KarpagamAcademy of Higher Education
(Established Under Section 3 of UGC Act 1956)
Eachanari Post, Coimbatore – 641 021. INDIA
Phone : 0422-2611146, 2611082 Fax No : 0422 -2611043

Semester – III

18CAU304B **STRUTS FRAMEWORK** **3H – 3C**

Instruction Hours / week: L: 3 T: 0 P: 0 **Marks:** Int : **40** Ext : **60** Total: **100**

End Semester Exam: 3 Hours

Course Objectives:

To help students to

- Know the components of Struts Application
- Implement JSP functions using Struts
- Develop web applications using Struts
- understand the Model, View, Controller (MVC) design pattern and how it is applied by Struts Framework
- how to perform client & server side validation using Struts Validator Framework

Course Outcome:

- able to construct web based applications and Identify where data structures are appearing in them.
- able to generate dynamic content using JSP
- able to develop EJB programs and get familiar with Struts framework

UNIT -I

Introduction, Understanding the MVC Design Pattern, The Struts Implementation of the MVC, Directory Structure, Web Application Deployment Descriptor, The Tomcat JSP/Servlet Container, Installing and Configuring Tomcat, Testing Your Tomcat Installation, An Overview of the Java Servlet and JavaServer Pages, The GenericServlet and HttpServlet Classes, Life Cycle of a Servlet. Struts OverView, Life Cycle of Struts.

UNIT -II

Components of a Struts Application, The Controller, The View, DynaActionForm & LazyDynaBean, ActionServlet, RequestProcessor, ActionForm, IncludeAction, Forward Action, LocaleAction, DispatchAction, LookupDispatchAction, MappingDispatchAction, EventDispatchAction, SwitchAction, Interceptors, Implementing Custom interceptors, Struts Validation, Exception Handling, Managing Errors, Struts Error Management - ActionError, ActionErrors, Creating Custom ActionMappings, Struts JDBC Connection, Using a DataSource in Struts Application, Debugging Struts Applications.

UNIT -III

The struts-config.xml, The Struts Subelements, The icon Tag Subelement, display-name Tag Subelement, description Tag Subelement, set-property Tag Subelement, Adding a Struts DataSource, Adding FormBean Definitions, Adding Global Forwards, Adding Actions, Adding a RequestProcessor, Adding Message Resources, Adding a Plug-in. The Bean Tag Library, Installing the Bean Tags, bean:cookie Tag, bean:define Tag, bean:header Tag, bean:include Tag,
bean:message Tag, bean:page Tag, bean:parameter Tag ,bean:resource Tag,<bean:size Tag, bean:struts Tag, bean:write Tag

UNIT -IV

HTML Tag Library, Base Tag, Button Tag, Cancel Tag, Checkbox Tag, Errors Tag, Form Tag, Hidden Tag, Html Tag, Image Tag, Img Tag, Link Tag, Multibox Tag, Select Tag, Option Tag, Options Tag, Password Tag, Radio Tag, Reset Tag, Rewrite Tag, Submit Tag, Text Tag, Textarea Tag

UNIT -V

The Logic Tag Library , Empty Tag, notEmpty Tag ,equal Tag, notEqual Tag, forward Tag, redirect Tag, greaterEqual Tag, greaterThan Tag, iterate Tag, lessEqual Tag, lessThan Tag, match Tag, notMatch Tag, present Tag, notPresent Tag

Suggested readings

1. James Goodwill,(2002). *Mastering Jakarta Struts*, Wiley Publishing, Inc.

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
Coimbatore-21

SYLLABUS

DEPARTMENT OF CS, CA & IT

STAFF NAME: K. GEETHA

SUBJECT NAME: STRUTS FRAMEWORK **SUBJECT CODE: 18CAU304B**

SEMESTER: III **CLASS: II BCA**

S.No.	Lecture Duration (Period)	Topics to be Covered	Support Materials
		Unit – I	
1.	1	Introduction, Understanding the MVC Design pattern, The Struts Implementation of MVC	T1: 1-4
2.	1	Directory Structure, Web Application Deployment Descriptor	T1: 6-7
3.	1	The Tomcat JSP/Servlet container Installing and configuring Tomcat	T1: 8-11
4.	1	Testing your Tomcat Installation, An overview of the Java Servlet and Javaserver pages	T1: 9-12
5.	1	The Generic Servlet and HttpServlet Classes, Life Cycle of a Servlet	T1: 13-14
6.	1	Struts Overview, Life Cycle of Struts	T1: 48-58
7.	1	Recapitulation and Discussion of important questions	T1:

Text Book 1 (T1) : James Goodwill (2002), Mastering Jakarta Struts, Wilely Publishing, Inc.

		Total No. of Hours Planned for Unit-I	**7**
S.No.	**Lecture Duration (Period)**	**Topics to be Covered**	**Support Materials**
		Unit – II	
1.	1	Components of a Struts Application, The Controller, The View, DynaActionForm & LazyDynaBean	T1: 82-84
2.	1	Acttion Servlet, Request Processor, ActionForm, IncludeAction, Forward Action, LocaleAction	T1: 62-72
3.	1	DispatchAction, LookupDispatchAction, MappingDispatchAction,EventDispatchAction,SwitchAction	T1: 75-76
4.	1	Interceptors, Implementing Custom Interception, Struts Validation, Exception Handling	T1: 93-96
5.	1	Managing Errors, Struts Error Management, Action Error, Action Errors	T1: 90-94
6.	1	Creating Custom Action Mapping, Struts JDBC Connection	T1: 100-103
7.	1	Using a Datasource in Struts Application, Debugging Stuts Applications	T1: 107-109
8.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours Planned for Unit-II	**8**

Text Book 1 (T1) : James Goodwill (2002), Mastering Jakarta Struts, Wilely Publishing, Inc.

S.No.	Lecture Duration (Period)	Topics to be Covered	Support Materials
Unit – III			
1.	1	The Struts – Confil.Xml, The Struts Subelements, The iconTag subelement, display – nametag subelement	T1: 177-178
2.	1	description Tag subelement, set property Tag subelement, Adding a struts Datasource	T1: 178-179
3.	1	Adding FormBean Definitions, Adding Global Forwards, Adding Actions, Adding a Request Processor	T1:181-182
4.	1	Adding Message Resources, Adding a Plug in, The Bean Tag Library, Installing the Bean Tags, bean: cookie Tag	T1:184-187
5.	1	bean: define Tag, bean: header Tag, bean: include Tag, bean: message Tag, bean : page Tag	T1: 188-192
6.	1	bean: parameter Tag, bean: resource Tag, bean: size Tag, bean: struts Tag, bean: write Tag	T1: 193-195
7.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours Planned for Unit-III	**7**

Text Book 1 (T1) : James Goodwill (2002), Mastering Jakarta Struts, Wilely Publishing, Inc.

S.No.	Lecture Duration (Period)	Topics to be Covered	Support Materials
Unit – IV			
1.	1	HTML Tag Library, Base Tag, Button Tag, Cancel Tag	T1: 197-200
2.	1	Check box Tag, Errors Tag, Form Tag, Hidden Tag	T1: 202-206
3.	1	HTML Tag, Image Tag, Img Tag, Link Tag, Multibox Tag	T1: 207-216
4.	1	Select Tag, Option Tag, Options Tag, Password Tag, Radio Tag	T1: 219-227
5.	1	Reset Tag, Rewrite Tag, Submit Tag, Text Tag, Textarea Tag	T1: 229-237
6.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours Planned for Unit-IV	**6**

Text Book 1 (T1) : James Goodwill (2002), Mastering Jakarta Struts, Wilely Publishing, Inc.

S.No.	Lecture Duration (Period)	Topics to be Covered	Support Materials
Unit – V			
1.	1	Introduction to Perl, Perl Documentation	T2: 57-59,W8
2.	1	Perl Syntax Rules, Declaring Variables with use Strict	T2: 59-63, W9
3.	1	Scalar, Array and Hash Variables	T2: 63-71
4.	1	Operators	T2: 71-76
5.	1	Flow Control Constructs	T2: 77-81, W9
6.	1	Regular Expressions, Functions	T2: 81-92
7.	1	File I/O, Additional Perl Constructs	T2: 93-103
8.	1	Making Operating System Calls, A Quick Introduction to Object Oriented Programming, What we didn't talk about	T2: 103-107
9.	1	Recapitulation and Discussion of important questions	
10.	1	Discussion of previous ESE question papers	
11.	1	Discussion of previous ESE question papers	
12.	1	Discussion of previous ESE question papers	
Total No. of Hours Planned for Unit-V			**12**
Total No. of Hours Planned for this Syllabus			**48**

Text Book 1 (T1) : James Goodwill (2002), Mastering Jakarta Struts, Wilely Publishing, Inc.

Introduction

Struts combines two of the most popular server−side Java technologies—JSPs and servlets—into a server−side implementation of the Model−View−Controller design pattern. It was conceived by Craig McClanahan in May of 2000, and has been under the watchful eye of the Apache Jakarta open source community since that time.

The remarkable thing about the Struts project is its early adoption, which is obviously a testament to both its quality and utility. The Java community, both commercial and private, has really gotten behind Struts. It is currently supported by all of the major application servers including BEA, Sun, HP, and (of course) Apache's Jakarta−Tomcat. The Tomcat group has even gone so far as to use a Struts application, in its most recent release 4.0.4, for managing Web applications hosted by the container.

Chapter 1: Introducing the Jakarta Struts Project and Its Supporting Components

We start by providing a high−level description of the Jakarta Struts project. We then describe Java Web applications, which act as the packaging mechanism for all Struts applications. We conclude this chapter with a discussion of the Jakarta Tomcat JSP/servlet container, which we use to host all of our examples throughout the remainder of this text.

The Jakarta Struts Project

The Jakarta Struts project, an open−source project sponsored by the Apache Software Foundation, is a server−side Java implementation of the Model−View−Controller (MVC) design pattern. The Struts project was originally created by Craig McClanahan in May 2000, but since that time it has been taken over by the open−source community.

The Struts project was designed with the intention of providing an open−source framework for creating Web applications that easily separate the presentation layer and allow it to be abstracted from the transaction/data layers. Since its inception, Struts has received quite a bit of developer support, and is quickly becoming a dominant factor in the open−source community.

Note There is a small debate going on in the development community as to the type of design pattern that the Struts project most closely resembles. According to the documentation provided by the actual developers of the Struts project, it is patterned after the MVC, but some folks insist that it more closely resembles the Front Controller design pattern described by Sun's J2EE Blueprints Program. The truth is that it does very much resemble the Front Controller pattern, but for the purpose of our discussions, I am sticking with the developers. If you would like to examine the Front Controller yourself, you can find a good article on this topic at the Java Developer Connection site: http://developer.java.sun.com/developer/technicalArticles/J2EE/despat/.

