCode _type () {
22         return sports.FootballHelper.type();
23     }
24 }
```

Lab 1: Writing an IDL interface

Objective: To become familiar writing and compiling an IDL file. You will write an IDL file that will be used to represent a bank account object.

Perform the following tasks:

1. Create an IDL file called "bank.idl". Save the file in a new folder on your PC called "CorbaBank".
2. Add a module called bank.
3. Within the bank module, add an interface called BankAccount.
4. The BankAccount interface has two attributes: a double called balance and a string called name.
5. Add a method called deposit() to BankAccount that takes in a double and returns a double. (It is going to return the new balance.)
6. Similarly, add a withdraw() method that takes in a double and returns a double.
7. Add another interface to the bank module called AccountManager.
8. Add a method to AccountManager called openAccount() that takes in a string called name and returns a BankAccount.
9. Save the "bank.idl" file.
10. Use the idl2java compiler to generate the Java interfaces and classes.

What you should see: The idl2java compiler will generate the various classes and interfaces for the two interfaces you defined in the IDL.

Defining Exceptions

The IDL contains an exception keyword for defining an exception. Exceptions are similar to structs, and they map to Java classes. You can define an exception, then have a method declare that it throws the exception by using the `raises` IDL keyword.

The following IDL file shows a module that contains a definition of an exception called `InvalidPoint` and a method that throws the exception.

```
25  module sports
26  {
27      exception InvalidPoint
28      {
29          octet point;
30      };
31
32      interface Basketball
33      {
34          attribute long score;
35          void addToScore(in octet points) raises (InvalidPoint);
36      };
37  };
```

The corresponding interface for `Basketball` will look like:

```
package sports;

public interface BasketballOperations {
    public int score ();
    public void score (int score);
    public void addToScore (byte points) throws sports.InvalidPoint;
}
```

The corresponding Java code for `InvalidPoint` looks something like:

```
38  package sports;
39
40  public final class InvalidPoint extends
org.omg.CORBA.UserException {
41      public byte point;
42  }
```

```
43     public InvalidPoint () {
44         super(sports.InvalidPointHelper.id());
45     }
46
47     public InvalidPoint (byte point) {
48         this();
49         this.point = point;
50     }
51
52     public InvalidPoint (java.lang.String _reason, byte point) {
53         super(sports.InvalidPointHelper.id() + ' ' + _reason);
54         this.point = point;
55     }
56
57     public java.lang.String toString () {
58         final java.lang.StringBuffer _ret =
59             new java.lang.StringBuffer("exception sports.InvalidPoint
60 {");
61         _ret.append("\n");
62         _ret.append("byte point=");
63         _ret.append(point);
64         _ret.append("\n");
65         _ret.append("}");
66         return _ret.toString();
67     }
68
69     public boolean equals (java.lang.Object o) {
70         if (this == o) return true;
71         if (o == null) return false;
72         if (o instanceof sports.InvalidPoint) {
73             final sports.InvalidPoint obj = (sports.InvalidPoint)o;
74             boolean res = true;
75             do {
76                 res = this.point == obj.point;
77             } while (false);
78             return res;
79         }
80         else {
81             return false;
82         }
83     }
```

Notice the class contains a field called point and a constructor was added to initialize this field.

Lab 2: Defining Exceptions

Objective: To become familiar defining exceptions in IDL. You will have the `withdraw()` method of the `BankAccount` interface throw an exception if the account has insufficient funds.

Perform the following tasks:

1. You will be modifying the "bank.idl" file you created in the previous lab. Open the file in a text editor.
2. Within the bank module, define an exception called `InsufficientFunds`. It will contain a single field called `amount` of type `double`.
3. Have the `withdraw()` method of the `BankAccount` interface declare that it raises an `InsufficientFunds` exception.
4. Save the "bank.idl" file.
5. Run the `idl2java` compiler to generate the Java interfaces and classes.

What you should see: The `idl2java` compiler will generate a class for your newly-defined exception.

Implementing CORBA on the Server

We are now ready to write an application that runs on the server to instantiate a servant and initialize it with an ORB. The following is a list of steps we need to follow to accomplish this task.

1. Implement the IDL interfaces: You need to write a Java class that contains the implementation of the interfaces declared in the IDL interface.
2. Initialize the ORB: The servant needs to be instantiated and bound with the ORB. To accomplish, you need to first make sure the ORB is initialized.
3. Create the POA: Use the ORB initialized in the previous step to create a POA with the desired policies.
4. Generate an IOR for the servant: Servants are located by using an Interoperable Object Reference (IOR). You generate the IOR and make it accessible to the client through a file or some other resource.
5. Run the ORB: When all is initialized, your servant needs to wait for incoming requests.

Notes

Step 1: Implement the IDL Interfaces

The first step is to write a Java class that implements the methods defined in the IDL interface. The `idltojava` compiler generates the interface and also a class for you to extend. The following diagram illustrates the relationship between the generated code and the class that you must create, which you will typically name `InterfaceNameImpl`:

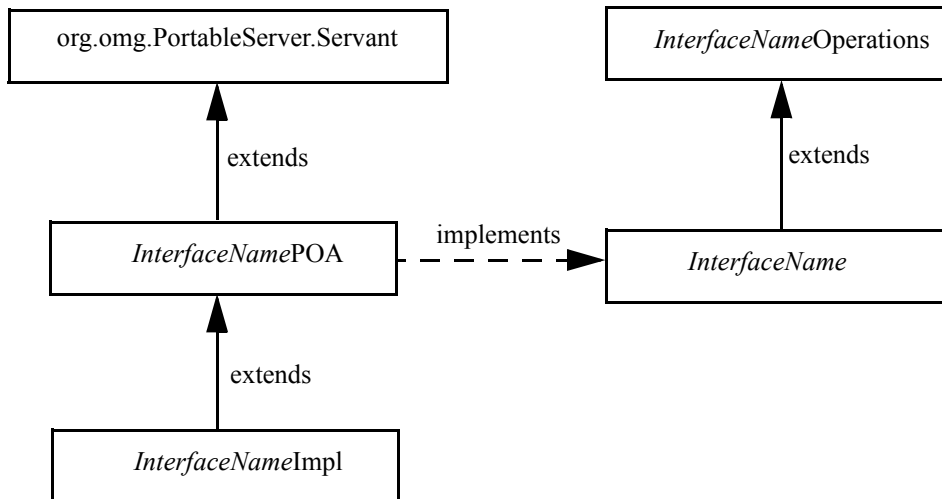


FIGURE 4. You typically write a class that extends the generated POA class.

The following example shows an implementation of the Basketball interface defined earlier.

```
1 import sports.*;
2
3 public class BasketballImpl extends sports.BasketballPOA
4 {
5     private int score;
6
7     public void score(int newScore)
8     {
9         score = newScore;
10    }
11
12    public int score()
13    {
14        return score;
15    }
16
17    public void addToScore(byte points) throws InvalidPoint
18    {
19        if(points >= 0 && points <= 3)
20        {
21            score += points;
22        }
23        else
24            throw new InvalidPoint("Not a valid point", points);
25    }
26 }
```

FIGURE 5. *The BasketballImpl class.*

Step 2: Initialize the ORB and the POA

The next step in using CORBA on the server is to create the servant and bind it with the ORB. To accomplish this task, the ORB needs to be initialized. In addition, a POA needs to be created with the various policies that you want the servant to have.

To initialize the ORB, you use the static `init()` method in the `org.omg.CORBA.ORB` class, which is declared as:

```
public static ORB init(String [] args, Properties props);
```

The `args` parameter is typically the command-line arguments of the application, which may contain various parameters to set when initializing the ORB. The `props` argument is also used for properties and may contain various parameters as well. Both arguments can be null.

Notice that the `init()` method returns a reference of type `ORB`, which is a class in the `org.omg.CORBA` package. This package is a part of the core Java API. The `ORB` class contains methods for accessing the features of the ORB. Some of the more commonly-used methods in the `ORB` class include:

```
public Object resolve_initial_references(String name) throws  
InvalidName
```

used to locate a reference to the various services provided by the ORB. Examples of name might include "RootPOA", "NamingService", "EventService", and so on.

```
public String object_to_string(Object obj)
```

converts the `Object` reference to a `String` that satisfies the IIOP protocol. This `String` representation is referred to as the IOR. The server will use this method to create an IOR for a servant.

```
public Object string_to_object(String str)
```

converts the given String back to its corresponding CORBA object reference. The client might use this method to convert an IOR to the actual servant reference.

```
public void run()
```

causes the thread invoking this method to wait. This method does not return until the ORB has been shut down.

The following program demonstrates an ORB being initialized.

```
1 public class BasketballServer
2 {
3     public static void main(String [] args)
4     {
5         try
6         {
7             System.out.println("Initializing the ORB");
8             org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
9         }catch(Exception e)
10        {
11            e.printStackTrace();
12        }
13    }
14 }
```

FIGURE 6. *Initializing the ORB*

Step 3: Create the POA

A servant communicates with the ORB through a portable object adapter (POA). When implementing CORBA, you need to create a POA with the policies that you desire for the servant. Note that each servant does not need its own POA. A POA can be associated with any number of servants.

Locating the Root POA

The first step in creating a POA is to locate the root POA of the ORB. All POAs are created from an existing POA, creating a tree of adapters all starting from the root POA. To obtain a reference to an ORB's root POA, use the `resolve_initial_references()` method of the ORB class, as the following statement illustrate:

```
org.omg.CORBA.Object objRef =  
    orb.resolve_initial_references("RootPOA");
```

The return value is of type `org.omg.CORBA.Object` because the `resolve_initial_references()` method is used for obtaining references to many different objects. In the statement above, the `objRef` reference needs to be casted to an `org.omg.PortableServer.POA` object. Casting in CORBA should never be done using the cast operator.

Instead, you should use the Helper class that every CORBA data type has. Each Helper class contains a method called `narrow()` for "narrowing" a reference down to its actual data type, as the following statement demonstrates:

```
POA rootPOA = POAHelper.narrow(objRef);
```

Setting the Policies

A POA does not inherit the policies of its parent POA, and you can not change a POA's policies once it is created. Therefore, you need to set the policies of the POA when it is initially created.

NOTE

The policies of a POA are placed in an array of type `org.omg.CORBA.Policy`. A Policy object is created by using the `create_xxx_policy()` methods of the POA class.

For example, the following statement creates a lifespan policy of `PERSISTENT`, meaning that the servants in this POA can outlive the process that created them. (The default lifespan policy is `TRANSIENT`.)

```
org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
    rootPOA.create_request_processing_policy(
        RequestProcessingPolicyValue.USE_DEFAULT_SERVANT),
    rootPOA.create_id_uniqueness_policy(
        IdUniquenessPolicyValue.MULTIPLE_ID)
};
```

Creating the POA

Once the policies are ready, invoke the `create_POA()` method on the POA that you are branching out from. For example,

```
POA sportsPOA = rootPOA.create_POA("sports_poa",
                                   rootPOA.the_POAManager(),
                                   policies);
```

The first argument of `create_POA()` is the name of the new POA. The second argument is the manager of the POA. Every POA can have a manager that monitors whether the POA is processing or not, among other monitoring capabilities. Typically, you will have a single POA manager that monitors all of your POAs. Note you will need to `activate()` the manager if you plan on using this functionality:

```
rootPOA.the_POAManager().activate();
```

Step 4: Generate an IOR for the servant

Once the ORB is initialized and the POA has been created, you are now ready to instantiate the servant (object on the server that handles client requests). The servant needs to then be associated with a POA. Finally, an Interoperable Object Reference (IOR) is created for the servant, which allows a client to be able to uniquely identify a desired servant to handle a request.

Instantiating the Servant

The following statements instantiate a servant and associate it with a POA. Use the `set_servant()` method of the POA class to assign a servant to a POA:

```
BasketballImpl bball_servant = new BasketballImpl();  
sportsPOA.set_servant(bball_servant);
```

The `bball_servant` is now set with a POA and will become available for client requests when the ORB is run.

Generating an IOR

A servant can be located by a client using an IOR, which is a persistent reference that be stored in a file or database. This reference is not a pointer to the servant on the server, but instead is a unique identifier for locating the particular servant.

The POA class contains a `create_reference_with_id()` method for creating an IOR using an ID:

```
org.omg.CORBA.Object iorRef;  
iorRef = sportsPOA.create_reference_with_id("Basketball".getBytes(),  
    "IDL:sports/Basketball:1.0");  
  
String iorString = orb.object_to_string(iorRef);
```

The first argument is an identifier for the reference, which needs to be an array of bytes. The second argument is a repository ID which can be found in the comments of the interface code generated from the `idltojava` compiler. In the statements above, the resulting IOR is converted to a `String` using the `object_to_string()` method of the ORB. This method converts the reference into a readable `String` that can be written to an output stream.

A typical technique for a client to obtain an IOR is to output the `String` version of the IOR to a file or database. The following statements demonstrate how you might write the `String` to a file.