Understanding the MVC Design Pattern

To gain a solid understanding of the Struts Framework, you must have a fundamental understanding of the MVC design pattern, which it is based on. The MVC design pattern, which originated from Smalltalk, consists of three components: a Model, a View, and a Controller. Table 1.1 defines each of these components.
Table 1.1: The Three Components of the MVC

Component	Description
Model	Represents the data objects. The Model is what is being manipulated and presented to the user.
View	Serves as the screen representation of the Model. It is the object that presents the current state of the data objects
Controller	Defines the way the user interface reacts to the user's input. The Controller component is the object that manipulates the Model, or data object.

We will discuss each of these components in more detail throughout this chapter. Some of the major benefits of using the MVC include:

Reliability: The presentation and transaction layers have clear separation, which allows you to change the look and feel of an application without recompiling Model or Controller code.

High reuse and adaptability: The MVC lets you use multiple types of views, all accessing the same server−side code. This includes anything from Web browsers (HTTP) to wireless browsers (WAP).

Very low development and life−cycle costs: The MVC makes it possible to have lower−level programmers develop and maintain the user interfaces.

Rapid deployment: Development time can be significantly reduced because Controller programmers (Java developers) focus solely on transactions, and View programmers (HTML and JSP developers) focus solely on presentation.

Maintainability: The separation of presentation and business logic also makes it easier to maintain and modify a Struts−based Web application.

The Struts Implementation of the MVC

The Struts Framework models its server−side implementation of the MVC using a combination of JSPs, custom JSP tags, and Java servlets. In this section, we briefly describe how the Struts Framework maps to each component of the MVC. When we have completed this discussion, we will have drawn a portrait similar to Figure 1.1.

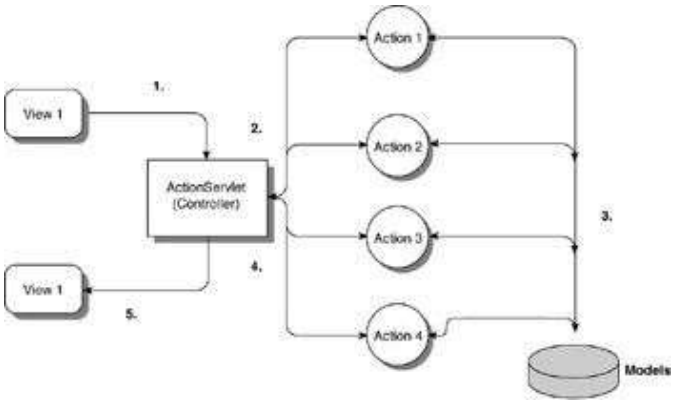


Figure 1.1: The Struts implementation of the MVC.

Figure 1.1 depicts the route that most Struts application requests follow. This process can be broken down into five basic steps. Following these steps is a description of the ActionServlet and Action classes.

1. A request is made from a previously displayed View.
2. The request is received by the ActionServlet, which acts as the Controller, and the ActionServlet looks up the requested URI in an XML file (described in Chapter 3, "Getting Started with Struts"), and determines the name of the Action class that will perform the necessary business logic.
3. The Action class performs its logic on the Model components associated with the application.
4. Once the Action has completed its processing, it returns control to the ActionServlet. As part of the return, the Action class provides a key that indicates the results of its processing. The ActionServlet uses this key to determine where the results should be forwarded for presentation.
5. The request is complete when the ActionServlet responds by forwarding the request to the View that was linked to the returned key, and this View presents the results of the Action.

The Model

The Struts Framework does not provide any specialized Model components; therefore, we will not dedicate an entire chapter to the Model component. Instead, we will reference Model components as they fit into each example.

The View

Each View component in the Struts Framework is mapped to a single JSP that can contain any combination of Struts custom tags. The following code snippet contains a sample Struts View:

```
<%@page language="java">
<%@taglib uri="/WEB-INF/struts-html.tld" prefix="html">

<html:form action="loginAction.do"
  name="loginForm"
  type="com.wiley.loginForm" >

  User Id: <html:text property="username"><br/>
  Password: <html:password property="password"><br/>
  <html:submit />
</html:form>
```

As you can see, several JSP custom tags are being leveraged in this JSP. These tags are defined by the Struts Framework, and provide a loose coupling to the Controller of a Struts application. We build a working Struts View in Chapter 3; and in Chapter 5, "The Views," we examine the Struts Views in more detail.

The Controller

The Controller component of the Struts Framework is the backbone of all Struts Web applications. It is implemented using a servlet named org.apache.struts.action.ActionServlet. This servlet receives all requests from clients, and delegates control of each request to a user−defined org.apache.struts.action.Action class. The ActionServlet delegates control based on the URI of the incoming request. Once the Action class has completed its processing, it returns a key to the ActionServlet, which is then used by the ActionServlet to determine the View that will present the results of the Action's processing. The ActionServlet is similar to a factory that creates Action objects to perform the actual business logic of the application.

The Controller of the Struts Framework is the most important component of the Struts MVC.

Web Applications

All Struts applications are packaged using the Java Web application format. Therefore, before we continue, let's take a brief look at Java Web applications.

Java Web applications are best described by the Java Servlet Specification 2.2, which introduced the idea using the following terms: "A Web Application is a collection of servlets, HTML pages, classes, and other resources that can be bundled and run on multiple containers from multiple vendors." In simpler terms, a Java Web application is a collection of one or more Web components that have been packaged together for the purpose of creating a complete application to be executed in the Web layer of an enterprise application. Here is a list of the common components that can be packaged in a Web application:

- Servlets
- JavaServer Pages (JSPs)
- JSP custom tag libraries
- Utility classes and application classes
- Static documents, including HTML, images, JavaScript, etc.
- Metainformation describing the Web application

The Directory Structure

All Web applications are packed into a common directory structure, and this directory structure is the container that holds the components of a Web application. The first step in creating a Web application is to create this structure. Table 1.2 describes a sample Web application named wileyapp, and lists the contents of each of its directories. Each one of these directories will be created from the <*SERVER_ROOT*> of the Servlet/JSP container.

Table 1.2: The Web Application Directory Structure

Directory	Contains
/wileyapp	This is the root directory of the Web application. All JSP and HTML files are stored here.
/wileyapp/WEB−INF	This directory contains all resources related to the application that are not in the document root of the application. This is where your Web application deployment descriptor is located. You should note that the WEB−INF directory is not part of the public document. No files contained in this directory can be served directly to a client.
/ wileyapp/WEB−INF/classes	This directory is where servlet and utility classes are located.
/ wileyapp/WEB−INF/lib	This directory contains Java Archive (JAR) files that the Web application is dependent on.

If you're using Tomcat as your container, the default root directory is <*CATALINA_HOME*>/webapps/. Figure 1.2 shows the wileyapp as it would be hosted by a Tomcat container.

Note Web applications allow compiled classes to be stored in both the /WEB−INF/classes and /WEB−INF/lib directories. Of these two directories, the class loader will load classes from the /classes directory first,

followed by the JARs in the /lib directory. If you have duplicate classes in both the /classes and /lib directories, the classes in the /classes directory will take precedence.

The Web Application Deployment Descriptor

The backbone of all Web applications is its deployment descriptor. The Web application deployment descriptor is an XML file named web.xml that is located in the /<*SERVER_ROOT*>/*applicationname*/WEB−INF/ directory. The web.xml file describes all of the components in the Web application. If we use the previous Web application name, wileyapp, then the web.xml file would be located in the /<*SERVER_ROOT*>/wileyapp /WEB−INF/ directory. The information that can be described in the deployment descriptor includes the following elements:

- ServletContext init parameters
- Localized content
- Session configuration
- Servlet/JSP definitions
- Servlet/JSP mappings
- Tag library references
- MIME type mappings
- Welcome file list
- Error pages
- Security information

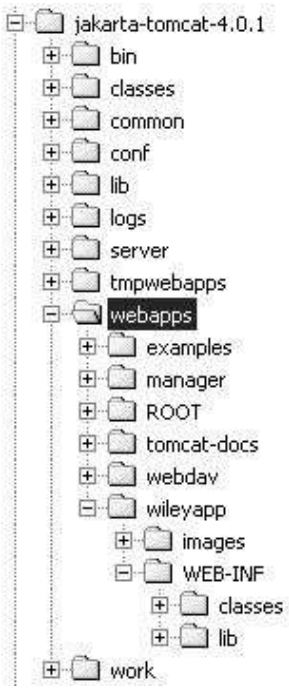


Figure 1.2: The wileyapp Web application hosted by Tomcat.

This code snippet contains a sample deployment descriptor that defines a single servlet. We examine the web.xml file in much more detail later in this text.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app PUBLIC
  '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
  'http://java.sun.com/dtd/web-app_2_3.dtd'>

<servlet>
  <servlet-name>SimpleServlet</servlet-name>
  <servlet-class>com.wiley.SimpleServlet</servlet-class>
</servlet>

</web-app>
```

Packaging a Web Application

The standard packaging format for a Web application is a Web Archive file (WAR). A WAR file is simply a JAR file with the extension .war, as opposed to .jar. You can create a WAR file by using jar, Java's archiving tool. To create a WAR file, you simply need to change to the root directory of your Web application and type the following command:

```
jar cvf wileyapp.war .
```

This command will produce an archive file named wileyapp.war that contains the entire wileyapp Web application. Now you can deploy your Web application by simply distributing this file.

The Tomcat JSP/Servlet Container

The Tomcat server is an open−source Java−based Web application container created to run servlet and JavaServer Page Web applications. It has become Sun's reference implementation for both the Servlet and JSP specifications. We will use Tomcat for all of our examples in this book.

Before we get started with the installation and configuration of Tomcat, you need to make sure you have acquired the items listed in Table 1.3.

Table 1.3: Tomcat Installation Requirements

Component	Location
Jakarta−Tomcat 4	http://jakarta.apache.org/
JDK 1.3 Standard Edition	http://java.sun.com/j2se/1.3/

Installing and Configuring Tomcat

For our purposes, we will install Tomcat as a stand−alone server on a Windows NT/2000 operating system (OS). To do this, you need to install the JDK; be sure to follow the installation instructions included with the JDK archive. For our example, we will install the JDK to drive D, which means our JAVA_HOME directory is D:\jdk1.3.

Now we need to extract the Tomcat server to a temporary directory. The default Tomcat archive does not contain an installation program; therefore, extracting the Tomcat archive is equivalent to installation. Again, we are installing to drive D, which will make the TOMCAT_HOME directory D:\ jakarta−tomcat−4.0.x.

After we have extracted Tomcat, the next step is to set the JAVA_HOME and TOMCAT_HOME environment variables. These variables are used to compile JSPs and run Tomcat, respectively. To do this under NT/2000, perform these steps:

1. Open the NT/2000 Control Panel.
2. Start the NT/2000 System Application and then select the Advanced tab**.**
3. Click the Environment Variables button. You will see a screen similar to Figure 1.3.

Figure 1.3: The Windows NT/2000 Environment Variables dialog box.

4. Click the New button in the System Variables section of the Environment Variables dialog box. Add a Variable named *JAVA_HOME*, and set its value to the location of your JDK installation. Figure 1.4 shows the settings associated with our installation.

Figure 1.4: The JAVA_HOME environment settings for our installation.

5. Your final step should be to repeat Step 4, using *CATALINA_HOME* for the variable name and the location of your Tomcat installation as the value. For my installation, I set the value to *D:\ jakarta−tomcat−4.0.1*.

That's all there is to it. You can now move on to the next section, in which we test our Tomcat installation.

Testing Your Tomcat Installation

Before continuing, we need to test the steps that we have just completed. To begin, first start the Tomcat

server by typing the following command (be sure to replace <*CATALINA_HOME*> with the location of your Tomcat installation):