```
PrintWriter out = new PrintWriter(new FileWriter("bball.ior"));  
out.println(iorString);  
out.close();
```

The file `bball.ior` can be given to the client and the string IOR can be retrieved by the client using a `PrintReader` and `FileReader` object.

Step 5: Run the ORB

Once the servant is instantiated and associated with a POA, the servant is now ready to become activated and handle requests from clients. The servant does not get activated until the `run()` method is invoked on the ORB:

```
orb.run();
```

The `run()` method does not return until the ORB is shut down, which can be accomplished using the `shutdown()` method of the ORB class.

The Server Application

Now that you have seen all of the pieces involved, you are ready to write an application that runs on the server to create and activate a servant. The following `BasketballServer` class puts together all of the steps discussed above.

When running the server application, you can specify a host where the ORB is running using the `ORBagentAddr` property. For example,

```
java -DORBagentAddr=my_server BasketballServer
```

When the `ORB.init()` method is invoked, the ORB on "my_server" will be the initial host where your application looks. If no ORB is found on "my_server", then your application will broadcast the network looking for an ORB.

```
1 import org.omg.PortableServer.*;
2 import java.io.*;
3
4 public class BasketballServer
5 {
6     public static void main(String [] args)
7     {
```



```
8      try
9      {
10         System.out.println("Initializing the ORB");
11         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
12
13         System.out.println("Locating the root POA");
14         org.omg.CORBA.Object objRef =
15             orb.resolve_initial_references("RootPOA");
16         POA rootPOA = POAHelper.narrow(objRef);
17
18         System.out.println("Creating the POA");
19         org.omg.CORBA.Policy[] policies =
20             {
21 rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
22         rootPOA.create_request_processing_policy(
23             RequestProcessingPolicyValue.USE_DEFAULT_SERVANT),
24         rootPOA.create_id_uniqueness_policy(
25             IdUniquenessPolicyValue.MULTIPLE_ID)
26     };
27         POA sportsPOA = rootPOA.create_POA("sports_poa",
28             rootPOA.the_POAManager(), policies);
29
30         System.out.println("Activating the POA manager");
31         rootPOA.the_POAManager().activate();
32
33         System.out.println("Create servant and set with the POA");
34         BasketballImpl bball_servant = new BasketballImpl();
35         sportsPOA.set_servant(bball_servant);
36
37         System.out.println("Create an IOR and write to a file");
38         org.omg.CORBA.Object iorRef;
39         iorRef =
40 sportsPOA.create_reference_with_id("Basketball".getBytes(),
41             "IDL:sports/Basketball:1.0");
42         String iorString = orb.object_to_string(iorRef);
43
44         PrintWriter out = new PrintWriter(new
45             FileWriter("bball.ior"));
46         out.println(iorString);
47         out.close();
48
49         System.out.println("Running the orb...");
50         orb.run();
51     } catch (Exception e)
52     {
53         e.printStackTrace();
54     }
55 }
56 }
```

Lab 3: Writing the Server Application

Objective: To become familiar with writing a server application that creates a servant and activates it with a POA of an ORB.

Perform the following tasks:

1. In the previous lab, you created an IDL file and used the `idltojava` compiler to generate the necessary Java classes and interfaces to implement the IDL interfaces. Compile these classes now. You will probably need to adjust your `CLASSPATH` to include the necessary packages supplied by your ORB vendor.
2. Write a class called `BankAccountImpl` to contain the implementation of the methods in the `BankAccount` interface. This class will need to extend `BankAccountPOA`.
3. Add two private fields to `BankAccountImpl`: a double called `balance` and a String called `name`. Add the implementation of the accessor and mutator methods.
4. You also need to add the implementation of the `deposit()` and `withdraw()` methods. Have the `withdraw()` method throw an `InsufficientFunds` exception if the amount being withdrawn is more than the balance of the account. Both methods should return the new balance after the transaction has occurred.
5. At the beginning the `deposit()` method, display a message reading "Making a deposit for " + `name`. Have the `withdraw()` method display a similar message.
6. Save and compile the `BankAccountImpl` class.
7. Write a class called `BankAccountServer`. Add `main()` to this class.
8. Within `main()`, initialize the ORB and use it to obtain a reference to the root POA.

9. Using the root POA, create a new POA called "bank_poa".
10. Activate the POA manager of the root POA.
11. Instantiate a BankAccountImpl object and set it with the "bank_poa".
12. Create an IOR for the BankAccountImpl and write it to a file (using PrintWriter) called "bankaccount.ior".
13. Run the orb using the run() method of the ORB class. Print out a message before invoking run() to verify that your program has made it this far.
14. Save and compile the BankAccountServer class.
15. Make sure your ORB is running somewhere on your network.
16. Run the BankAccountServer application.

What you should see: It is more of a question of what you won't see. If it all works successfully, then you won't see any exceptions or error messages.

The Client Application

For a client to invoke a method on a servant, the client needs to perform the following steps:

1. Locate the IOR: Wherever the IOR was stored, the client needs to locate it and retrieve the string value of the IOR.
2. Initialize the ORB: Use the `init()` method of the ORB class to initialize the ORB.
3. Convert the string to a reference: if the IOR was converted into a string, it now needs to be converted back into a reference.
4. Narrow the reference: The reference needs to be narrowed to the appropriate data type using the Helper class.
5. Invoke the methods: The reference is now ready to use by the client. The servant is waiting for requests and the client can invoke any of the methods that were declared in the IDL interface.

The following program shows a client accessing the various methods of the Basketball servant created earlier. This client application demonstrates the steps involved in obtaining and using a servant reference.

```
1 import java.io.*;
2 import sports.*;
3
4 public class BasketballClient
5 {
6     public static void main(String [] args)
7     {
8         String bball_string = null;
9         try
10        {
11            System.out.println("Reading the IOR");
12            BufferedReader in = new
13                BufferedReader(new FileReader("bball.ior"));
14            bball_string = in.readLine();
15            in.close();
16        }catch(IOException e)
17        {
18            e.printStackTrace();
19            return;
20        }
21
22        try
23        {
24            System.out.println("Initializing the ORB");
25            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
26
27            System.out.println("Converting the reference");
28            org.omg.CORBA.Object bball_ref =
29                orb.string_to_object(bball_string);
30            Basketball bball = BasketballHelper.narrow(bball_ref);
31
32            System.out.println("Initial score = " + bball.score());
33
34            byte pts = 2;
35            bball.addToScore(pts);
36            pts = 1;
37            bball.addToScore(pts);
38            System.out.println("Score is now = " + bball.score());
39            pts = 10;
40            bball.addToScore(pts);
41            System.out.println("This should not be displayed");
42        }catch(Exception e)
43        {
44            e.printStackTrace();
45        }
46    }
47 }
```

Lab 4: Writing the Client

Objective: To become familiar with writing client applications that access a CORBA object. You are going to write a class that obtains a reference to a remote `BankAccount` and invokes `deposit()` and `withdraw()` on the account.

Perform the following tasks:

1. Write a class called `BankAccountClient`. Within this class, declare the `main()` method.
2. Within `main()`, read the IOR string from the file created by the `BankAccount-Server` application.
3. Initialize the ORB using the `init()` method.
4. Convert the string IOR to a CORBA object reference. Use the `narrow()` method in the `BankAccountHelper` class to narrow this reference to a `BankAccount` type.
5. Set the name on the account to be your name.
6. Make several deposits and withdrawals by invoking the `deposit()` and `withdraw()` methods of the `BankAccount` interface. Display the balance after each transaction to verify that it is working successfully.
7. Try to make a withdrawal that is larger than the balance on the account to verify that the exception is thrown properly.
8. Save and compile the `BankAccountClient` class.
9. Before running the client, be sure that the `BankAccountServer` program from the previous lab is running.

10. Run the BankAccountClient.

What you should see: On the client side, you should see the balance changing on the account. On the server side, you will see a message displayed each time a client invokes a the deposit() or withdraw() methods.

The Naming Service

The CORBA specification provides for a standard naming service that allows for a client to locate a servant reference by using a name for the servant. The entries in the naming service consists of two types of entities: contexts and objects.

The naming service consists of a root context from which all other contexts are attached, creating a tree-like data structure. The branches on the tree are contexts, and at the end of the tree can be attached objects.

Creating a tree structure allows for a client to be able to search for a servant without possibly knowing where the servant is actually located. Compare this to looking for a restaurant to eat supper at using the yellow pages. You don't know which particular restaurant you want, but you do know which city you want to eat in and what type of food you are hungry for.

A client can search a naming service tree similarly, by viewing the contexts and searching for a particular context that suits their needs.

Figure 7 illustrates what a naming service tree might look like. The circular shapes represent naming contexts, to which objects or other contexts can be bound. The square shapes represent objects.

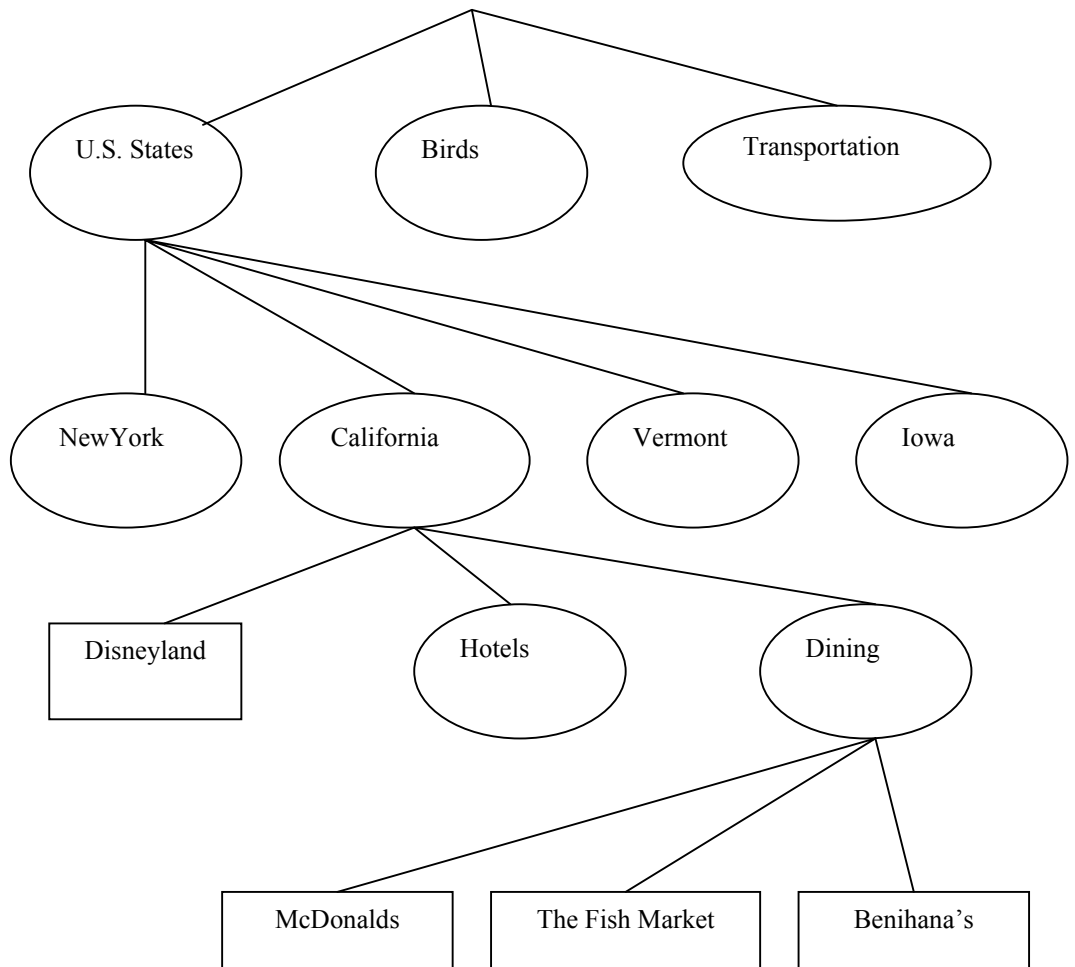


FIGURE 7. *A naming service is a tree-like data structure.*

COS Naming

COS stands for CORBA Object Services, a general term used to describe a service provided by the CORBA specification. The naming service is just such a service, with a standard set of APIs found in the `org.omg.CosNaming` package.

The `org.omg.CosNaming` package contains an interface called `NamingContext`, which represents the API that allows you to communicate with an actual naming service. To obtain a reference to a naming service, you use the `resolve_initial_references()` method of the ORB object:

```
org.omg.CORBA.Object rootObj =  
    orb.resolve_initial_references("NameService");  
NamingContext root = NamingContextHelper.narrow(rootObj);
```

Notice in the statement above that the ORB is locating the "NameService" it provides. The `narrow()` method of the `NamingContextHelper` class is used to narrow the reference to a `NamingContext` type.

This reference represents the root context of the naming service tree.

The NamingContext Interface

Now that you have seen how to obtain a reference to the root context of the naming service provided by an ORB, you will want to know what to do with that reference. You can bind a context or object to any location on the naming tree. The `NameComponent` class is a simple class that encapsulates a context or object name. The `NameComponent` class consists of two strings: an id and a kind. The id is the name of the context or object and kind is an optional value which can be used to explain what kind of object or context it is.

The following two statements instantiate two `NameComponent` objects:

```
NameComponent california = new NameComponent("California", "State");
NameComponent mcd = new NameComponent("McDonalds", "Fast Food");
```

Keep in mind that the kind is optional and that most clients will rely on the id to locate an object or context in the naming service.

The following is a description of some of the methods in the `NamingContext` interface:

```
public void bind(NameComponent [] n, Object obj)
    throws NotFound, CannotProceed, InvalidName, AlreadyBound
    binds the given object in the naming service at the context and names specified
    in the array of NameComponent objects. If an object at the same context with
    the same name is found, then an AlreadyBound exception occurs.
```

```
public void rebind(NameComponent [] n, Object obj)
    throws NotFound, CannotProceed, InvalidName
    the same behavior as bind, except an AlreadyBound exception will not occur.
    The newer object will simply replace any existing object with the same name.
```

```
public NamingContext bind_new_context(NameComponent [] n)
    throws NotFound, CannotProceed, InvalidName, AlreadyBound
    binds a new naming context of the specified name to this context.
public Object resolve(NameComponent [] n)
    throws NotFound, CannotProceed, InvalidName
    locates an object in this naming context that has the given name.
```

Using the Naming Service

The following server application creates a `BasketballImpl` object and stores it in the naming service.

```
1 import org.omg.PortableServer.*;
2 import org.omg.CosNaming.*;
3
4 public class BasketballServer
5 {
6     public static void main(String [] args)
7     {
8         try
9         {
10             System.out.println("Initializing the ORB");
11             org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
12
13             System.out.println("Locating the root POA");
14             org.omg.CORBA.Object objRef =
15                 orb.resolve_initial_references("RootPOA");
16             POA rootPOA = POAHelper.narrow(objRef);
17
18             System.out.println("Locating the NameService");
19             org.omg.CORBA.Object nameRef =
20                 orb.resolve_initial_references("NameService");
21             NamingContext rootName =
22                 NamingContextHelper.narrow(nameRef);
23
24             System.out.println("Creating the POA");
25             org.omg.CORBA.Policy[] policies = {
26                 rootPOA.create_lifespan_policy(
27                     LifespanPolicyValue.PERSISTENT)
28             };
29
30             POA sportsPOA = rootPOA.create_POA("sports_poa",
31                 rootPOA.the_POAManager(), policies);
32
33             System.out.println("Activating the POA manager");
34             rootPOA.the_POAManager().activate();
35
36             BasketballImpl bball_servant = new BasketballImpl();
37             sportsPOA.activate_object_with_id("Basketball".getBytes(),
38                 bball_servant);
39
40             System.out.println("Binding the servant in name service");
41
```

```
42     NameComponent [] node = {
43         new NameComponent("Sports", "")
44     };
45
46     NamingContext sports = rootName.bind_new_context(node);
47
48     NameComponent [] leaf = {
49         new NameComponent("Basketball", "")
50     };
51
52     sports.rebind(leaf,
53         sportsPOA.servant_to_reference(bball_servant));
54
55     System.out.println("Running the orb...");
56     orb.run();
57 } catch (Exception e)
58 {
59     e.printStackTrace();
60 }
61 }
62 }
```

The following class demonstrates a client using the naming service to locate a servant reference.

```
1 import org.omg.CosNaming.*;
2 import sports.*;
3
4 public class BasketballClient
5 {
6     public static void main(String [] args)
7     {
8         try
9         {
10             System.out.println("Initializing the ORB");
11             org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
12
13             org.omg.CORBA.Object objRef =
14                 orb.resolve_initial_references("NameService");
15             NamingContext rootName =
16                 NamingContextHelper.narrow(objRef);
17
18             NameComponent [] names = {
19                 new NameComponent("Sports", ""),
20                 new NameComponent("Basketball", "")
21             };
22
23             org.omg.CORBA.Object ref = rootName.resolve(names);
24             Basketball bball = BasketballHelper.narrow(ref);
25
26             System.out.println("Initial score = " + bball.score());
27             byte pts = 2;
28             bball.addToScore(pts);
29             pts = 1;
30             bball.addToScore(pts);
31             System.out.println("Score is now = " + bball.score());
32
33             pts = 10;
34             bball.addToScore(pts);
35             System.out.println("This should not be displayed");
36         } catch (Exception e)
37         {
38             e.printStackTrace();
39         }
40     }
41 }
```

Running the Name Service

The network that your CORBA applications are running on may have multiple naming services available. You can specify which naming service you want to use using the `SVCnameroot` property.

For example, the following command line runs the `BasketballServer` program using the default naming service, which is specified by `"NameService"`.

```
vbj -DSVCnameroot=NameService BasketballServer
```

You can also use the `ORBInitRef` property to specify the location of the naming service. The following command line demonstrates running the `BasketballClient` program using a remote naming service:

```
vbj -DORBInitRef=NameService=corbaloc://208.141.16.192:20002/  
NameService BasketballClient
```

Lab 5: Using the Naming Service

Objective: To become familiar using the naming service of an ORB.

Perform the following tasks:

1. You will begin by modifying the BankAccountServer class from the previous lab. Open the source code in your editor.
2. Add a call to `resolve_initial_references()` to obtain a reference to the NameService. Narrow the return value accordingly.
3. After you instantiate your servant, activate the servant using the POA method `activate_object_with_id()`.
4. Bind a new context called "Banks" to the naming service's root context.
5. Bind your servant to the "Banks" context, giving it the name "MyAccount". Be sure to invoke `servant_to_reference()` to convert the servant reference to an IOR.
6. Save and compile the BankAccountServer class.
7. Modify the BankAccountClient class. Make the necessary changes so that the client looks up the reference in the name service instead of reading the IOR from a file.
8. Save and compile the BankAccountClient class.
9. Before running the server, make sure the naming service application is running on your machine.
10. Run the server and client and verify that it is working successfully.

What you should see: The output will look the same as before. The only difference is that in this lab, the reference was obtained through a naming service instead of an IOR read from a file.

Lab 6: Returning a servant reference from a method

Objective: You have seen how to obtain a reference to a servant using an IOR and also using a naming service. In this lab, you will learn how a client can obtain a reference to a servant via the return value of a method.

Perform the following tasks:

1. Create a class called `AccountManagerImpl`. It is going to be the implementation for the `AccountManager` interface, so have it extend the appropriate base class.
2. You will now implement the `openAccount()` method. Note that `openAccount()` returns a `BankAccount`, which is a CORBA object.
3. Within `openAccount()`, instantiate a new `BankAccountImpl` object. Assign the name of this new account to be the name passed in to `openAccount()`.
4. You are going to activate the `BankAccountImpl` object with the POA by using the `servant_to_reference()` method. Add the following statement to `openAccount()`:

```
_default_POA().servant_to_reference(accountImpl);
```

Hang on to the return value, which is a reference of type `org.omg.CORBA.Object`.

5. Use the `narrow()` method of `BankAccountHelper` to narrow the servant reference to type `BankAccount` so that it can be returned from this method. Return the narrowed reference.
6. Save and compile the `AccountManagerImpl` class.

7. Write a class called `AccountManagerServer` and declare the `main()` method within this class.
8. Initialize the ORB.
9. Obtain a reference to the root POA and use it to create a new POA. Activate the root POA manager.
10. Create an instance of the `AccountManagerImpl` class.
11. Using either the naming service or an IOR saved to a file, make the servant reference accessible to a client.
12. Run the ORB.
13. Save and compile the `AccountManagerServer` class.
14. Write a class called `AccountManagerClient`. Declare `main()` within this class.
15. Locate the reference to the `AccountManager` servant and narrow it accordingly.
16. Use the `openAccount()` method of the `AccountManager` servant to create a `BankAccount` object.
17. Invoke the methods of the `BankAccount` object to verify that everything is working.
18. Save and compile the `AccountManagerClient` class.
19. Run the server and client and verify that everything is working.

What you should see: The `AccountManager` should successfully obtain a `BankAccount` object for you.



An Introduction to XML

This chapter discusses the details of XML and XML documents. XML plays a key role in all Java development, and it is necessary to understand the various components of an XML document and their corresponding Document Type Definition (DTD).

Topics covered include:

- XML Tags
- Attributes
- When to Use Attributes
- Entity Elements
- Well-formed vs. Valid
- Document Type Definition
- Creating a DTD
- XML Namespaces

XML Tags

XML documents consist of *tags* that describe a particular piece of data. The tags themselves describe the data, and the text that appears between the tags is the particular content. The term tag is used interchangeably with the term *element*.

For example, the following tag is used to denote the age of something, with this particular age being 21:

```
<age>21</age>
```

Notice the beginning `<age>` tag has a matching ending `</age>` tag using a “/” to denote the ending tag. A beginning XML tag must have a corresponding ending tag, unless it is an empty element.

EMPTY ELEMENTS

An *empty element* does not contain any data and is useful when just the existence of the tag says something. For example, an XML document describing an email message might use a `<replied>` element to specify that the email message has been replied to. Since this element is basically a flag, there is no content and could be used as:

```
<replied/>
```

This is basically a shortcut way of writing:

```
<replied></replied>
```

The Root Element

The *root element* of an XML document is the element at the highest-level. It encompasses all of the other elements in the document, meaning that all the other tags are nested within the root element.

An XML document must contain a single instance of the root element. The root element is the *type* of document being described. For example, the information involved in an airline flight could be described in a <flight> document, where the root element is <flight>.

```
<flight>
  <number>344</number>
  <airline>United Airlines</airline>
  <departure>
    <city>Los Angeles</city>
    <state>CA</state>
    <airport>LAX</airport>
  </departure>
  <destination>
    <city>Orlando</city>
    <state>FL</state>
    <airport>MCO</airport>
  </destination>
  <departure-time>1130</departure-time>
  <arrival-time>1845</arrival-time>
  <meal/>
  <movie/>
</flight>
```

FIGURE 1. A <flight> document.

- The XML document in Figure 1 describes an airline flight, and <flight> is the *type* of the document. Notice that the other tags are nested within <flight>. Also notice how straightforward it is to read XML, since the tags are often self-describing. It appears that the flight in Figure 1 goes for L.A. to Orlando at 11:30 a.m. and offers a meal and an in-flight movie.

Attributes

A tag can contain an *attribute*, which consists of additional information relevant to the tag. An attribute is a name=value pair that appears within the beginning tag, as opposed to content, which appears between the beginning and ending tags.

For example, Figure 1 on page 313 contains a <flight> document with a <number> element nested within it. The flight number could also be denoted by an attribute, as follows:

```
<flight number="344">
```

```
</flight>
```

USE DOUBLE QUOTES FOR ATTRIBUTES

Notice the number attribute of the flight appears within the <flight> tag and its value is enclosed in double quotes. Attributes should always appear within double quotes.

A common use of attributes is when the choice of the value comes from a small list. For example, the choice of <meal> might be breakfast, lunch, snack or supper and could be denoted using an attribute:

```
<meal type="snack"/>
```

Perhaps the <airline> tag might use an attribute to denote a partner airline for flights that are provided by a smaller regional airline. For example, United Airlines has many of regional flights handled by United Express:

```
<airline partner="United Airlines">  
    United Express  
</airline>
```

When to Use Attributes

A common design concern with XML is whether a particular bit of information should be contained within an element or used as an attribute. In many situations, it is a clear decision between an element or an attribute:

- If the data changes frequently, then it is best suited as an element.
- If the data can only take on a small number of values, then an attribute might be better suited because any invalid values can be determined during the validation process (discussed next).
- If the data is large or covers more than one line, then an element must be used.
- If the data could possibly contain subelements, then an element must be used.
- If the data is a small string that does not change often, then an attribute might be a good choice.

If the data changes frequently, then it would probably be best to not use the “number” attribute for the <flight> tag, since there are hundreds of possible values. However, using a “type” attribute for the <meal> tag seems like a good design.

NOTE

In some situations, an element is required, while in other situations it is a design decision based on whatever style you prefer. Attributes are widely-used in XML documents, and since using attributes is style decision, you can expect to see them used in many different ways.

Entity Elements

XML provides the **<!ENTITY>** tag, an essential feature of XML, for allowing the definition of entities. An *entity* is either a selection of text or a separate file, where the text or file can be thought of a variable to be used later in the XML document. You define an entity at the beginning of the XML document or in the DTD file.

For example, the following document has an entity called "author" and also entities to represent other files.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE Novel SYSTEM "MyTag.dtd" [
3      <!ENTITY author "Mark Twain">
4      <!ENTITY lt"<">
5      <!ENTITY gt ">">
6      <!ENTITY intro SYSTEM "introduction.xml">
7      <!ENTITY toc SYSTEM "tableofcontents.xml">
8  ]>
9
10 <Novel>
11     <Title>Tom Sawyer</Title>
12     <Author> &author; </Author>
13     &intro;
14     &toc;
15     <Farewell>
16         We hope you enjoyed this novel by &author;
17     </Farewell>
18 </Novel>
```

FIGURE 2. *An XML document with ENTITY elements.*

NOTE

An ampersand & and a semi-colon are how to use an entity in an XML document. Also notice that entities are necessary for characters that have a specific meaning in an XML document, like the less-than and greater-than characters.

Notes

Well-formed vs. Valid

The format of an XML document must be well-formed. In addition, a validation can occur where the tags are verified for proper use. These are two different but important concepts in XML.

Well-formed: An XML document is said to be well-formed if the grammar of the tags is correct. For example, every beginning tag needs an ending tag, tags must be nested, and so on. The parser enforces the grammar of XML when parsing the tags. For example, the following tags are not well-formed because they are not nested properly:

```
<tagA>
  <tagB>
</tagA>
  </tagB>
```

The following tags are not well-formed because not all tags are not ended properly:

```
<html>
  <body>
    <p>Hello
  <br>
</html>
```

- **Valid:** A document may be well-formed, but the parser does not check for validity. For example, you might have followed all the rules of grammar so that your document is well-formed, but what you created might have been a misuse of the tags.

For example, the following tags are well-formed but are probably not valid, since they are not used in the context of how they were intended:

```
<airline>
  <city>Los Angeles</city>
  <number/>
  <meal>
    <departure-time>red</departure-time>
  </meal>
</airline>
```

As you can see, an XML document can be well-formed, but the tags may be used improperly. This type of XML document is called invalid. An XML document is *valid* if all of the tags used have been nested properly, appear the appropriate number of times, and contain the correct type of content.

A separate document, called a *Document Type Definition* (DTD), is created to define all of the valid uses of the tags. A Document Type Definition is exactly what its name implies, a “definition” for a “type” of XML “document”, where the type of document is the name of the root element.

Document Type Definition

A Document Type Definition (DTD) is a text file that is used to validate an XML document. It allows you to determine the following about an XML document:

- The valid elements of a document.
- Which elements can be nested within an element.
- The number of elements that can appear within an element.
- The types of attributes an element can have.
- The specific values that an attribute can have.
- The default value of an attribute.

You associate an XML document with a DTD using either:

- SYSTEM - which represents a DTD that is generally not available to the public.
- PUBLIC - which represents a DTD that is well-known and available to the public.

Declaring a SYSTEM DTD

Use the SYSTEM keyword to declare an element is using a system DTD. For example, the following XML statement demonstrates a SYSTEM DTD named `flight.dtd`:

```
<!DOCTYPE Flight SYSTEM "flight.dtd">
```

Declaring a PUBLIC DTD

If the `<flight>` element was used world-wide by all airlines, then the DTD would most likely be publicly available:

```
<!DOCTYPE Flight PUBLIC "ISO//Airlines Assoc//Flight Data//EN" "http://  
www.airlines.org/dtds/flight.dtd">
```

Notice there are two items that follow PUBLIC:

- the name of the DTD,
- the URL where it is located.

The name of the DTD is broken down into sections separated by two forward slashes (`//`). ISO is used for approved ISO-standard DTDs (use a `“+”` for approved non-ISO standard DTDs and a `“-”` for non-approved non-ISO standard DTDs). `“Flight Data”` is the name of the DTD. `“Airlines Assoc”` denotes the owner of the DTD, and `“EN”` denotes the language of the DTD, which is English in this case.

Creating a DTD

The following document is an example of a DTD for the <flight> element defined earlier in this module.

```
1  <!ELEMENT flight (number, airline+, departure, destination,  
departure-time, arrival-time, meal*, movie?)>  
2  
3  <!ELEMENT number (#PCDATA)>  
4  <!ELEMENT airline (#PCDATA)>  
5  
6  <!ELEMENT departure (city, state, airport)>  
7  <!ELEMENT destination (city, state, airport)>  
8  <!ELEMENT city (#PCDATA)>  
9  <!ELEMENT state (#PCDATA)>  
10 <!ELEMENT airport (#PCDATA)>  
11  
12 <!ELEMENT departure-time (#PCDATA)>  
13 <!ELEMENT arrival-time (#PCDATA)>  
14  
15 <!ELEMENT meal EMPTY>  
16 <!ATTLIST meal type (breakfast|lunch|snack|supper) #REQUIRED>  
17  
18 <!ELEMENT movie EMPTY>  
19 <!ATTLIST movie title CDATA #IMPLIED>
```

FIGURE 3. *The <flight> Document Type Definition.*

- The ELEMENT keyword is used to define an element. PCDATA stands for Parsed Character Data and is used for elements that consist of text data. For example, the <number> and <airline> elements need to contain data.
- The parentheses are used to denote the nesting of elements. For example, the <flight> tag can have nested within it number, airline, departure, destination, departure-time, arrival-time, meal and movie. The order the elements are listed is the order that they must appear within the <flight> element.

The DTD in Figure 3 demonstrates the recurrence operators, which allow you to specify a minimum or maximum number of times a tag can appear. The possible values are:

- [Default] Tag must appear exactly once
- ? Tag can appear zero or one time only.
- + Tag must appear at least once
- * Tag may appear any number of times, including not at all

In the DTD in Figure 3, <number> must appear exactly once within <flight>, while <airline> can appear one or more times and <meal> can appear any number of times, including or zero.

The EMPTY keyword is used to denote an element that does not contain any nested tags or data. The <meal> and <movie> tags are both defined as EMPTY.

The ATTLIST keyword is used to define the attributes that an element can have. For example, the <meal> element must have the “type” attribute because it is defined as #REQUIRED, and it can only be one of the values listed: breakfast, lunch, snack or supper. A default value can also be supplied, as in the following example where the default value is assigned to be “snack”:

```
<!ATTLIST meal type (breakfast|lunch|snack|supper) "snack">
```

The <movie> element can contain an optional “title” attribute because it is denoted as #IMPLIED. The title can be CDATA, meaning any character data.

The <web-app> DTD

The Web application deployment descriptor for a Java Web application is an XML document that must adhere to the following DTD:

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

The <web-app> DTD is fairly long, but you can get a feel for what a <web-app> looks like by studying the child elements allowed under the <web-app> root:

```
<!ELEMENT web-app (icon?, display-name?, description?,
distributable?, context-param*, filter*, filter-mapping*, listener*,
servlet*, servlet-mapping*, session-config?, mime-mapping*,
welcome-file-list?, error-page*, taglib*, resource-env-ref*,
resource-ref*, security-constraint*, login-config?, security-role*,
env-entry*, ejb-ref*, ejb-local-ref*)>
```

NOTE

Notice that none of the child elements of <web-app> are required. Each one is either "?" or "*".

Figure 4 shows an example of a valid, well-formed <web-app> document.

```
1  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN"
2      "http://java.sun.com/dtd/web-app_2_3.dtd">
3
4  <web-app>
5      <display-name>Demo Web Application</display-name>
6
7      <filter>
8          <filter-name>timer</filter-name>
9          <filter-class>filters.TimerFilter</filter-class>
10         </filter>
11
12         <filter-mapping>
13             <filter-name>timer</filter-name>
14             <url-pattern>/*.jsp</url-pattern>
15         </filter-mapping>
16
17         <filter-mapping>
18             <filter-name>timer</filter-name>
19             <url-pattern>/*.htm*</url-pattern>
20         </filter-mapping>
21
22         <listener>
23             <listener-class>listeners.MySessionCounter</listener-class>
24         </listener>
25
26         <servlet>
27             <servlet-name>longtime</servlet-name>
28             <servlet-class>LongRunningDemo</servlet-class>
29         </servlet>
30
31         <servlet-mapping>
32             <servlet-name>longtime</servlet-name>
33             <url-pattern>/longtime</url-pattern>
34         </servlet-mapping>
35
36         <session-config>
37             <session-timeout>30</session-timeout>
38         </session-config>
39     </web-app>
40
```

FIGURE 4. *A sample <web-app> document.*

XML Namespaces

XML uses the concept of a namespace, where elements are defined not just by their name, but also by a prefix. The purpose for using namespaces is two-fold:

- to provide a way to avoid naming conflicts,
- and to allow for a group of elements that are associated with each other to be easily recognizable.

A naming conflict might arise when you have an XML document that contains two elements with the same name but different meanings. For example, the following XML document contains two `<title>` elements, one to represent the title of a book, and other to represent the title of a music CD. Suppose the `<book>` element is a standard type and cannot be changed. Similarly, suppose the `<cd>` element is also a standard type that is used in many situations, including `<order>` documents.

```
1  <?xml version="1.0"?>
2
3  <order>
4      <book>
5          <title>The Adventures of Tom Sawyer</title>
6          <author>Mark Twain</author>
7      </book>
8
9      <cd>
10         <title>The White Album</title>
11         <artist>The Beatles</artist>
12     </cd>
13 </order>
```

FIGURE 5. *An XML document that contains a possible naming conflict.*

- The problem arises when the document is being parsed and a `<title>` element is reached. Is it the title of a book, or the title of a CD? Using namespaces provides a solution for this problem.

If the <book> element is actually an industry standard, then it will most likely be defined within a namespace, along with all of its child elements. The namespace is prefixed to each element in the document. If the namespace is unique, then no naming conflict will occur with <book> or any of its child elements.

To ensure uniqueness, namespaces are chosen using a company or organization's URI. For example, suppose the <book> document is defined within the "http://www.read.org/books/" namespace. Since prefixing such a long namespace to each element would be tedious (not to mention invalid, since characters like a forward slash are not valid XML), we use the xmlns keyword to assign a simpler prefix "abc" equal to the namespace, as shown in Figure 6.

```
1  <?xml version="1.0"?>
2
3  <order>
4      <abc:book xmlns:abc="http://www.read.org/books/">
5          <abc:title>The Adventures of Tom Sawyer</abc:title>
6          <abc:author>Mark Twain</abc:author>
7      </abc:book>
8
9      <music:cd xmlns:music="http://www.ascap.org/en/media/">
10         <music:title>The White Album</music:title>
11         <music:artist>The Beatles</music:artist>
12     </music:cd>
13 </order>
```

FIGURE 6. *An XML document that uses namespaces.*

NOTE

Notice that in Figure 6, the <cd> element also appears within a namespace. Now in the <order> document there are no naming conflicts. The titles of books use a <abc:title> element and the titles of CDs use a <music:title> element.

Lab 1: Designing an XML Document

Objective: The purpose of this lab is to become familiar with the process of defining an XML document and a DTD.

Perform the following tasks:

1. When a customer completes a purchase on the BEST Web site, the information about their order is sent to different departments which use different systems and applications to process the order. For example, the shipping department needs to receive a copy of the order for shipping, and the billing department needs to process the customer's credit card. To allow for your BEST Web site to communicate with these existing systems, you are going to create an XML document that represents a customer's order.
2. Define an XML document of type `<order>` that contains the information of a customer's order. You need to determine the elements of an `<order>` and their usage. You might find it helpful to create a sample `<order>` document.
3. Write a DTD for your `<order>` document.

NOTE

There is no right or wrong answer to this lab. You will most likely find that in a real-world situation, it is worth investing time and resources into designing an appropriate DTD, since changing a DTD frequently becomes a maintenance issue.



The Java API for XML Processing

This chapter discusses the Java API for XML Processing, referred to as JAXP. XML is the foundation for Web services, and it is necessary to understand how to create new XML documents and parse existing ones, both tasks accomplished using the APIs of the JAXP.

Topics covered include:

- JAXP - The Java API for XML Processing
- SAX - The Simple API for XML Parsing
- DOM - The Document Object Model
- XSLT - The XML Stylesheet Language Transformations

The Java API for XML Processing

A common need in XML application development is for an XML document to be *parsed*. Parsing an XML document involves traversing the document element by element, reading each tag and its subsequent contents or subelements. Typically, you will have no need to develop a parser yourself. There are many readily-available XML-compliant parsers for use with Java or any other popular programming language.