```
<CATALINA_HOME>\bin\startup.bat
```

Once Tomcat has started, open your browser to the following URL:

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

You should see the default Tomcat home page, which is displayed in Figure 1.5.

Figure 1.5: The default Tomcat home page.

The next step is to verify the installation of our JDK. The best way to do this is to execute one of the JSP examples provided with the Tomcat server. To execute a sample JSP, start from the default Tomcat home page, shown in Figure 1.5, and choose JSP Examples. You should see a page similar to Figure 1.6.

Figure 1.6: The JSP Examples page.

Now choose the JSP example Snoop and click the Execute link. If everything was installed properly, you

should see a page similar to the one shown in Figure 1.7.

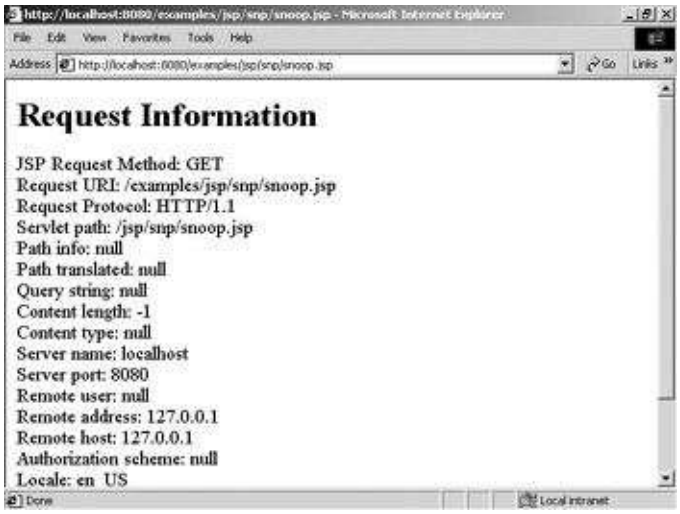


Figure 1.7: : The results of the Snoop JSP execution.

If you do not see the page shown in Figure 1.6, make sure that the location of your JAVA_HOME environment variable matches the location of your JDK installation.

Chapter 2: An Overview of the Java Servlet and JavaServer Pages Architectures

Overview

In this chapter, we discuss the two technologies that the Struts framework is based on: Java servlets and JavaServer Pages (JSPs). We begin by describing the servlet architecture, including the servlet life cycle; the relationship between the ServletContext and a Web application; and how you can retrieve form data using servlets.

Once we have a solid understanding of servlets, we move on to discussing JSPs, which act as the View component in the Struts framework. In our JSP discussions, we define JSPs and describe their components.

The goal of this chapter is to provide you with a brief introduction to the servlet and JSP technologies. At the end of this chapter, you will have a clear understanding of both servlets and JSPs, and where they fit into Java Web application development.

The Java Servlet Architecture

A *Java servlet* is a platform−independent Web application component that is hosted in a JSP/servlet container. Servlets cooperate with Web clients by means of a request/response model managed by a JSP/servlet container. Figure 2.1 depicts the execution of a Java servlet.

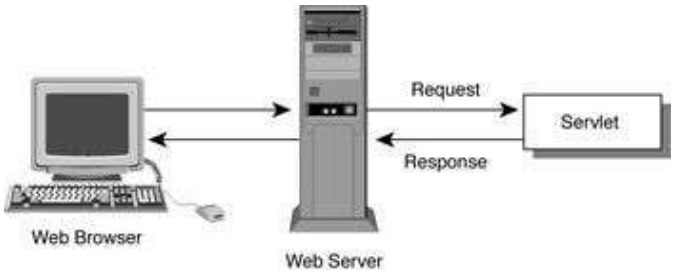


Figure 2.1: The execution of a Java servlet.

Two packages make up the servlet architecture: javax.servlet and javax. servlet.http. The first of these, the javax.servlet package, contains the generic interfaces and classes that are implemented and extended by all servlets. The second, the javax.servlet.http package, contains all servlet classes that are HTTP protocol−specific. An example of this would be a simple servlet that responds using HTML.

At the heart of this architecture is the interface javax.servlet.Servlet. It is the base class interface for all servlets. The Servlet interface defines five methods. The three most important of these methods are the

- init() method, which initializes a servlet
- service() method, which receives and responds to client requests
- destroy() method, which performs cleanup

These are the servlet life−cycle methods. We will describe these methods in a subsequent section. All servlets must implement the Servlet interface, either directly or through inheritance. Figure 2.2 is an object model that gives you a very high−level view of the servlet framework.

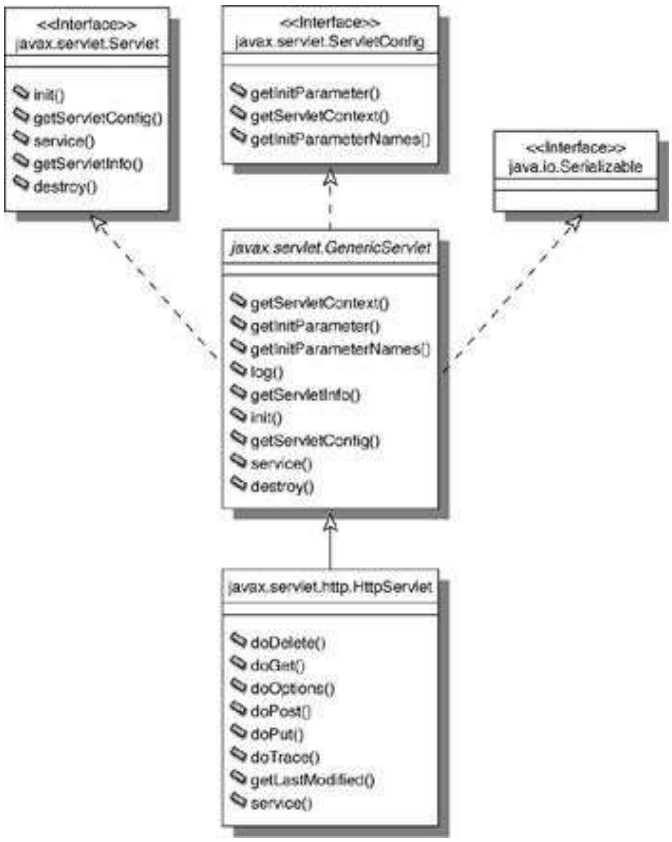


Figure 2.2: A simple object model showing the servlet framework.

The GenericServlet and HttpServlet Classes

The two main classes in the servlet architecture are the GenericServlet and HttpServlet classes. The HttpServlet class is extended from GenericServlet, which in turn implements the Servlet interface. When developing your own servlets, you will most likely extend one of these two classes.

When extending the GenericServlet class, you must implement the service() method. The GenericServlet.service() method has been defined as an abstract method in order to force you to follow this framework. The service() method prototype is defined as follows:

```
public abstract void service(ServletRequest request,
  ServletResponse response) throws ServletException, IOException;
```

The two parameters that are passed to the service() method are the ServletRequest and ServletResponse objects. The ServletRequest object holds the information that is being sent to the servlet, and the ServletResponse object is where you place the data you want to send back to the client.

In contrast to the GenericServlet, when you extend HttpServlet you don't usually implement the service() method; the HttpServlet class has already implemented the service() method for you. The following prototype contains the HttpServlet.service() method signature:

```
protected void service(HttpServletRequest request,
```

```
  HttpServletResponse  response)
  throws  ServletException,  IOException;
```

When the HttpServlet.service() method is invoked, it reads the method type stored in the request and determines which HTTP−specific methods to invoke based on this value. These are the methods that you will want to override. If the method type is GET, it will call doGet(). If the method type is POST, it will call doPost(). Five other method types are associated with the service() method, but the doGet() and doPost() methods are the methods used most often, and are therefore the methods that we are going to focus on.

The Life Cycle of a Servlet

The life cycle of a Java servlet follows a very logical sequence. The interface that declares the life−cycle methods is the javax.servlet.Servlet interface. These methods are the init(), the service(), and the destroy() methods. This sequence can be described in a simple three−step process:

1. A servlet is loaded and initialized using the init() method. This method will be called when a servlet is preloaded or upon the first request to this servlet.
2. The servlet then services zero or more requests. The servlet services each request using the service() method.
3. The servlet is then destroyed and garbage collected when the Web application containing the servlet shuts down. The method that is called upon shutdown is the destroy() method.

init() Method

The init() method is where the servlet begins its life. This method is called immediately after the servlet is instantiated. It is called only once. The init() method should be used to create and initialize the resources that it will be using while handling requests. The init() method's signature is defined as follows:

```
public  void  init(ServletConfig  config)  throws  ServletException;
```

The init() method takes a ServletConfig object as a parameter. This reference should be stored in a member variable so that it can be used later. A common way of doing this is to have the init() method call super.init() and pass it the ServletConfig object.

The init() method also declares that it can throw a ServletException. If for some reason the servlet cannot initialize the resources necessary to handle requests, it should throw a ServletException with an error message signifying the problem.

service() Method

The service() method services all requests received from a client using a simple request/response pattern. The service() method's signature is shown here:

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

The service() method takes two parameters:

- A ServletRequest object, which contains information about the service request and encapsulates information provided by the client
- A ServletResponse object, which contains the information returned to the client

You will not usually implement this method directly, unless you extend the GenericServlet abstract class. The most common implementation of the service() method is in the HttpServlet class. The HttpServlet class implements the Servlet interface by extending GenericServlet. Its service() method supports standard HTTP/1.1 requests by determining the request type and calling the appropriate method.

destroy() Method

This method signifies the end of a servlet's life. When a Web application is shut down, the servlet's destroy() method is called. This is where all resources that were created in the init() method should be cleaned up. The following code snippet contains the signature of the destroy() method:

```
public void destroy();
```

Building a Servlet

Now that we have a basic understanding of what a servlet is and how it works, we are going to build a very simple servlet of our own. Its purpose will be to service a request and respond by outputting the address of the client. After we have examined the source for this servlet, we will take a look at the steps involved in compiling and installing it. Listing 2.1 contains the source code for this example.

Listing 2.1: SimpleServlet.java. (continues)

```
package chapter2;

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

public class SimpleServlet extends HttpServlet {

  public void init(ServletConfig config)
    throws ServletException {

    // Always pass the ServletConfig object to the super class
    super.init(config);
  }

  //Process the HTTP Get request
  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    doPost(request, response);
  }

  //Process the HTTP Post request
```

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

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<html>");
    out.println("<head><title>Simple  Servlet</title></head>");
    out.println("<body>");

    // Outputs the address of the calling client
    out.println("Your address is " + request.getRemoteAddr()
      + "\n");

    out.println("</body></html>");
    out.close();
  }
}
```

Now that you have had a chance to look over the source of the SimpleServlet, let's take a closer look at each of its integral parts. We will examine where the servlet fits into the Java Servlet Development Kit (JSDK) framework, the methods that the servlet implements, and the objects being used by the servlet. The following three methods are overridden in the SimpleServlet:

- init()
- doGet()
- doPost()

Let's take a look at each of these methods in more detail.

init() Method

The SimpleServlet first defines a very straightforward implementation of the init() method. It takes the ServletConfig object that it is passed and then passes it to its parent's init() method, which stores the object for later use. The code that performs this action is as follows:

```
super.init(config);
```

Note The SimpleServlet's parent that actually holds on to the ServletConfig object is the GenericServlet. You should also notice that this implementation of the init() method does not create any resources. This is why the SimpleServlet does not implement a destroy() method.

doGet() and doPost() Methods

The SimpleServlet's doGet() and doPost() methods are where all of the business logic is truly performed, and in this case, the doGet() method simply calls the doPost() method. The only time that the doGet() method will be executed is when a GET request is sent to the container. If a POST request is received, then the doPost()

method will service the request.

Both the doGet() and the doPost() methods receive HttpServletRequest and HttpServletResponse objects as parameters. The HttpServletRequest contains information sent from the client, and the HttpServletResponse contains the information that will be sent back to the client.

The first executed line of the doPost() method sets the content type of the response that will be sent back to the client. This is done using the following code snippet:

```
response.setContentType("text/html");
```

This method sets the content type for the response. You can set this response property only once, and it must be set prior to writing to a Writer or an OutputStream. In our example, we are setting the response type to text/html.

The next thing we do is get a PrintWriter. This is accomplished by calling the ServletResponse's getWriter() method. The PrintWriter will let us write to the stream that will be sent in the client response. Everything written to the PrintWriter will be displayed in the client browser. This step is completed in the following line of code:

```
PrintWriter out = response.getWriter();
```

Once we have a reference to an object that will allow us to write text back to the client, we are going to use this object to write a message to the client. This message will include the HTML that will format this response for presentation in the client's browser. The next few lines of code show how this is done:

```
out.println("<html>");
out.println("<head><title>Simple  Servlet</title></head>");
out.println("<body>");

// Outputs the address of the calling client
out.println("Your address is " + request.getRemoteAddr()
  + "\n");
```

The SimpleServlet uses a very clear−cut method of sending HTML to a client. It simply passes to the PrintWriter's println() method the HTML text we want included in the response, and closes the stream. The only thing that you may have a question about is the following few lines:

```
// Outputs the address of the calling client
out.println("Your address is " + request.getRemoteAddr()
  + "\n");
```

This section of code takes advantage of information sent by the client. It calls the HttpServletRequest's getRemoteAddr() method, which returns the IP address of the calling client. The HttpServletRequest object holds a great deal of HTTP protocol−specific information about the client. If you would like to learn more about the HttpServletRequest or HttpServletResponse objects, you can find additional information at the Sun Web site:

```
http://java.sun.com/products/servlet/
```

Building and Deploying a Servlet

To see the SimpleServlet in action, we need to first create a Web application that will host the servlet, and then we need to compile and deploy this servlet to the Web application. These steps are described below:

1. Create a Web application named wileyapp, using the directory described in Chapter 1.
2. Add the servlet.jar file to your classpath. This file should be in the <*CATALINA_HOME*>/common/lib/ directory.
3. Compile the source for the SimpleServlet.
4. Copy the resulting class file to the <*CATALINA_HOME*>/webapps/wileyapp/WEB−INF/classes/chapter2/ directory. The /chapter2 reference is appended because of the package name.

Once you have completed these steps, you can execute the SimpleServlet and see the results. To do this, start Tomcat, and open your browser to the following URL:

```
http://localhost:8080/wileyapp/servlet/chapter2.SimpleServlet
```

You should see an image similar to Figure 2.3.

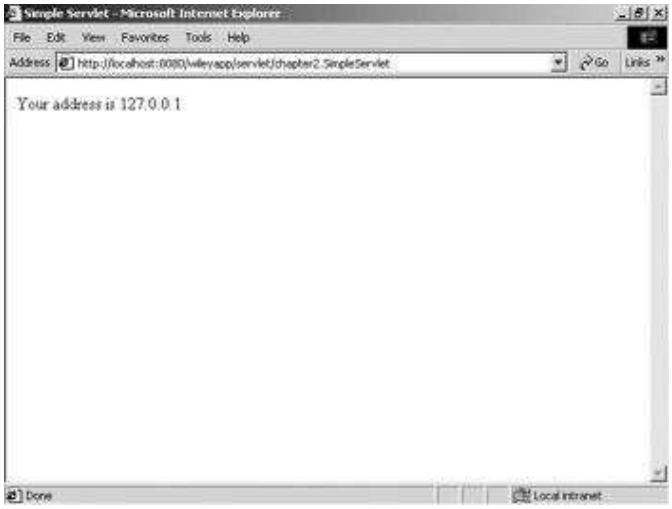


Figure 2.3: The output of the SimpleServlet.

Note You will notice that the URL to access the SimpleServlet includes the string */servlet* immediately preceding the reference to the actual servlet name. This text tells the container that you are referencing a servlet.

The ServletContext

A *ServletContext* is an object that is defined in the javax.servlet package. It defines a set of methods that are used by server−side components of a Web application to communicate with the servlet container.

The ServletContext is most frequently used as a storage area for objects that need to be available to all of the server−side components in a Web application.

You can think of the ServletContext as a shared memory segment for Web applications. When an object is

placed in the ServletContext, it exists for the life of a Web application, unless it is explicitly removed or replaced. Four methods defined by the ServletContext are leveraged to provide this shared memory functionality. Table 2.1 describes each of these methods.

Table 2.1: The Shared Memory Methods of the ServletContext

Method	Description
setAttribute()	Binds an object to a given name, and stores the object in the current ServletContext. If the name specified is already in use, this method will remove the old object binding and bind the name to the new object.
getAttribute()	Returns the object referenced by the given name, or returns null if there is no attribute bind to the given key.
removeAttribute()	Removes the attribute with the given name from the ServletContext.
getAttributeNames()	Returns an enumeration of strings containing the object names stored in the current ServletContext.

The Relationship between a Web Application and the ServletContext

The ServletContext acts as the container for a given Web application. For every Web application, there can be only one instance of a ServletContext. This relationship is required by the Java Servlet Specification, and is enforced by all servlet containers.

To see how this relationship affects Web components, we are going to use a servlet and a JSP. The first Web component we will see is a servlet that stores an object in the ServletContext, with the purpose of making this object available to all server−side components in this Web application. Listing 2.2 shows the source code for this servlet.

Listing 2.2: ContextServlet.java.

```
package chapter2;

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

public class ContextServlet extends HttpServlet {

  private static final String CONTENT_TYPE = "text/html";


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

```
    doPost(request,  response);
  }

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

    // Get a reference to the ServletContext
    ServletContext context = getServletContext();

    // Get the userName attribute from the ServletContext
    String userName = (String)context.getAttribute("USERNAME");

    // If there was no attribute USERNAME, then create
    // one and add it to the ServletContext
    if ( userName == null ) {

      userName = new String("Bob Roberts");
      context.setAttribute("USERNAME",  userName);
    }

    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Context  Servlet</title></head>");
    out.println("<body>");

    // Output the current value of the attribute USERNAME
    out.println("<p>The current User is : " + userName +
      ".</p>");
    out.println("</body></html>");
  }

  public void destroy() {
  }
}
```

As you look over the ContextServlet, you will notice that it performs the following steps:

1. It first gets a reference to the ServletContext, using the getServletContext() method:

```
ServletContext context = getServletContext();
```

9. Once it has a reference to the ServletContext, it gets a reference to the object bound to the name USERNAME from the ServletContext, using the getAttribute() method:

```
String userName =
  (String)context.getAttribute("USERNAME");
```

10. It then checks to see if the reference returned was valid. If getAttribute() returned null, then there was no object bound to the name USERNAME. If the attribute was not found, it is created and added to the ServletContext, bound to the name USERNAME, using the setAttribute() method:

```
// If there was no attribute USERNAME, then create
// one and add it to the ServletContext
if ( userName == null ) {

  userName = new String("Bob Roberts");
  context.setAttribute("USERNAME",  userName);
}
```

11. The value of this reference is then printed to the output stream, using an instance of the PrintWriter.println() method:

```
// Output the current value of the attribute USERNAME
out.println("<p>The current User is : " +
  userName + ".</p>");
```

After you have looked over this servlet, you should compile it and move the class file into the <*CATALINA_HOME*>/webapps/wileyapp/WEB−INF/classes/chapter2/ directory. This servlet is now deployed to the Web application wileyapp.

The JSP that we will be using is much like the servlet above; however, there are two differences:

- The code to access the ServletContext is in a JSP scriptlet, which we will discuss later in this chapter.
- If the JSP cannot find a reference to the USERNAME attribute, then it does not add a new one.

Otherwise, the code performs essentially the same actions, but it does them in a JSP. You can see the source for the JSP in Listing 2.3.

Listing 2.3: Context.jsp.