Another common need in XML application development is for an XML document to be created. In many situations, the data that represents the XML document is found in a database or is being input dynamically from a user or series of events. You need to take this information and generate the necessary tags, attributes, and content so that the data will be represented as XML.

How do you write a Java program that takes advantage of a parser? How do you write a Java program that generates an XML document? This is where the JAXP comes in to play. JAXP stands for Java API for XML Processing and is a collection of APIs that allow Java applications to parse, create, view or translate XML documents.

JAXP consists of the following APIs:

- SAX: The Simple API for XML
- DOM: The Document Object Model
- XSLT: The XML Stylesheet Language for Transformation

Information about JAXP, including a reference implementation and documentation, can be found on Sun's Web site at <http://java.sun.com/xml>.

NOTE

JAXP is a part of the Standard Edition of Java (J2SE).

The APIs of the JAXP are defined in the following Java packages:

- `org.xml.sax` - contains the classes and interfaces that represent the basic SAX APIs.
- `org.w3c.dom` - contains the classes and interfaces needed for using DOM
- `javax.xml.transform` - contains the XSLT APIs
- `javax.xml.parsers` - contains classes and interfaces for use with both SAX and DOM parsers.

There are additional packages in the JAXP which comprise the various components of SAX, DOM and XSLT which will be discussed throughout this chapter.

Overview of SAX

The Simple API for XML (SAX) allows you to develop Java applications that parse and process an XML document. The SAX is implemented by a parser vendor, and your Java applications use SAX to generate an instance of the parser, then have the parser process an XML document.

The SAX consists of the following packages:

- `org.xml.sax`: Contains the foundation of the SAX. Commonly used interfaces include `XMLReader`, for parsing an XML document, and `ContentHandler`, for determining the content of an XML document.
- `org.xml.sax.helpers`: Contains various classes and interfaces for locating a parser or creating a parser. For example, you can use the `XMLReaderFactory` to locate and initialize your parsing tool.
- `org.xml.sax.ext`: Contains two interfaces that are optional to the SAX.

Parsing an XML document using SAX involves the parser starting with the first element and serially processing the document element-by-element, firing events as it reads elements, attributes, and content.

NOTE

A SAX parser traverses the XML document once and simply generates events as the document is parsed. At no point does the SAX parser know where it is in the document relative to the nesting of elements or content. For example, when the parser reads an `<number>` tag, it does not realize that it has just previously read a `<flight>` tag and that `<number>` is nested within `<flight>`. The SAX parser just generates an event that a `<flight>` tag is read, then another event for the `<number>` tag.

- When using SAX, you write handler classes that listen for the events generated by the parser. It is up to your handler classes to be aware of which tags are nested within other tags, which content goes with which tags, and so on.
- SAX is useful for extracting the data from an XML document in an efficient and low-memory manner (as opposed to DOM, which reads an entire XML document into memory).

Notes

Overview of DOM

DOM stands for Document Object Model, and represents a method for describing an XML document as a tree data structure. The DOM API contains classes and interfaces for both parsing XML documents and also for creating XML documents.

NOTE

When an XML document is parsed using DOM, the entire document is read into memory and stored as a tree-like structure. Your Java application can then traverse the tree structure to determine the elements and content of the XML document.

Similarly, if you want to create an XML document using DOM, you build the tree structure in memory, which can then be output to a file or database in XML format.

DOM consists of the following package:

- `org.w3c.dom` - contains the Document interface to represent an XML document, the Node interface to represent the various components of the tree structure, and other interfaces that define DOM as specified by the W3C.

Other classes that are commonly used with DOM are found in the `javax.xml.parsers` package, which contains the `DocumentBuilderFactory` and `DocumentBuilder` classes for accessing DOM implementations.

Notes

Creating a SAX Parser

The first step in writing a Java application that processes an XML document is to obtain a reference to the actual parser. This parser will be a Java class that you will instantiate in your Java code.

The class `SAXParserFactory` in the `javax.xml.parsers` package is used to create a reference to a `SAXParser`, which will represent your parser. To obtain a `SAXParserFactory` object, you invoke the following static method in the `SAXParserFactory` class :

```
public static SAXParserFactory newInstance()
```

This method will search for a SAX parser factory in the following order:

1. Use the parser factory defined with the `javax.xml.parsers.SAXParserFactory` system property.
2. Use the `javax.xml.parsers.SAXParserFactory` system property defined in the `jaxp.properties` file, found in the `<jre-directory>/lib` folder.
3. Use the `META-INF/services/javax.xml.parsers.SAXParserFactory` property if using JAR files and the Services API.
4. If the parser factory property is not found in any of the above places, then the `newInstance()` method will use the default parser factory of the current platform.

Once you have the factory object, invoke its `newSAXParser()` method to create a new parser instance:

```
public SAXParser newSAXParser()
```


Before you obtain a new parser, there are several methods in the SAXParserFactory class available for configuring the parser:

- `public void setValidating(boolean b)`: By default, validation is set to false. Invoke this method if you want the parser to validate while parsing.
- `public void setNamespaceAware(boolean b)`: Use this method to denote if the parser being used provides support for XML namespaces. This value is false by default.
- `public void setFeature(String featureName, boolean value)`: This method is used to set specific features of the parser.

Similarly, in the SAXParser interface there is a method for setting properties of the parser after you have instantiated a new parser. The method is:

```
public void setProperty(String name, Object value)
```

According to the JAXP API documentation, a list of features and properties for SAX parsers can be found at <http://www.megginson.com/SAX/Java/features.html>.

A Simple Parser

The following program demonstrates creating a SAX parser with validation and namespace awareness turned on.

```
1  import org.xml.sax.helpers.DefaultHandler;
2  import javax.xml.parsers.SAXParserFactory;
3  import javax.xml.parsers.SAXParser;
4  import java.io.File;
5
6  public class SimpleParser
7  {
8      public static void main(String [] args)
9      {
10         try
11         {
12             //Use the default SAX parser factory
13             SAXParserFactory factory =
14                 SAXParserFactory.newInstance();
15
16             factory.setValidating(true);
17             factory.setNamespaceAware(true);
18
19             SAXParser parser = factory.newSAXParser();
20         }catch(Exception e)
21         {
22             e.printStackTrace();
23         }
24     }
25 }
```

FIGURE 1. *Creating a SAX parser that validates while parsing and is namespace aware.*

NOTE

The `org.xml.sax.helpers` and `javax.xml.parsers` packages used in the SimpleParser example in Figure 1 are all found in the J2SE, so all you need to compile and run this program is the J2SE SDK.

Notes

Parsing an XML Document Using SAX

Once you have configured the parser and obtained a reference to it, you will have an object of type `SAXParser`. The `SAXParser` class contains several methods for parsing an XML document, depending on the input source of the document.

- `public void parse(java.io.File source, DefaultHandler dh)`
- `public void parse(org.xml.sax.InputSource source, DefaultHandler dh)`
- `public void parse(java.io.InputStream source, DefaultHandler dh)`
- `public void parse(String uri, DefaultHandler dh)`

Each `parse()` method defined in the `SAXParser` class takes in as its first argument the source of the XML document to be parsed. The XML document can be a `File`, an `InputStream`, an `InputSource`, or a `String` representing the URI of the document.

Notice the second argument of each `parse()` method is of type `DefaultHandler`. This class represents the listeners that will be waiting for events to be fired from the SAX parser. The `DefaultHandler` will be discussed in the upcoming section “Handlers” on page 344.

The following Java program demonstrates using a SAX parser to parse an XML document. In this program, the XML document is a file, and the name of the file to be parsed is passed in as the first command line argument, `args[0]`.

```
1  import org.xml.sax.helpers.DefaultHandler;
2  import javax.xml.parsers.SAXParserFactory;
3  import javax.xml.parsers.SAXParser;
4  import java.io.File;
5
6  public class SimpleParser
7  {
8      public static void main(String [] args)
9      {
10         try
11         {
12             //Use the default SAX parser factory
13             SAXParserFactory factory =
14                 SAXParserFactory.newInstance();
15
16             factory.setValidating(true);
17             factory.setNamespaceAware(true);
18
19             SAXParser parser = factory.newSAXParser();
20
21             //parse the filename passed in on the command line
22             //the DefaultHandler will do nothing
23             parser.parse(new File(args[0]),
24                 new DefaultHandler());
25
26         } catch (Exception e)
27         {
28             e.printStackTrace();
29         }
30     }
31 }
```

FIGURE 2. *A simple SAX parsing program.*

NOTE

Executing the SimpleParser program does not display any output or create any files, unless the document is not well-formed or valid, in which case an exception will occur when the `parse()` method is invoked. The program above simply parses the XML document passed in to the `parse` method. We will find out next how to monitor the parsing process using content and error handlers.

Handlers

When an XML document is parsed, it generates a series of events. When the `parse()` method is invoked, you pass in the document to be parsed and also the handler class that is to be notified of these events. The events are separated into four categories, depending on what has just occurred during parsing.

There is an interface in the `org.xml.sax` package for each of the four different types of events that can occur. To listen to the events, you need to write a class that implements the methods in these interfaces. The four types interfaces are:

- **ContentHandler** - contains methods for listening to such events as the beginning and ending of the XML document, the beginning and ending of each tag in the document, and when parsed character data is being read.
- **ErrorHandler** - contains methods for listening to errors, fatal errors, and warnings.
- **DTDHandler** - contains methods for listening to DTD-related notation and unparsed entity declaration events.
- **EntityResolver** - contains a single method that is invoked by the parser whenever an external entity is parsed.

THE DEFAULTHANDLER CLASS

The simplest way to write a handler in Java is to write a new class that extends the `org.xml.sax.helpers.DefaultHandler` class. The `DefaultHandler` class implements all four of the above interfaces, defining each of the required methods with an empty method body. The new subclass that you write can override those methods in the four interfaces that your XML application is interested in.

We will now discuss the `ContentHandler` interface in detail.

Content Handlers

The `org.xml.sax.ContentHandler` interface contains methods that are invoked by the parser when the actual elements and data in the XML document are being parsed.

When using SAX to parse a document, the content handler needs to keep track of which element is currently being processed, what the contents of the element are, and when the element is finished being processed. The SAX parser simply parses the XML document and invokes one of the methods in the `ContentHandler` interface, depending on what the parser just processed.

The following class called `SimpleContentHandler` demonstrates a content handler for an XML document. It defines all eleven of the methods in the `ContentHandler` interface for demonstration purposes, to illustrate when the methods are invoked.

```
1  import org.xml.sax.*;
2  import org.xml.sax.helpers.DefaultHandler;
3
4  public class SimpleContentHandler extends DefaultHandler
5  {
6      private Locator locator;
7
8      public void setDocumentLocator(Locator locator)
9      {
10         System.out.println("Setting the locator");
11         this.locator = locator;
12     }
13
14     public void startDocument()
15     {
16         System.out.println("Starting document...");
17     }
18
19     public void endDocument()
20     {
21         System.out.println("...Ending document");
22     }
23
24     public void startPrefixMapping(String prefix, String uri)
25     {
26         System.out.println("Starting namespace " + uri
27 + " of " + prefix);
28     }
```

```
29
30     public void endPrefixMapping(String prefix)
31     {
32         System.out.println("Ending namespace for " + prefix);
33     }
34
35     public void startElement(String uri, String name,
36                             String rawname, Attributes atts)
37     {
38         System.out.println("Starting element <" + rawname +
39 ">");
39     }
40
41     public void endElement(String uri, String name,
42                           String rawname)
43     {
44         System.out.println("Ending element <" + rawname + ">");
45     }
46
47     public void characters(char [] ch, int start, int end)
48     {
49         String s = new String(ch, start, end);
50         System.out.println("Characters: " + s);
51     }
52
53     public void ignorableWhitespace(char [] ch, int start, int
54 end)
55     {
56         System.out.println("Whitespace");
57     }
58
59     public void processingInstruction(String target,
60                                     String data)
61     {
62         System.out.println("Processing " + target
63 + " with " + data);
63     }
64
65     public void skippedEntity(String name)
66     {
67         System.out.println("Skipping " + name);
68     }
69 }
```

FIGURE 3. *An example of a SAX content handler.*

NOTE

Since SimpleContentHandler extends DefaultHandler in Figure 3, and DefaultHandler implements all eleven of the methods in the ContentHandler interface, the SimpleContentHandler did not need to include all eleven of these methods. When you extend DefaultHandler, you can include as many or as few of the ContentHandler methods that you need.