```
<HTML>
<HEAD>
<TITLE>
Context
</TITLE>
</HEAD>
<BODY>
<%
  // Try to get the USERNAME attribute from the ServletContext
  String userName = (String)application.getAttribute("USERNAME");

  // If there was no attribute USERNAME, then create
  // one and add it to the ServletContext
  if ( userName == null ) {

    // Don't try to add it just, say that you can't find it
    out.println("<b>Attribute USERNAME not found");
  }
  else {

    out.println("<b>The current User is : " + userName +
      "</b>");
  }
%>
</BODY>
</HTML>
```

Note In the Context.jsp, we are using two JSP implicit objects: the application object, which references the ServletContext, and the out object, which references an output stream to the client. We will discuss each of these later in this chapter.

Now, copy Context.jsp to the <*CATALINA_HOME*>/webapps/wileyapp/directory, restart Tomcat, and open your browser first to the following URL:

```
http://localhost:8080/wileyapp/Context.jsp
```

You should see a page similar to Figure 2.4.

Figure 2.4: The output of the Context.jsp prior to the execution of the servlet ContextServlet.

You should notice that the Context.jsp cannot find a reference to the attribute USERNAME. It will not be able to find this reference until the reference is placed there by the ContextServlet. To do this, open your browser to the following URL:

```
http://localhost:8080/wileyapp/servlet/chapter2.ContextServlet
```

You should see output similar to Figure 2.5.

Figure 2.5: The output of the ContextServlet.

After running this servlet, the wileyapp Web application has an object bound to the name USERNAME stored in its ServletContext. To see how this affects another Web component in the wileyapp Web application, open the previous URL that references the Context.jsp, and look at the change in output. The JSP can now find the USERNAME, and it prints this value to the response.

Note To remove an object from the ServletContext, you can restart the JSP/servlet container or use the ServletContext.removeAttribute() method.

Using Servlets to Retrieve HTTP Data

In this (our final) section on servlets, we are going to examine how servlets can be used to retrieve information from the client. Three methods can be used to retrieve request parameters: the ServletRequest's getParameter(), getParameterValues(), and getParameterNames() methods. Each method signature is listed here:

```
public String ServletRequest.getParameter(String name);
public String[] ServletRequest.getParameterValues(String name);
public Enumeration ServletRequest.getParameterNames ();
```

The first method in this list, getParameter(), returns a string containing the single value of the named parameter, or returns null if the parameter is not in the request. You should use this method only if you are sure the request contains only one value for the parameter. If the parameter has multiple values, you should use the getParameterValues() method.

The next method, getParameterValues(), returns the values of the specified parameter as an array of java.lang.Strings, or returns null if the named parameter is not in the request.

The last method, getParameterNames(), returns the parameter names contained in the request as an enumeration of strings, or an empty enumeration if there are no parameters. This method is used as a supporting method to both getParameter() and getParameterValues(). The enumerated list of parameter names returned from this method can be iterated over by calling getParameter() or getParameterValues() with each name in the list.

To see how we can use these methods to retrieve form data, let's look at a servlet that services POST requests: it retrieves the parameters sent to it and returns the parameters and their values back to the client. The servlet is shown in Listing 2.4.

Listing 2.4: ParameterServlet.java.

```
package chapter2;

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

public class ParameterServlet extends HttpServlet {

  public void init(ServletConfig config)
    throws ServletException {

    // Always pass the ServletConfig object to the super class
    super.init(config);
  }

  // Process the HTTP GET request
  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    doPost(request, response);
  }

  // Process the HTTP POST request
  public void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<html>");
    out.println("<head>");
    out.println("<title>Parameter Servlet</title>");
    out.println("</head>");
    out.println("<body>");

    // Get an enumeration of the parameter names
    Enumeration parameters = request.getParameterNames();

    String param = null;

    // Iterate over the paramater names,
    // getting the parameters values
    while ( parameters.hasMoreElements() ) {
      param = (String)parameters.nextElement();
      out.println(param + " : " +
        request.getParameter(param) +
```

```
            "<BR>");
        }

        out.println("</body></html>");
        out.close();
    }
}
```

The first notable action performed by this servlet is to get all of the parameter names passed in on the request. It does this using the getParameterNames() method. Once it has this list, it performs a while loop, retrieving and printing all of the parameter values associated with the matching parameter names, using the getParameter() method. You can invoke the ParameterServlet by encoding a URL string with parameters and values, or simply by using the HTML form found in Listing 2.5.

Listing 2.5: Form.html.

```
<HTML>
<HEAD>
<TITLE>
Parameter Servlet Form
</TITLE>
</HEAD>
<BODY>

<form
 action="servlet/chapter2.ParameterServlet"
 method=POST>
  <table width="400" border="0" cellspacing="0">
    <tr>
      <td>Name: </td>
      <td>
        <input type="text"
               name="name"
               size="20"
               maxlength="20">
      </td>
      <td>SSN:</td>
      <td>
        <input type="text" name="ssn" size="11" maxlength="11">
      </td>
    </tr>
    <tr>
      <td>Age:</td>
      <td>
        <input type="text" name="age" size="3" maxlength="3">
      </td>
      <td>email:</td>
      <td>
        <input type="text"
               name="email"
               size="30"
               maxlength="30">
      </td>
    </tr>
```

```
    <tr>
      <td> </td>
      <td>  </td>
      <td>  </td>
      <td>
        <input type="submit" name="Submit" value="Submit">
        <input type="reset" name="Reset" value="Reset">
      </td>
    </tr>
  </table>
</FORM>

</BODY>
</HTML>
```

This HTML document contains a simple HTML form that can be used to pass data to the ParameterServlet. To see this example in action, compile the servlet, and move the class file to the <*CATALINA_HOME*>/webapps/ wileyapp/WEB−INF/classes/chapter2 directory and the HTML file to the <*CATALINA_HOME*>/webapps/wileyapp/ directory. Now open your browser to the following URL:

```
http://localhost:8080/wileyapp/Form.html
```

Go ahead and populate the form (similar to what I've done in Figure 2.6), and then click the Submit button.

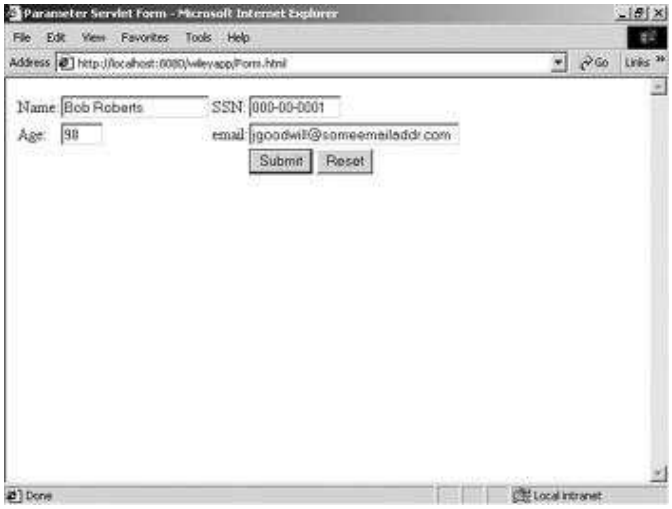


Figure 2.6: Output from Form.html.

The response you receive will, of course, depend on your entries, but it should resemble Figure 2.7.

Figure 2.7: The response of the ParameterServlet.

This example shows just how easy it is to retrieve request parameters in a servlet. While the ParameterServlet works well for most requests, it does contain an error. When we chose to use getParameter() to retrieve the parameter values, we were counting on receiving only one value per request parameter. If we could not rely on this fact, then we should have used the getParameterValues() method discussed previously.

What Are JavaServer Pages?

JavaServer Pages, or JSPs, are a simple but powerful technology used most often to generate dynamic HTML on the server side. JSPs are a direct extension of Java servlets designed to let the developer embed Java logic directly into a requested document. A JSP document must end with the extension .jsp. The following code snippet contains a simple example of a JSP file; its output is shown in Figure 2.8.

```
<HTML>
<BODY>

<% out.println("HELLO JSP READER"); %>

</BODY>
</HTML>
```

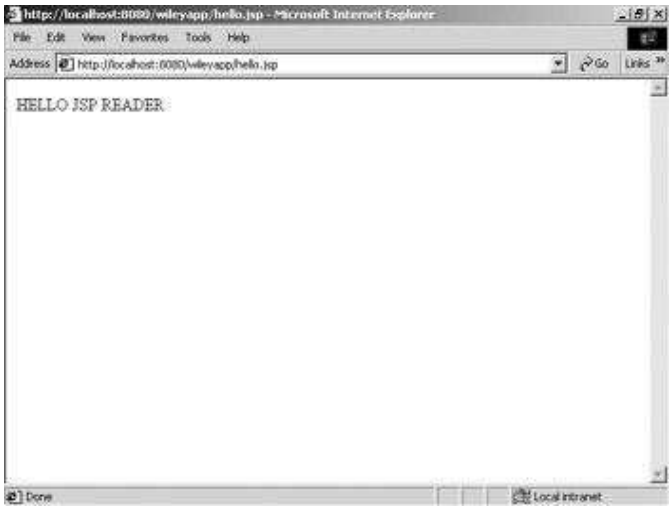


Figure 2.8: The output of the JSP example.

This document looks like any other HTML document, with some added tags containing Java code. The source code is stored in a file called hello.jsp, and should be copied to the document directory of the Web application to which this JSP will be deployed. When a request is made for this doc− ument, the server recognizes the .jsp extension and realizes that special handling is required. The JSP is then passed to the JSP engine (which is just another servlet mapped to the extension .jsp) for processing.

The first time the file is requested, it is translated into a servlet and then compiled into an object that is loaded into resident memory. The generated servlet then services the request, and the output is sent back to the requesting client. On all subsequent requests, the server will check to see whether the original JSP source file has changed. If it has not changed, the server invokes the previously compiled servlet object. If the source has changed, the JSP engine will reparse the JSP source. Figure 2.9 shows these steps.

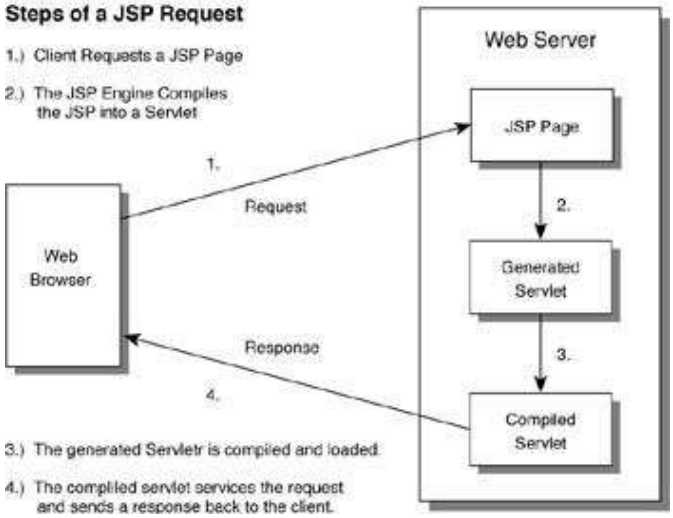


Figure 2.9: The steps of a JSP request.

Note It's essential to remember that JSPs are just servlets created from a combination of HTML and Java source. Therefore, they have the resources and functionality of a servlet.

The Components of a JavaServer Page

This section discusses the components of a JSP, including directives, scripting, implicit objects, and standard actions.

JSP Directives

JSP directives are JSP elements that provide global information about a JSP page. An example would be a directive that included a list of Java classes to be imported into a JSP. The syntax of a JSP directive follows:

```
<%@ directive {attribute="value"} %>
```

Three possible directives are currently defined by the JSP specification v1.2: page, include, and taglib. These directives are defined in the following sections.

The page Directive

The page directive defines information that will globally affect the JSP containing the directive. The syntax of a JSP page directive is

```
<%@ page {attribute="value"} %>
```

Table 2.2 defines the attributes for the page directive.

Note Because all mandatory attributes are defaulted, you are not required to specify any page directives.

Table 2.2: Attributes for the page Directive (continues)

Attribute	Definition
language=”scriptingLanguage”	Tells the server which language will be used to compile the JSP file. Java is currently the only available JSP language, but we hope there will be other language support in the not−too−distant future.
extends=”className”	Defines the parent class from which the JSP will extend. While you can extend JSP from other servlets, doing so will limit the optimizations performed by the JSP/servlet engine and is therefore not recommended.
import=”importList”	Defines the list of Java packages that will be imported into this JSP. It will be a comma−separated list of package names and fully qualified Java classes.
session=”true|false”	Determines whether the session data will be available to this page. The default is true. If your JSP is not planning on using the session, then this attribute should be set to false for better performance.

buffer=”none|size in kb”	Determines whether the output stream is buffered. The default value is 8KB.
autoFlush=”true|false”	Determines whether the output buffer will be flushed automatically, or whether it will throw an exception when the buffer is full. The default is true.
isThreadSafe=”true|false”	Tells the JSP engine that this page can service multiple requests at one time. By default, this value is true. If this attribute is set to false, the SingleThreadModel is used.
info=”text”	Represents information about the JSP page that can be accessed by invoking the page’s Servlet.getServletInfo() method.
errorPage=”error_url”	Represents the relative URL to a JSP that will handle JSP exceptions.
isErrorPage=”true|false”	States whether the JSP is an errorPage. The default is false.
contentType=”ctinfo”	Represents the MIME type and character set of the

	response sent to the client.

The following code snippet includes a page directive that imports the java.util package:

```
<%@ page import="java.util.*" %>
```

The include Directive

The include directive is used to insert text and/or code at JSP translation time. The syntax of the include directive is shown in the following code snippet:

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

The file attribute can reference a normal text HTML file or a JSP file, which will be evaluated at translation time. This resource referenced by the file attribute must be local to the Web application that contains the include directive. Here's a sample include directive:

```
<%@ include file="header.jsp" %>
```

Note Because the include directive is evaluated at translation time, this included text will be evaluated only once. Thus, if the included resource changes, these changes will not be reflected until the JSP/servlet container is restarted or the modification date of the JSP that includes that file is changed.

The taglib Directive

The taglib directive states that the including page uses a custom tag library, uniquely identified by a URI and associated with a prefix that will distinguish each set of custom tags to be used in the page.

Note If you are not familiar with JSP custom tags, you can learn what they are and how they are used in my book "Mastering JSP Custom Tags and Tag Libraries," also published by Wiley.

The syntax of the taglib directive is as follows:

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

The taglib attributes are described in Table 2.3.

Table 2.3: Attributes for the taglib Directive

Attribute	Definition
uri	A URI that uniquely names a custom tag library
prefix	The prefix string used to distinguish a custom tag instance

The following code snippet includes an example of how the taglib directive is used:

```
<%@ taglib
  uri="http://jakarta.apache.org/taglibs/random-1.0"
  prefix="rand" %>
```

JSP Scripting

Scripting is a JSP mechanism for directly embedding Java code fragments into an HTML page. Three scripting language components are involved in JSP scripting. Each component has its appropriate location in the generated servlet. This section examines these components.

Declarations

JSP declarations are used to define Java variables and methods in a JSP. A JSP declaration must be a complete declarative statement.

JSP declarations are initialized when the JSP page is first loaded. After the declarations have been initialized, they are available to other declarations, expressions, and scriptlets within the same JSP. The syntax for a JSP declaration is as follows:

```
<%! declaration %>
```

A sample variable declaration using this syntax is shown here:

```
<%! String name = new String("BOB"); %>
```

A sample method declaration using the same syntax is as follows:

```
<%! public String getName() { return name; } %>
```

To get a better understanding of declarations, let's take the previous string declaration and embed it into a JSP document. The sample document would look similar to the following code snippet:

```
<HTML>
<BODY>

<%! String name = new String("BOB"); %>

</BODY>
</HTML>
```

When this document is initially loaded, the JSP code is converted to servlet code and the name declaration is placed in the declaration section of the generated servlet. It is now available to all other JSP components in the JSP.

Note It should be noted that all JSP declarations are defined at the class level, in the servlet generated from the JSP, and will therefore be evaluated prior to all JSP expressions and scriptlet code.

Expressions

JSP expressions are JSP components whose text, upon evaluation by the container, is replaced with the resulting value of the container evaluation. JSP expressions are evaluated at request time, and the result is inserted at the expression's referenced position in the JSP file. If the resulting expression cannot be converted to a string, then a translation−time error will occur. If the conversion to a string cannot be detected during

translation, a ClassCastException will be thrown at request time.

The syntax of a JSP expression is as follows:

```
<%= expression %>
```

A code snippet containing a JSP expression is shown here:

```
Hello <B><%= getName() %></B>
```

Here is a sample JSP document containing a JSP expression:

```
<HTML>
<BODY>

<%! public String getName() { return "Bob"; } %>

Hello <B><%= getName() %></B>

</BODY>
</HTML>
```

Scriptlets

Scriptlets are the JSP components that bring all the JSP elements together. They can contain almost any coding statements that are valid for the language referenced in the language directive. They are executed at request time, and they can make use of all the JSP components. The syntax for a scriptlet is as follows:

```
<% scriptlet source %>
```

When JSP scriptlet code is converted into servlet code, it is placed into the generated servlet's service() method. The following code snippet contains a simple JSP that uses a scripting element to print the text "Hello Bob" to the requesting client:

```
<HTML>
<BODY>

<% out.println("Hello Bob"); %>

</BODY>
</HTML>
```

You should note that while JSP scriplet code can be very powerful, composing all your JSP logic using scriptlet code can make your application difficult to manage. This problem led to the creation of custom tag libraries.

JSP Error Handling

Like all development methods, JSPs need a robust mechanism for handling errors. The JSP architecture provides an error−handling solution through the use of JSPs that are written exclusively to handle JSP errors.

The errors that occur most frequently are runtime errors that can arise either in the body of the JSP page or in some other object that is called from the body of the JSP page. Request−time errors that result in an exception being thrown can be caught and handled in the body of the calling JSP, which signals the end of the error. Exceptions that are not handled in the calling JSP result in the forwarding of the client request, including the uncaught exception, to an error page specified by the offending JSP.

Creating a JSP Error Page

Creating a JSP error page is a simple process: create a basic JSP and then tell the JSP engine that the page is an error page. You do so by setting the JSP's page directive attribute, isErrorPage, to true. Listing 2.6 contains a sample error page.

Listing 2.6: Creating a JSP error page: errorpage.jsp.

```
<html>

<%@ page isErrorPage="true" %>

Error: <%= exception.getMessage() %> has been reported.

</body>
</html>
```

The first JSP−related line in this page tells the JSP compiler that this JSP is an error page. This code snippet is

```
<%@ page isErrorPage="true" %>
```

The second JSP−related section uses the implicit exception object that is part of all JSP error pages to output the error message contained in the unhandled exception that was thrown in the offending JSP.

Using a JSP Error Page

To see how an error page works, let's create a simple JSP that throws an uncaught exception. The JSP shown in Listing 2.7 uses the error page created in the previous section.

Listing 2.7: Using a JSP error page: testerror.jsp.

```
<%@ page errorPage="errorpage.jsp" %>

<%

  if ( true ) {

    // Just throw an exception
    throw new Exception("An uncaught Exception");
  }

%>
```

Notice in this listing that the first line of code sets errorPage equal to errorpage.jsp, which is the name of the

error page. To make a JSP aware of an error page, you simply need to add the errorPage attribute to the page directive and set its value equal to the location of your JSP error page. The rest of the example simply throws an exception that will not be caught. To see this example in action, copy both JSPs to the <*CATALINA_HOME*>/webapps/wileyapp/ directory, and open the testerror.jsp page in your browser. You will see a page similar to Figure 2.10.

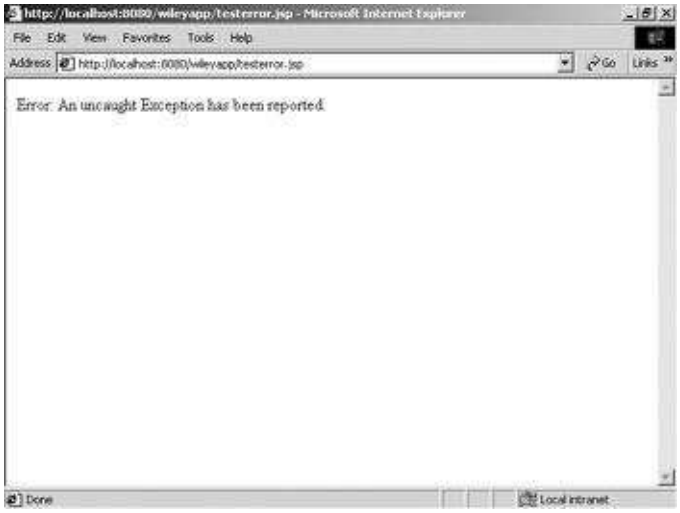


Figure 2.10: The output of the testerror.jsp example.

Implicit Objects

As a JSP author, you have implicit access to certain objects that are available for use in all JSP documents. These objects are parsed by the JSP engine and inserted into the generated servlet as if you defined them yourself.

out

The implicit out object represents a JspWriter (derived from a java.io.Writer) that provides a stream back to the requesting client. The most common method of this object is out.println(), which prints text that will be displayed in the client's browser. Listing 2.8 provides an example using the implicit out object.

Listing 2.8: Using the out object: out.jsp.