Now that we have a ContentHandler, we need to tell the parser so it knows which handler to fire the events to while parsing. This is done when we invoke the parse() method, as shown in the following program.

```
1  import org.xml.sax.helpers.DefaultHandler;
2  import javax.xml.parsers.SAXParserFactory;
3  import javax.xml.parsers.SAXParser;
4  import java.io.File;
5
6  public class SimpleParserWithHandler
7  {
8      public static void main(String [] args)
9      {
10         try
11         {
12             //Use the default SAX parser factory
13             SAXParserFactory factory =
14                 SAXParserFactory.newInstance();
15
16             factory.setValidating(true);
17             factory.setNamespaceAware(true);
18
19             SAXParser parser = factory.newSAXParser();
20
21             //parse the filename passed in on the command line
22             //using a SimpleContentHandler to handle events
23             parser.parse(new File(args[0]),
24                 new SimpleContentHandler());
25
26         } catch (Exception e)
27         {
28             e.printStackTrace();
29         }
30     }
31 }
```

FIGURE 4. Using a SAX parser with a content handler.

The following shows the output of the SimpleParserWithHandler program in Figure 4 when parsing the flight.xml document defined in Figure 1 on page 313.

```
1  Setting the locator
2  Starting document...
3  Starting element <flight>
4  Characters:
5
6  Starting element <number>
7  Characters: 344
8  Ending element <number>
9  Characters:
10
11 Starting element <airline>
12 Characters: United Airlines
13 Ending element <airline>
14 Characters:
15
16 Starting element <departure>
17 Characters:
18
19 Starting element <city>
20 Characters: Los Angeles
21 Ending element <city>
22 Characters:
23
24 Starting element <state>
25 Characters: CA
26 Ending element <state>
27 Characters:
28
29 Starting element <airport>
30 Characters: LAX
31 Ending element <airport>
32 Characters:
33
34 Ending element <departure>
35 Characters:
36
37 Starting element <destination>
38 Characters:
39
40 Starting element <city>
41 Characters: Orlando
42 Ending element <city>
43 Characters:
44
```

```
45 Starting element <state>
46 Characters: FL
47 Ending element <state>
48 Characters:
49
50 Starting element <airport>
51 Characters: MCO
52 Ending element <airport>
53 Characters:
54
55 Ending element <destination>
56 Characters:
57
58 Starting element <departure-time>
59 Characters: 1130
60 Ending element <departure-time>
61 Characters:
62
63 Starting element <arrival-time>
64 Characters: 1845
65 Ending element <arrival-time>
66 Characters:
67
68 Starting element <meal>
69 Ending element <meal>
70 Characters:
71
72 Starting element <movie>
73 Ending element <movie>
74 Characters:
75
76 Ending element <flight>
77 ...Ending document
```

FIGURE 5. *Sample output of the SimpleParserWithHandler program.*

Error Handlers

When an XML document is parsed, errors and warnings may occur. If you do not write an error handler, then these errors are silently ignored, which "may result in bizarre behavior" according to the Java specification. For this reason, it is strongly suggested that you register an error handler with your parser.

To write an error handler, you write a class that implements the `ErrorHandler` interface, which in most circumstances is done by extending the `org.xml.sax.helpers.DefaultHandler` class, just like was done with the content handler.

The `ErrorHandler` interface contains three methods:

- `public void error(SAXParseException exception) throws SAXException`: This method is invoked when a recoverable error has occurred. Parsing will still continue unless your implementation of this method throws an exception.
- `public void warning(SAXParseException exception) throws SAXException`: This method is invoked when a problem has arisen during parsing that is not an error or a fatal error.
- `public void fatalError(SAXParseException exception) throws SAXException`: This method is invoked when a problem has arisen during parsing and the parser cannot continue successfully.

The following version of SimpleContentHandler has been modified to include the three methods of the ErrorHandler interface.

```
1  import org.xml.sax.*;
2  import org.xml.sax.helpers.DefaultHandler;
3
4  public class SimpleContentHandler extends DefaultHandler
5  {
6
7      public void error(SAXParseException exception)
8          throws SAXException
9      {
10         System.out.println("An error has occurred...");
11         exception.printStackTrace();
12     }
13
14     public void warning(SAXParseException exception)
15         throws SAXException
16     {
17         System.out.println("A warning has occurred..."
18 + exception);
19     }
20
21     public void fatalError(SAXParseException exception)
22         throws SAXException
23     {
24         System.out.println("Oops! Fatal error");
25         exception.printStackTrace();
26         System.exit(0);
27     }
28
29     //the remainder of the class remains the same
30 }
```

FIGURE 6. *An example of a SAX error handler.*

NOTE

When extending DefaultHandler, you do not need to define all three of the methods in the ErrorHandler interface, only those you are interested in.

An Example of an Error

To illustrate the case where an error occurs, suppose we modify `flight.xml` so that it does contain an error. The following shows a `<flight>` document being parsed using the `SimpleContentHandler` in Figure 6, except the `<arrival-time>` element is missing the `"/"` on the ending tag, thereby making it appear that there are two `<arrival-time>` elements, neither with a matching ending tag.

Running the `SimpleParserWithHandler` program in Figure 4 generates the following output:

```
Starting element <arrival-time>
Characters: 1845
Starting element <arrival-time>
Characters:

Starting element <meal>
Ending element <meal>
Characters:

Starting element <movie>
Ending element <movie>
Characters:

Oops! Fatal error
org.xml.sax.SAXParseException: The element type "arrival-time" must be
terminated by the matching end-tag "</arrival-time>".
    at
org.apache.xerces.util.ErrorHandlerWrapper.createSAXParseException(Er
rorHandlerWrapper.java:232)
    at
org.apache.xerces.util.ErrorHandlerWrapper.fatalError(ErrorHandlerWra
pper.java:213)
    at
SimpleParserWithHandler.main(SimpleParserWithHandler.java:22)
```

FIGURE 7. *A fatal error occurred while parsing.*

NOTE

The stack trace shown in Figure 7 has been shortened greatly for simplicity. What is important to notice is that the `fatalError()` method in the `SimpleContentHandler` class was invoked when the matching ending tag for `<arrival-time>` was not found.

Notes

Validating XML Using a DTD

A parser will check automatically to see if an XML document is well-formed. If the document is not well-formed, then a fatal error will occur. Parsers do not automatically check for validity, however. Instead, the `setValidation()` method in the `SAXParser` class must be invoked, passing in `true` to turn on validation.

NOTE

If the parser is told to check for validation, then while parsing the document the parser will compare the use of elements to the corresponding Document Type Definition (DTD) to verify the elements are being used properly.

The DTD file is specified in the XML document using the `DOCTYPE` keyword. For example, the following `<flight>` document specifies “flight.dtd” as the file containing the DTD for the `<flight>` element.

```
1  <?xml version='1.0' ?>
2
3  <!DOCTYPE flight SYSTEM "flight.dtd">
4
5  <flight>
6      <number>344</number>
7      <airline>United Airlines</airline>
8      <departure>
9          <city>Los Angeles</city>
10         <state>CA</state>
11         <airport>LAX</airport>
12     </departure>
13     <destination>
14         <city>Orlando</city>
15         <state>FL</state>
16         <airport>MCO</airport>
17     </destination>
18     <departure-time>1130</departure-time>
19     <arrival-time>1845</arrival-time>
20     <meal/>
21     <movie/>
22 </flight>
```

FIGURE 8. An XML document specifying the DTD to be used for validation.

IMPORTANT!

By default, SAX parsers do not validate while parsing. The SimpleParserWithHandler program defined in Figure 4 on page 347 used the `setValidation()` method to turn on validation. With validation turned on and a DTD specified in the XML document, the SAX parser will now validate the document as it parses it, generating errors and warnings when they occur.

For example, suppose the file in Figure 9 is the definition of “flight.dtd”. Notice that the `<meal>` element requires an attribute called “type”, which the `<flight>` document in Figure 8 does not contain.

```
1  <!ELEMENT flight (number, airline+, departure, destination,  
departure-time, arrival-time, meal*, movie?)>  
2  
3  <!ELEMENT number (#PCDATA)>  
4  <!ELEMENT airline (#PCDATA)>  
5  
6  <!ELEMENT departure (city, state, airport)>  
7  <!ELEMENT destination (city, state, airport)>  
8  <!ELEMENT city (#PCDATA)>  
9  <!ELEMENT state (#PCDATA)>  
10 <!ELEMENT airport (#PCDATA)>  
11  
12 <!ELEMENT departure-time (#PCDATA)>  
13 <!ELEMENT arrival-time (#PCDATA)>  
14  
15 <!ELEMENT meal EMPTY>  
16 <!ATTLIST meal type (breakfast|lunch|snack|supper) #REQUIRED>  
17  
18 <!ELEMENT movie EMPTY>  
19 <!ATTLIST movie title CDATA #IMPLIED>
```

FIGURE 9. The “flight.dtd” file.

Running the SimpleParserWithHandler program on the <flight> document in Figure 8 generates the following output:

```
Starting element <arrival-time>
Characters: 1845
Ending element <arrival-time>
Whitespace
An error has occurred...
org.xml.sax.SAXParseException: Attribute "type" is required and must
be specified for element type "meal".
    at
org.apache.xerces.util.ErrorHandlerWrapper.createSAXParseException(Error
HandlerWrapper.java:232)
    at
SimpleParserWithHandler.main(SimpleParserWithHandler.java:22)
Starting element <meal>
Ending element <meal>
Whitespace
Starting element <movie>
Ending element <movie>
Whitespace
Ending element <flight>
...Ending document
```

FIGURE 10. *The validation has successfully found an error.*

- Note that the stack trace in the output has been shortened for clarity. An error occurred when the parser read the <meal> element and checked it against the “flight.dtd”. According to the DTD, a <meal> element requires a “type” attribute, which is not present in the <meal> element being parsed.

A NOTE ABOUT MISSING ATTRIBUTES

Note that the missing “type” attribute does not generate a fatal error, just an error. The parser invoked the error() method of the ErrorHandler, then continued on with parsing. If there was no error handler assigned, then this problem with the <meal> element would have gone by unnoticed.

Notes

Creating a DOM Parser

We will now look at the steps involved in using the *Document Object Model* (DOM) to parse and also to create an XML document. DOM parses a document entirely different than SAX. A DOM parser takes the XML document and generates a tree-like data structure as it parses it, reading the entire document into memory.

DOM VS. SAX

DOM is useful in situations where the information in the XML document needs to be modified or traversed in a manner that is not possible with SAX, where the elements are parsed, then simply forgotten about.

For example, suppose a customer wanted to update the information in their account. The information could be parsed into a DOM tree, any changes could be made in memory to the tree, then the tree could be output back into an XML document. This would be quite tedious to code this using SAX, while DOM makes it fairly straightforward.

The class `DocumentBuilderFactory` in the `javax.xml.parsers` package is used to create a reference to a DOM parser, which is represented by the `DocumentBuilder` class, also found in the `javax.xml.parsers` package. To obtain a `DocumentBuilderFactory` object, you invoke the following static method in the `DocumentBuilderFactory` class :

```
public static DocumentBuilderFactory newInstance()
```

This method will search for a DOM parser factory in the following order:

1. Use the parser factory defined with the `javax.xml.parsers.DocumentBuilderFactory` system property.
2. Use the `javax.xml.parsers.DocumentBuilderFactory` system property defined in the `jaxp.properties` file, found in the `/<jre-directory>/lib` folder.
3. Use the `META-INF/services/javax.xml.parsers.DocumentBuilderFactory` property if using JAR files and the Services API.
4. If the parser factory property is not found in any of the above places, then the `newInstance()` method will use the default parser factory of the current platform.

Once you have the factory object, invoke its `newDocumentBuilder()` method to create a new parser instance:

```
public DocumentBuilder newDocumentBuilder()
```

Before you obtain a new `DocumentBuilder`, there are several methods in the `DocumentBuilderFactory` class available for configuring the parser:

- `public void setValidating(boolean b)`: By default, validation is set to false. Invoke this method if you want the parser to validate while parsing.
- `public void setNamespaceAware(boolean b)`: Use this method to denote if the parser being used provides support for XML namespaces. This value is false by default.
- `public void setAttribute(String featureName, boolean value)`: This method is used to set specific features of the parser.
- `public void setCoalescing(boolean b)`: If this is set to true, the DOM parser will convert any CDATA to a Text node and append it to the adjacent text node. This value is false by default.
- `public void setIgnoringWhitespace(boolean b)`: Tells the DOM parser to ignore whitespace or not. This value is false by default.
- `public void setIgnoringElementContentWhitespace(boolean b)`: Tells the DOM parser to ignore whitespace found in the content of an element. This value is false by default.
- `public void setExpandEntityReferences(boolean b)`: Tells the DOM parser to expand entity reference nodes. This value is true by default.

Parsing an XML Document Using DOM

Once you have configured the DOM parser and obtained a reference to it, you will have an object of type `DocumentBuilder`. The `DocumentBuilder` class contains several methods for parsing an XML document, depending on the input source of the document.