```
<%@ page errorPage="errorpage.jsp" %>

<html>
  <head>
    <title>Use Out</title>
  </head>
  <body>
    <%
      // Print a simple message using the implicit out object.
      out.println("<center><b>Hello Wiley" +
        " Reader!</b></center>");
    %>
  </body>
```

```
</html>
```

To execute this example, copy this file to the <*CATALINA_HOME*>/webapps/ wileyapp/ directory and then open your browser to the following URL:

```
http://localhost:8080/wileyapp/out.jsp
```

You should see a page similar to Figure 2.11.

Figure 2.11: The output of out.jsp.

request

The implicit request object represents the javax.servlet.http.HttpServletRequest interface, discussed later in this chapter. The request object is associated with every HTTP request.

One of the more common uses for the request object is to access request parameters. You can do this by calling the request object's getParameter() method with the parameter name you are seeking. It will return a string with the value matching the named parameter. An example using the implicit request object appears in Listing 2.9.

Listing 2.9: Using the request object: request.jsp.

```
<%@ page errorPage="errorpage.jsp" %>

<html>
  <head>
    <title>UseRequest</title>
  </head>
  <body>
    <%
      out.println("<b>Welcome: " +
        request.getParameter("user") + "</b>");
    %>
  </body>
```

```
</html>
```

This JSP calls the request.getParameter() method, passing it the parameter *user*. This method looks for the key user in the parameter list and returns the value, if it is found. Enter the following URL into your browser to see the results from this page:

```
http://localhost:8080/wileyapp/request.jsp?user=Robert
```

After loading this URL, you should see a page similar to Figure 2.12.

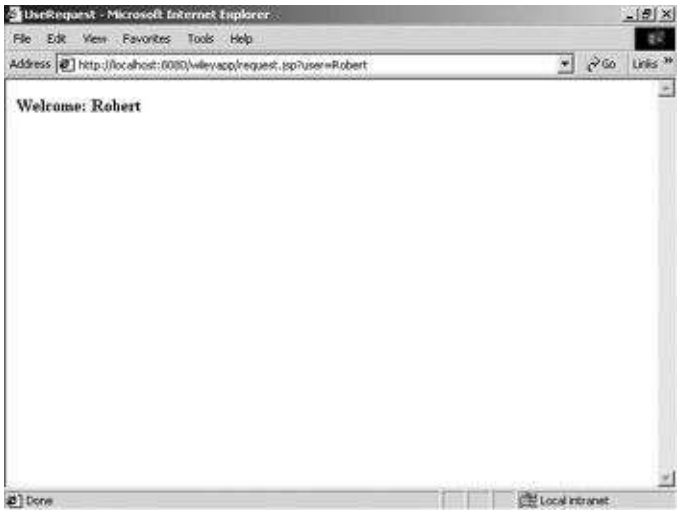


Figure 2.12: The output of request.jsp.

response

The implicit response object represents the javax.servlet.http.HttpServletResponse object. The response object is used to pass data back to the requesting client. This implicit object provides all the functionality of the HttpServletRequest, just as if you were executing in a servlet. One of the more common uses for the response object is writing HTML output back to the client browser; however, the JSP API already provides access to a stream back to the client using the implicit out object, as described in the previous implicit out discussion.

pageContext

The pageContext object provides access to the namespaces associated with a JSP page. It also provides accessors to several other JSP implicit objects. A common use for the pageContext is setting and retrieving objects using the setAttribute() and getAttribute() methods.

session

The implicit session object represents the javax.servlet.http.HttpSession object. It's used to store objects between client requests, thus providing an almost stateful HTTP interactivity.

An example of using the session object is shown in Listing 2.10.

Listing 2.10: Using the session object: session.jsp.

```
<%@ page errorPage="errorpage.jsp" %>

<html>
  <head>
    <title>Session  Example</title>
  </head>
  <body>
    <%
      // get a reference to the current count from the session
      Integer count = (Integer)session.getAttribute("COUNT");

      if ( count == null ) {

        // If the count was not found create one
        count = new Integer(1);
        // and add it to the HttpSession
        session.setAttribute("COUNT", count);
      }
      else {

        // Otherwise increment the value
        count = new Integer(count.intValue() + 1);
        session.setAttribute("COUNT", count);
      }
      out.println("<b>You have accessed this page: "
        + count + " times.</b>");
    %>
  </body>
</html>
```

To use this example, copy the JSP to the <*CATALINA_HOME*>/wileyapp/ directory, and open your browser to the following URL:

```
http://localhost:8080/wileyapp/session.jsp
```

If everything went okay, you should see a page similar to Figure 2.13.

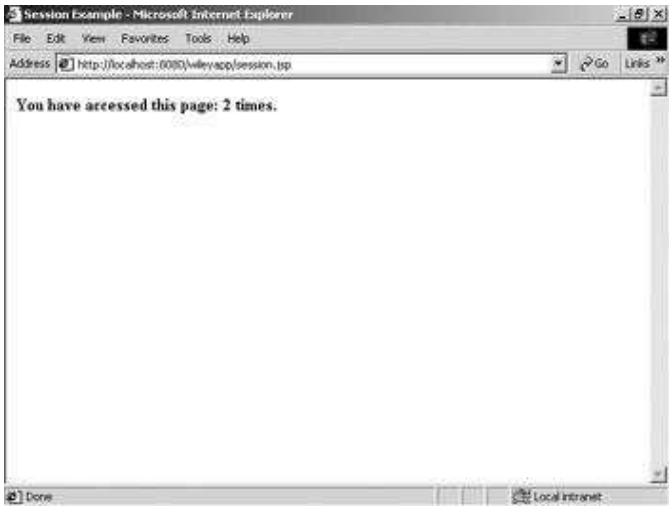


Figure 2.13: The output of session.jsp.

Click the Reload button a few times to see the count increment.

application

The application object represents the javax.servlet.ServletContext, discussed earlier in this chapter. The application object is most often used to access objects stored in the ServletContext to be shared between Web components in a global scope. It is a great place to share objects between JSPs and servlets. An example using the application object can be found earlier in this chapter, in the section "The ServletContext."

config

The implicit config object holds a reference to the ServletConfig, which contains configuration information about the JSP/servlet engine containing the Web application where this JSP resides.

page

The page object contains a reference to the current instance of the JSP being accessed. The page object is used just like this object, to reference the current instance of the generated servlet representing this JSP.

exception

The implicit exception object provides access to an uncaught exception thrown by a JSP. It is available only in JSPs that have a page with the attribute isErrorPage set to true.

Standard Actions

JSP standard actions are predefined custom tags that can be used to encapsulate common actions easily. There are two types of JSP standard actions: the first type is related to JavaBean functionality, and the second type consists of all other standard actions. Each group will be defined and used in the following sections.

Three predefined standard actions relate to using JavaBeans in a JSP: <useBean>, <setProperty>, and <getProperty>. After we define these tags, we will create a simple example that uses them.

<jsp:useBean>

The <jsp:useBean> JavaBean standard action creates or looks up an instance of a JavaBean with a given ID and scope. Table 2.4 contains the attributes of the <jsp:useBean> action, and Table 2.5 defines the scope values for that action. The <jsp:useBean> action is very flexible. When a <useBean> action is encountered, the action tries to find an existing object using the same ID and scope. If it cannot find an existing instance, it will attempt to create the object and store it in the named scope associated with the given ID. The syntax of the <jsp:useBean> action is as follows:

```
<jsp:useBean  id="name"
        scope="page|request|session|application"
        typeSpec>
        body
</jsp:useBean>

typeSpec  ::=class="className"  |
        class="className"  type="typeName"  |
        type="typeName"  class="className"  |
        beanName="beanName"  type="typeName"  |
        type="typeName"  beanName="beanName"  |
        type="typeName"
```

Table 2.4: Attributes for the <jsp:useBean> Standard Action

Attribute	Definition
id	The key associated with the instance of the object in the specified scope. This key is case−sensitive. The id attribute is the same key as used in the page.getAttribute() method.
scope	The life of the referenced object. The scope options are page, request, session, and application. They are defined in Table 2.5.

class	The fully qualified class name that defines the implementation of the object. The class name is case−sensitive.
beanName	The name of the JavaBean.
type	The type of scripting variable defined. If this attribute is unspecified, then the value is the same as the value of the class attribute.

The scope attribute listed in Table 2.4 can have four possible values, which are described in Table 2.5.

Table 2.5: Scope Values for the <jsp:useBean> Standard Action

Value	Definition
page	Beans with page scope are accessible only within the page where they were created. References to an object with page scope will be released when the current JSP has completed its evaluation.
request	Beans with request scope are accessible only within pages servicing the same request, in which the object was instantiated, including forwarded requests. All references to the object will be released after the request is complete.
session	Beans with session scope are accessible only within pages processing requests that are in the same session as the one in which the bean was created. All references to beans with session scope will be released after their associated session expires.
application	Beans with application scope are accessible within pages processing requests that are in the same Web application. All references to beans will be released when the JSP/servlet container is shut down.

<jsp:setProperty>

The <jsp:setProperty> standard action sets the value of a bean's property. Its name attribute represents an object that must already be defined and in scope. The syntax for the <jsp:setProperty> action is as follows:

```
<jsp:setProperty name="beanName" propexpr />
```

In the preceding syntax, the name attribute represents the name of the bean whose property you are setting, and propexpr can be represented by any of the following expressions:

```
property="*" |
property="propertyName" |
property="propertyName" param="parameterName" |
property="propertyName" value="propertyValue"
```

Table 2.6 contains the attributes and their descriptions for the <jsp:setProperty> action.

Table 2.6: Attributes for the <jsp:setProperty> Standard Action

Attribute	Definition
name	The name of the bean instance defined by a <jsp:useBean> action or some other action.
property	The bean property for which you want to set a value. If you set propertyName to an asterisk (*), then the action will iterate over the current ServletRequest parameters, matching parameter names and value types to property names and setter method types, and setting each matched property to the value of the matching parameter. If a parameter has an empty string for a value, the corresponding property is left unmodified.
param	The name of the request parameter whose value you want to set the named property to. A <jsp:setProperty> action cannot have both param and value attributes referenced in the same action.
value	The value assigned to the named bean's property.

<jsp:getProperty>

The last standard action that relates to integrating JavaBeans into JSPs is <jsp:getProperty>. It takes the value of the referenced bean's instance property, converts it to a java.lang.String, and places it on the output stream. The referenced bean instance must be defined and in scope before this action can be used. The syntax for the <jsp:getProperty> action is as follows:

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

Table 2.7 contains the attributes and their descriptions for the <jsp:getProperty> action.

Table 2.7: Attributes for the <jsp:getProperty> Standard Action

Attribute	Definition
name	The name of the bean instance from which the property is obtained, defined by a <jsp:useBean> action or some other action.
property	The bean property for which you want to get a value.

A JavaBean Standard Action Example

To learn how to use the JavaBean standard actions, let's create an example. This example uses a simple JavaBean that acts as a counter. The Counter bean has a single int property, count, that holds the current

number of times the bean's property has been accessed. It also contains the appropriate methods for getting and setting this property. Listing 2.11 contains the source code for the Counter bean.

Listing 2.11: Example of a Counter bean: Counter.java.