- `public Document parse(java.io.File source)`
- `public Document parse(org.xml.sax.InputSource source)`
- `public Document parse(java.io.InputStream source)`
- `public Document parse(String uri)`

Each `parse()` method defined in the `DocumentBuilder` class takes in as its argument the source of the XML document to be parsed. The XML document can be a `File`, an `InputStream`, an `InputSource`, or a `String` representing the URI of the document.

NOTE

Notice the `Document` parse methods are similar to those in the `SAXParser` class, except the parse methods in `DocumentBuilder` do not have a `DefaultHandler` argument. This is because a DOM parser does not generate events like the SAX parser does. For example, there is no such thing as a content handler for DOM. The content will be part of the tree structure, which is returned as a `org.w3c.dom.Document` object from each parse method.

- Error handling is still important in DOM, just like it was when using SAX. If you want to be aware of parsing errors in DOM, you need to write a class that implements `ErrorHandler` and assign it to the parser using the method:

```
public void setErrorHandler(ErrorHandler eh)
```

- Similarly, you can write a class to resolve entities by implementing `EntityResolver` and assigning the handler to the parser using the method:

```
public void setEntityResolver(EntityResolver er)
```

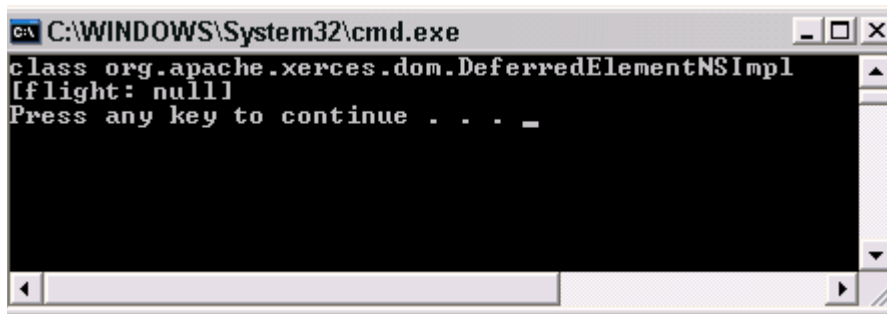
The following program shows how to create a DOM parser and use it to parse an XML document. In this program, the XML document is a File whose filename is passed in as the first command line argument.

```
1  import javax.xml.parsers.DocumentBuilderFactory;
2  import javax.xml.parsers.DocumentBuilder;
3  import org.w3c.dom.*;
4  import java.io.File;
5
6  public class MyDOMParser
7  {
8      public static void main(String [] args)
9      {
10         try
11         {
12             //Use the default DOM parser factory
13             DocumentBuilderFactory factory =
14                 DocumentBuilderFactory.newInstance();
15
16             factory.setValidating(true);
17             factory.setNamespaceAware(true);
18
19             DocumentBuilder parser =
20                 factory.newDocumentBuilder();
21
22             //assign an error handler
23             parser.setErrorHandler(new MyErrorHandler());
24
25             //parse the filename passed in on the command line
26             Document document =
27                 parser.parse(new File(args[0]));
28
29             //invoke the toString() method of the root element
30             Element root = document.getDocumentElement();
31             System.out.println(root.getClass());
32             System.out.println(root);
33
34         }catch(Exception e)
35         {
36             e.printStackTrace();
37         }
38     }
39 }
```

FIGURE 11. *A simple DOM parser.*

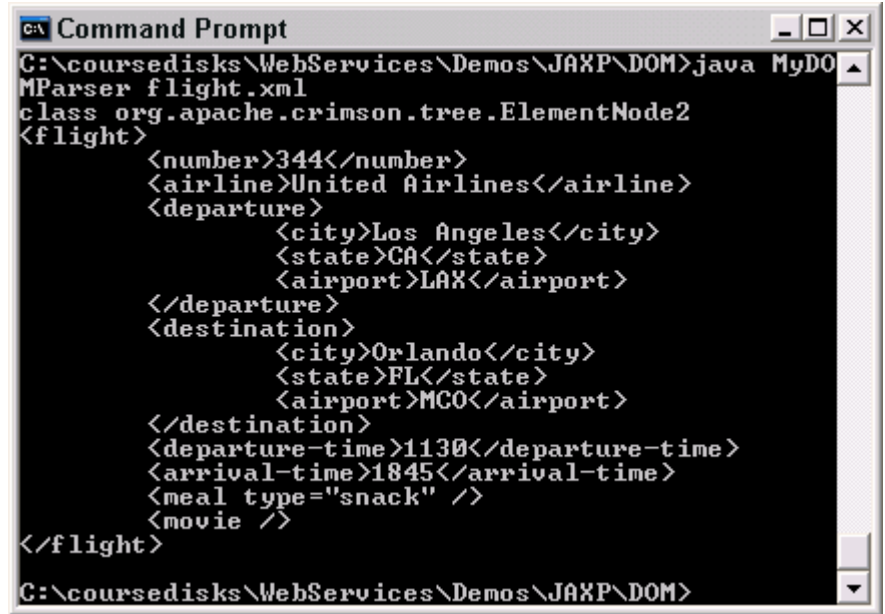
Choosing a DOM Parser

The output of the MyDOMParser varies depending on which parser is being used by the DocumentBuilderFactory. For example, the Xerces parser, available from Apache at <http://xml.apache.org>, generates the following output:



```
C:\WINDOWS\System32\cmd.exe
class org.apache.xerces.dom.DeferredElementNSImpl
[flight: null]
Press any key to continue . . . _
```


Running the same program using the Crimson parser, also available at <http://xml.apache.org>, generates the following output:



```
Command Prompt
C:\coursedisks\WebServices\Demos\JAXP\DOM>java MyDOMParser flight.xml
class org.apache.crimson.tree.ElementNode2
<flight>
  <number>344</number>
  <airline>United Airlines</airline>
  <departure>
    <city>Los Angeles</city>
    <state>CA</state>
    <airport>LAX</airport>
  </departure>
  <destination>
    <city>Orlando</city>
    <state>FL</state>
    <airport>MCO</airport>
  </destination>
  <departure-time>1130</departure-time>
  <arrival-time>1845</arrival-time>
  <meal type="snack" />
  <movie />
</flight>
C:\coursedisks\WebServices\Demos\JAXP\DOM>
```

NOTE

To choose a parser specifically, use the following property:

```
javax.xml.parsers.DocumentBuilderFactory
```

- For example, the following command line runs MyDOMParser specifying that the class `org.apache.crimson.jaxp.DocumentBuilderFactory` (which is the Crimson parser) be used as the factory for building documents:

```
java -Djavax.xml.parsers.DocumentBuilderFactory=
org.apache.crimson.jaxp.DocumentBuilderFactory MyDOMParser flight.xml
```

Understanding DOM Nodes

The DOM tree structure created by the DOM parser consists of nodes to represent the elements, content, and other information of the XML document. There are many types of nodes, but two of importance that we will discuss are:

- **Element node:** A node that represents an element (tag) in an XML document.
- **Text node:** A node that represents the actual contents of an element in an XML document.

The `org.w3c.dom` package contains interfaces for representing the different types of nodes. Some of these interfaces include:

- **Node:** The parent interface of all nodes (including `Document`), it contains methods for traversing the tree as well as removing or appending child nodes.
- **Document:** Represents the entire XML document. This was the data type returned by the `parse()` methods. You use the methods of this interface to obtain and/or create nodes off the root node of the tree.
- **Element:** Represents a tag in an XML document. This interface contains methods for obtaining the name of the tag and also for obtaining child nodes of the element.
- **Text:** Represents the characters of the XML document. You use this interface to set and get the data of an element.
- **Comment:** Represents a comment in an XML document.
- **Attr:** Represents an attribute of an element.

The Document Interface

The Document interface contains the following methods for creating the various nodes of the XML tree using DOM.

- `public Element createElement(String tagName)`: creates a tag with the given name.
- `public Text createTextNode(String data)`: creates a text node that can be appended to an element. The resulting Text object represent the characters of an XML document.
- `public Attr createAttribute(String attrName)`: creates an Attribute that can be attached to an Element using the `setAttributeNode()` method in the Element interface.
- `public EntityReference createEntityReference(String name)`: used for creating a reference to an entity.
- `public Comment createComment(String comment)`: creates a Comment node, using the given String as the comment.

Once you have created a Node, you can append it to an existing Node using the following method, found in the Node interface:

- `public Node appendNode(Node newChild)`: adds the `newChild` node to the Node that is invoking this method. The return value is a reference to the new Node that was added.

Creating an XML Document Using DOM

The following program demonstrates how to use the methods above in the Document interface along with the `appendChild()` method of the Node interface to create a new XML document in memory.

```
1  import org.w3c.dom.*;
2  import javax.xml.parsers.*;
3  import java.io.*;
4
5  public class CreateFlightDocument
6  {
7      public static void main(String [] args)
8      {
9          try
10         {
11             //obtain a reference to a DocumentBuilder
12             DocumentBuilderFactory dbf =
13                 DocumentBuilderFactory.newInstance();
14
15             DocumentBuilder builder = dbf.newDocumentBuilder();
16
17             //create a new DOM document
18             Document doc = builder.newDocument();
19
20             //create the root node and append to the Document
21             Element root = doc.createElement("flight");
22             doc.appendChild(root);
23
24             //create the <number> element and append to <flight>
25             Element number = doc.createElement("number");
26             number.appendChild(doc.createTextNode("1255"));
27             root.appendChild(number);
28
29             //create the <airline> element and append to <flight>
30             Element airline = doc.createElement("airline");
31             airline.appendChild(doc.createTextNode("Northwest
Airlines"));
32             root.appendChild(airline);
33
34             //create the <departure> element and all its subelements
35             Element depart = doc.createElement("departure");
36
37             Element city = doc.createElement("city");
38             city.appendChild(doc.createTextNode("Denver"));
39             Element state = doc.createElement("state");
```

```
40      state.appendChild(doc.createTextNode("CO"));
41      Element airport = doc.createElement("airport");
42      airport.appendChild(doc.createTextNode("DEN"));
43
44      depart.appendChild(city);
45      depart.appendChild(state);
46      depart.appendChild(airport);
47
48      root.appendChild(depart);
49
50      //create the <meal> element with a "type" attribute
51      Element meal = doc.createElement("meal");
52      Attr type = doc.createAttribute("supper");
53      meal.setAttributeNode(type);
54      root.appendChild(meal);
55
56      System.out.println(root.toString());
57  } catch (Exception e)
58  {
59      e.printStackTrace();
60  }
61  }
62 }
```

FIGURE 12. *Creating a new XML document in memory using DOM.*

The XML document created in Figure 12 looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<flight>
  <number>1255</number>
  <airline>Northwest Airlines</airline>
  <departure>
    <city>Denver</city>
    <state>CO</state>
    <airport>DEN</airport>
  </departure>
  <meal type="supper"/>
</flight>
```

NOTE

If you use the Crimson parser, then invoking the `toString()` method on the root element causes the document to be displayed. If the parser you are using does not display the XML document when you invoke `toString()`, then you can use XSLT, discussed in the next section, to output a DOM document created in memory to a file or other output stream.

An Overview of XSLT

XSLT stands for *XML Stylesheet Language for Transformations*. The XSLT API defines a mechanism for translating XML into other formats. For example, you can use XSLT to convert XML to HTML, or XML to a PDF file, or XML to plain text, or any transformation you create.

XSLT uses a transformer, which is obtained through JAXP using the `TransformerFactory` class found in the `javax.xml.transform` package. The transformer takes an XML document and converts it to another document following the rules defined in a *style sheet*. A style sheet is written using the XML Stylesheet Language, referred to as XSL.

We won't discuss the details of the XML Stylesheet Language for Transformations in this course. However, some of the common transformations are provided in the JAXP API. For example, you can transform a Document in memory to an XML file by performing the following steps:

- Obtain a reference to a `javax.xml.transform.TransformerFactory` by invoking the static method `newInstance()` in the `TransformerFactory` class.
- Create a new `javax.xml.transform.Transformer` object by invoking the `newTransformer()` method of the `TransformerFactory` object.
- If the source of the XML is a Document, then instantiate a new `DOMSource` object, which is found in the `javax.xml.transform.dom` package. The constructor of `DOMSource` can take in any Node on a Document tree.
- The `SAXSource` class in the `javax.xml.transform.sax` package is used for transforming XML being parsed by a SAX parser. The constructors of `SAXSource` take in the `XMLReader` that is parsing the document.
- To convert the source of the XML to text, use the `StreamSource` class, which is found in the `javax.xml.transform.stream` package. The constructors of `StreamSource` take in a file or output stream where the text is to be output.

Notes

Output a DOM Document to a File

To demonstrate using XSLT, the following CreateFlightDocument program is a modification of the one in Figure 12 on page 367. The same Document is created in memory, but in Figure 13 the Transformer class is used to transform the XML Document in memory to text, that is sent to an output stream (which in this example is a file).

```
1  import org.w3c.dom.*;
2  import javax.xml.parsers.*;
3  import java.io.*;
4  import javax.xml.transform.*;
5  import javax.xml.transform.dom.*;
6  import javax.xml.transform.stream.*;
7
8  public class CreateFlightDocument
9  {
10     public static void main(String [] args)
11     {
12         try
13         {
14             //obtain a reference to a DocumentBuilder
15             DocumentBuilderFactory dbf =
16                 DocumentBuilderFactory.newInstance();
17             DocumentBuilder builder = dbf.newDocumentBuilder();
18             //create a new DOM document
19             Document doc = builder.newDocument();
20
21             //remainder of the class definition stays the same...
22
23             //Use XSLT to transform doc into a text stream
24             TransformerFactory tf =
25                 TransformerFactory.newInstance();
26             Transformer transformer = tf.newTransformer();
27             DOMSource source = new DOMSource(doc);
28             StreamResult out = new StreamResult("test.xml");
29             transformer.transform(source, out);
30
31         }catch(Exception e)
32         {
33             e.printStackTrace();
34         }
35     }
36 }
```

FIGURE 13. *Outputting a DOM Document to a file.*

The test.xml file that gets created from the CreateFlightDocument program in Figure 13 looks like:

```
<?xml version="1.0" encoding="UTF-8"?>

<flight>
  <number>1255</number>
  <airline>Northwest Airlines</airline>
  <departure>
    <city>Denver</city>
    <state>CO</state>
    <airport>DEN</airport>
  </departure>
  <meal type=""/>
</flight>
```

Lab 1: Generating an XML Document

Objective: In this lab, you will write a class that generates an XML document using DOM. You will modify the `FinalServlet` from the previous chapter so that when a customer completes an order, the servlet will generate an XML document to represent that order.

Perform the following tasks:

1. Write a new class named `OrderDocument`, declaring it in the `com.best` package. Save the file in the `%CATALINA_HOME%\webapps\best\WEB-INF\classes` directory.
2. Add a static method named `createDocument()` that has a single parameter of type `com.best.Order` and returns a `org.w3c.dom.Document`.
3. Within the `createDocument()` method, create an XML document to represent the `Order` object passed in. You can use the DTD you created from “Lab 1: Designing an XML Document” on page 328 of Chapter 9, or you can use the following sample as a guideline:

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
  <name>Robert Smith</name>
  <address>123 Main St., London, UK</address>
  <credit-card>
    <number>1111222233334444</number>
    <exp-month>12</exp-month>
    <exp-year>2006</exp-year>
  </credit-card>
  <items>
    <item>
      <number>6532</number>
      <quantity>1</quantity>
    </item>
    <item>
      <number>431</number>
      <quantity>1</quantity>
    </item>
    <item>
      <number>694</number>
      <quantity>2</quantity>
    </item>
  </items>
</order>
```

```
        </item>
    </items>
    <total>2538.96</total>
</order>
```

FIGURE 14. *The order.xml file.*

4. Be sure to return the Document that you created in the createDocument() method.
5. Save and compile your OrderDocument class.
6. Open the FinalServlet.java file, which is found in the directory %CATALINA_HOME%\webapps\best\WEB-INF\classes.
7. After the statement order.setItems(cart.getItems()), invoke the createDocument() method, passing in the "order" reference.
8. Using a Transformer object, output the Document that is returned from createDocument() to a new file named "order.xml".
9. Save and compile the FinalServlet class.
10. Restart the Web server (or reload the /best application) and fill a shopping cart with items. Purchase the items and view the "order.xml" file that is created.

What you should see: Your order.xml file should look similar to the one in Figure 14.



Index

Symbols

134, 224, 232–233
<EMBED> 178
<OBJECT> 178

A

Accept header 57
Accept-Encoding header 57
Accept-Language header 57
action 178
addCookie() 63, 92
Apache 362
appendChild() 366
appendNode() 365
application object 158, 208
application scope 78, 88, 204, 208–209
ATTLIST 323
Attr 364
attribute 314

B

beanName 189
body tag 225
BodyContent 245
BodyTag 227–228, 240–241, 248
BodyTagHandler 245
BodyTagSupport 227, 240–241

C

CDATA 323, 359
class 189
Comment 364
config object 158
Connection 153
Connection header 56, 58
Connectors 19
ContentHandler 334, 344–345
Content-Length 59
contentType 177
Cookie 92, 94

Cookies 90, 92
CORBA 16–17
createAttribute() 365
createComment() 365
createElement() 365
createEntityReference() 365
createTextNode() 365
Crimson 363, 367
custom tags 224

D

Declarations 150, 152
default threading model 117
DefaultHandler 342, 345, 350, 360
Delegation Model 125
DELETE 32, 54
destroy() 131
Directives 150
directives 176
doAfterBody() 241
doCatch() 248
DOCTYPE 354
Document 360, 364, 366, 368
Document Object Model. See DOM
Document Object Model. See DOM
Document Type Definition 319–323
DocumentBuilder 336, 359–361
DocumentBuilderFactory 336, 358, 361–362
doEndTag() 241, 245
doFilter() 130
doFinally() 248
doGet() 31, 33
doInitBody() 241
DOM 332, 336, 360
 creating XML 366
 nodes 364
 parser 358
 vs. SAX 358
DOMSource 368
doPost() 31

doStartTag() 228, 241, 245
DTD 354–356
DTD - see Document Type Definition
DTDHandler 344

E

ELEMENT 322
Element 364
element 312
EMPTY 323
empty element 312
encodeURL() 102
Enterprise JavaBeans
 definition 4
entity 316
EntityResolver 344, 360
Error Handlers
 for XML parsers 350
error page 218
error() 350, 356
ErrorHandler 344, 356, 360
ErrorHandler interface 350–351
errorPage 177
EVAL_BODY_BUFFERED 241
EVAL_BODY_INCLUDE 228, 241
EVAL_PAGE 228
exception object 158, 218
Expressions 150, 156
extends 177

F

fatalError() 350
Filter 130–131
filter chain 130
forward() 68

G

GenericServlet 30–31
GET 32, 54–56, 64
getAttribute() 61, 80, 88, 98, 161, 206, 208, 212
getAttributeNames() 80, 88, 206, 208
getComment() 95
getCookies() 60, 92, 161
getHeader() 60, 161
getMethod() 60, 161
getName() 94
getOutputStream() 62
getParameter() 61, 64, 161–162
getParameters() 64, 162

getProperty 193
getRequestDispatcher() 60, 68, 161
getServletContext() 68
getSession() 60, 98, 161
getValue() 94
getWriter() 62

H

HEAD 54
Host header 56
HTML 33
HTTP 32, 54
HTTP requests 56
HTTP response 58
HttpServlet 30–31
HttpServletRequest 60, 68, 80, 92, 98, 158, 160
HttpServletResponse 62–63, 68, 102, 158
HttpSession 98, 100, 158, 210, 212
HttpSessionActivationListener 124
HttpSessionAttributeListener 124
HttpSessionBindingEvent 124
HttpSessionBindingListener 124
HttpSessionEvent 124
HttpSessionListener 124, 126

I

id 189
IMPLIED 323
import 177
include 172, 174, 176
include() 68
init() 88
init(FilterConfig fc) 131
instantiate() 189
isErrorPage 158, 177, 218
isThreadSafe 177
IterationTag 227, 240
IteratorTag 248

J

J2EE 1–22
 list of technologies 3
Java API for XML Processing. See JAXP
Java IDL 16
Java Message Service, see JMS
Java Naming and Directory Interface, see JNDI
Java Transaction API
 overview 14
Java Transaction Service 14

JavaBeans 187
 in JSPs 188
JavaMail 18
JavaServer Pages 9
 overview 140
JavaServer Pages. See also JSP 139
JavaServer Pages. See also JSP. 140
javax.servlet 30
javax.servlet.http 30
javax.servlet.jsp.tagext Package 227
javax.xml.parsers 333, 340
javax.xml.transform 333
JAXP 331, 333, 368
 overview 332
JDBC 7–8, 26
JMS
 overview 15
JNDI
 overview 13
JSP 142, 177, 224, 236
 action tags 179
 actions 178
 and JavaBeans 187, 189
 creating custom tags 226
 custom tags 223
 declarations 152
 deploying 148
 directive 151
 directives 176
 error pages 218
 expressions 156
 Hello, World 146
 include 172
 lifecycle 144–145
 parameters 162
 scope 204
 scriptlets 166
 tags 150–151
 the implicit objects 158
jsp
 forward 178
 getProperty 178, 193
 include 172, 178
 param 178
 plugin 178
 setProperty 178, 192
 useBean 178, 189–190, 204
jspInit() 153
JspTag 227

JspWriter 158

L

language 177

M

Model 188

MVC 188

N

namespace 326

newDocumentBuilder() 359

newInstance()

 DocumentBuilderFactory 358

 TransformerFactory 368

Node 364–366, 368

O

OPTIONS 54

org.w3c.dom 333, 336, 364

org.xml.sax 333–334

org.xml.sax.helpers 334, 340

out object 158

P

page 177

page directive 177

page object 158

page scope 78, 204

PageContext 158, 245

pageContext object 158

parameters 162

parse() method 342, 344, 347, 360

parser 334, 338, 342

PCDATA 322

PDF 368

POST 32, 54–55, 64

PUT 32, 54

R

release() 241

removeAttribute() 61, 80, 88, 206, 208

request header 60

request object 158, 160, 206

 methods 161

request scope 78, 80, 204, 206–207

RequestDispatcher 68

REQUIRED 323

response body 59

response object 158

RMI-IIOP 17

root element 313

S

SAX 332, 334, 338, 342, 345

SAXParser 338–340, 360

SAXParserFactory 338–340

SAXSource 368

scope 78, 189, 204, 207

scriptlet 166

Scriptlets 150

sendError() 63

Server header 58

service() 27, 31, 33, 60, 62, 92, 116–118, 120

Servlet

- forwarding 68

- including 68

- multiple clients 116

- parameters 64

- scope 78

- threading models 110

Servlet API 30

Servlet interface 31

Servlet Listeners 124

Servlet Thread Models 116

ServletConfig 88, 158

ServletContext 68, 158, 208

ServletContextAttributeEvent 124

ServletContextAttributeListener 124

ServletContextEvent 124

ServletContextListener 124

ServletOutputStream 62

ServletRequest 80

Servlets 5, 23–24, 26, 113, 142

- advantages of 26

- destroying 115

- destroying cleanly 120

- initializing 114

- lifecycle 28

- using filters 130

servlets

- definition 24

session 177, 212–213

session object 158, 210, 212–213

session scope 78, 90, 204, 210, 214

setAttribute() 61, 80, 88, 98, 161, 206, 208, 212, 359

setBodyContent(BodyContent) 241

setCoalescing() 359

setComment() 95

setContentLength() 62

setContentType() 62

setEntityResolver() 360

setErrorHandler() 360

setExpandEntityReferences() 359

setIgnoringElementContentWhitespace() 359

setIgnoringWhitespace() 359

setMaxAge() 94

setNamespaceAware() 359

setPageContext() 228, 241

setParent() 228, 241

setProperty 192

setValidating() 359

setValidation() 354–355

setValue() 94

SimpleTag 228

Simple API for XML. See SAX

simple tag 225

SimpleTagSupport 227

SingleThreadModel 116, 118

SKIP_BODY 228, 241, 245

SKIP_PAGE 228

status line 58

StreamSource 368

style sheet 368

T

Tag 248

tag 312

Tag Handler 228

Tag Library Descriptor 224, 226, 232

taglib 176, 236

TagSupport 227–228

Text 359, 364

TLD 224, 226, 232, 234, 246

Tomcat 148

TRACE 54

TransformerFactory 368

Transformer 368, 370

TryCatchFinally 248

type 189

U

URL Rewriting 90, 102

useBean 189, 207, 209

User-Agent header 56

V

valid 318
validating forms 198
View 188

W

warning() 350
Web services 20–21
well-formed 318
writeOut() 245

X

Xerces 362
XML 11–12, 311–327
 attributes 314–315
 document type 313
 elements 312
 empty element 312
 entity elements 316
 namespaces 326
 root element 313
 tags 312
 valid 318
 validating with a DTD 354
 well-formed 318
XML Stylesheet Language for Transformations. See also
 XSLT.
XMLReader 334, 368
XMLReaderFactory 334
XSLT 332, 367–368, 370