```
package chapter2;

public class Counter {

  int count = 0;
  public Counter() {

  }

  public int getCount() {

    count++;

    return count;
  }

  public void setCount(int count) {

    this.count = count;
  }
}
```

Let's look at integrating this sample JavaBean into a JSP, using the JavaBean standard actions. Listing 2.12 contains the JSP that leverages the Counter bean.

Listing 2.12: A JSP that uses the Counter bean: counter.jsp.

```
<!-- Set the scripting language to java -->
<%@ page language="java" %>

<HTML>
<HEAD>
<TITLE>Bean Example</TITLE>
</HEAD>

<BODY>

<!-- Instantiate the Counter bean with an id of "counter" -->
<jsp:useBean id="counter" scope="session"
  class="chapter2.Counter" />

<%

  // write the current value of the property count
  out.println("Count from scriptlet code : "
    + counter.getCount() + "<BR>");

%>
```

```
<!-- Get the the bean's count property, -->
<!-- using the jsp:getProperty action. -->
Count from jsp:getProperty :
  <jsp:getProperty name="counter" property="count" /><BR>

</BODY>
</HTML>
```

Counter.jsp has four JSP components. The first component tells the JSP container that the scripting language is Java:

```
<%@ page language="java" %>
```

The next step uses the standard action <jsp:useBean> to create an instance of the class Counter with a scope of session and ID of counter. Now you can reference this bean using the name counter throughout the rest of the JSP. The code snippet that creates the bean is as follows:

```
<jsp:useBean id="counter" scope="session"
  class="chapter2.Counter" />
```

The final two actions demonstrate how to get the current value of a bean's property. The first of these two actions uses a scriptlet to access the bean's property, using an explicit method call. It simply accesses the bean by its ID, counter, and calls the getCount() method. The scriptlet snippet is listed here:

```
<%

  // write the current value of the property count
  out.println("Count from scriptlet code : "
    + counter.getCount() + "<BR>");

%>
```

The second example uses the <jsp:getProperty> standard action, which requires the ID of the bean and the property to be accessed. The action takes the attribute, calls the appropriate accessor, and embeds the results directly into the resulting HTML document, as follows:

```
<!-- Get the bean's count property, -->
<!-- using the jsp:getProperty action. -->
Count from jsp:getProperty :
  <jsp:getProperty name="counter" property="count" /><BR>
```

When you execute the Counter.jsp, notice that the second reference to the count property results in a value that is one greater than the first reference. This is the case because both methods of accessing the count property result in a call to the getCount() method, which increments the value of count.

To see this JSP in action, compile the Counter class, move it to the <*CATALINA_HOME*>/wileyapp/WEB−INF/classes/chapter2/ directory, and copy the Counter.jsp file to the <*CATALINA_HOME*>/wileyapp/ directory. Then, open your browser to the following URL:

```
http://localhost:8080/wileyapp/counter.jsp
```

Once the JSP is loaded, you should see an image similar to Figure 2.14.

Figure 2.14: The results of counter.jsp.

The remaining standard actions are used for generic tasks, from basic parameter action to an object plug−in action. These actions are described in the following sections.

<jsp:param>

The <jsp:param> action provides parameters and values to the JSP standard actions <jsp:include>, <jsp:forward>, and <jsp:plugin>. The syntax of the <jsp:param> action is as follows:

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

Table 2.8 contains the attributes and their descriptions for the <jsp:param> action.

Table 2.8: Attributes for the <jsp:param> Action

Attribute	Definition
name	The name of the parameter being referenced
value	The value of the named parameter

<jsp:include>

The <jsp:include> standard action provides a method for including additional static and dynamic Web components in a JSP. The syntax for this action is as follows:

```
<jsp:include page="urlSpec" flush="true">
    <jsp:param ... />
</jsp:include>
```

Table 2.9 contains the attributes and their descriptions for the <jsp:include> action.

Table 2.9: Attributes for the <jsp:include> Action

Attribute	Definition
page	The relative URL of the resource to be included
flush	A mandatory Boolean value stating whether the buffer should be flushed

Note It is important to note the difference between the include directive and the include standard action. The directive is evaluated only once, at translation time, whereas the standard action is evaluated with every request.

The syntax description shows a request−time inclusion of a URL that is passed an optional list of param subelements used to argument the request. An example using the include standard action can be found in Listing 2.13.

Listing 2.13: Example of the include action: include.jsp.

```
<html>
  <head>
    <title>Include  Example</title>
  </head>
  <body>
    <table  width="100%"  cellspacing="0">
      <tr>
        <td  align="left">
          <jsp:include  page="header.jsp"  flush="true">
            <jsp:param  name="user"
              value='<%=  request.getParameter("user")  %>'  />
          </jsp:include>
        </td>
      </tr>
    </table>
  </body>
</html>
```

This file contains a single include action that includes the results of evaluating the JSP header.jsp, shown in Listing 2.14.

Listing 2.14: The JSP evaluated in include.jsp: header.jsp.

```
<%
  out.println("<b>Welcome:  </b>"  +
    request.getParameter("user"));
%>
```

This JSP simply looks for a parameter named user, and outputs a string containing a welcome message. To deploy this example, copy these two JSPs to the <*CATALINA_HOME*>/webapps/wileyapp/ directory. Open your browser to the following URL:

```
http://localhost:8080/wileyapp/include.jsp?user=Bob
```

The results should look similar to Figure 2.15.

Figure 2.15: The results of include.jsp.

<jsp:forward>

The <jsp:forward> standard action enables the JSP engine to execute a runtime dispatch of the current request to another resource existing in the current Web application, including static resources, servlets, or JSPs. The appearance of <jsp:forward> effectively terminates the execution of the current JSP.

Note A <jsp:forward> action can contain <jsp:param> subattributes. These subattributes act as parameters that will be forwarded to the targeted resource.

The syntax of the <jsp:forward> action is as follows:

```
<jsp:forward  page="relativeURL">
    <jsp:param  .../>
</jsp:forward>
```

Table 2.10 contains the attribute and its description for the <jsp:forward> action.

Table 2.10: Attribute for the <jsp:forward> Action

Attribute	Definition
page	The relative URL of the target of the forward

The example in Listing 2.15 contains a JSP that uses the <jsp:forward> action. This example checks a request parameter and forwards the request to one of two JSPs based on the value of the parameter.

Listing 2.15: Example of the forward action: forward.jsp.

```
<html>
  <head>
```

```
    <title>JSP Forward Example</title>
  </head>
  <body>
    <%

      if ( (request.getParameter("role")).equals("manager") ) {
        %>
          <jsp:forward page="management.jsp" />
        <%
      }
      else {
        %>
          <jsp:forward page="welcome.jsp">
          <jsp:param name="user"
            value='<%=request.getParameter("user") %>' />
          </jsp:forward>
        <%
      }
    %>
  </body>
</html>
```

The forward.jsp simply checks the request for the parameter role, and forwards the request, along with a set of request parameters, to the appropriate JSP based on this value. Listings 2.16 and 2.17 contain the source of the targeted resources.

Listing 2.16: welcome.jsp.

```
<html>
<!-- Set the scripting language to java -->
<%@ page language="java" %>

<HTML>
<HEAD>
<TITLE>Welcome Home</TITLE>
</HEAD>

<BODY>
<table>
  <tr>
    <td>
      Welcome User: <%= request.getParameter("user") %>
    </td>
  </tr>
</table>
```

Listing 2.17: management.jsp.

```
<html>
<!-- Set the scripting language to java -->
<%@ page language="java" %>

<HTML>
<HEAD>
```

```
<TITLE>Management  Console</TITLE>
</HEAD>

<BODY>
<table>
  <tr>
    <td>
      Welcome Manager: <%= request.getParameter("user") %>
    </td>
  </tr>
```

```
</table>
```

To test this example, copy all three JSPs to the <*CATALINA_HOME*>/webapps/ wileyapp/ directory and open your browser to the following URL:

```
http://localhost:8080/wileyapp/forward.jsp?role=user&user=Bob
```

You will see an image similar to Figure 2.16.

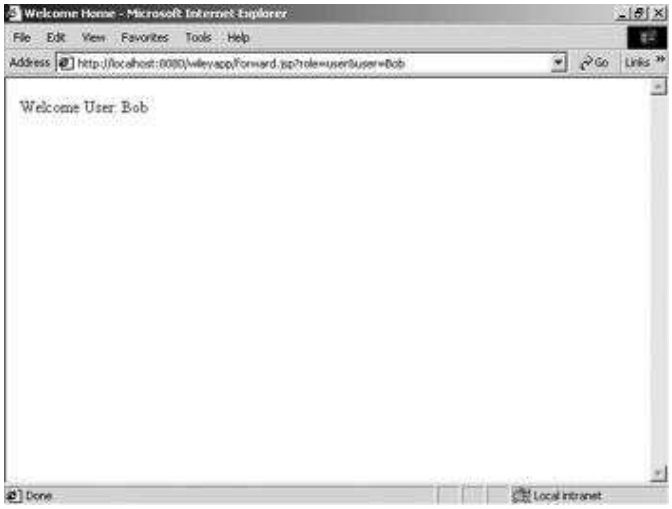


Figure 2.16: The output of forward.jsp.

You can also change the value of the role parameter to manager, to change the forwarded target.

<jsp:plugin>

The last standard action we will discuss is <jsp:plugin>. This action enables a JSP author to generate the required HTML, using the appropriate client−browser independent constructs, to result in the download and subsequent execution of the specified applet or JavaBeans component.

The <jsp:plugin> tag, once evaluated, will be replaced by either an <object> or <embed> tag, as appropriate for the requesting user agent. The attributes of the <jsp:plugin> action provide configuration data for the presentation of the embedded element. The syntax of the <jsp:plugin> action is as follows:

```
<jsp:plugin  type="pluginType"
    code="classFile"
    codebase="relativeURLpath">

    <jsp:params>

    </jsp:params>
</jsp:plugin>
```

Table 2.11 contains the attributes and their descriptions for the <jsp:plugin> action.

Table 2.11: Attributes for the <jsp:plugin> Action

Attribute	Definition

type	The type of plug−in to include (an applet, for example)
code	The name of the class that will be executed by the plug−in
codebase	The base or relative path where the code attribute can be found

The <jsp:plugin> action also supports the use of the <jsp:params> tag to supply the plug−in with parameters, if necessary.

Chapter 3: Getting Started with Struts

In this chapter, we begin our Jakarta Struts coverage. First, we explain the steps that you must perform when installing and configuring a Struts application. Then, we create a sample application that displays the components of a working Struts application. We conclude this chapter by walking through our sample application.

The goal of this chapter is to provide you with a quick introduction to the components of a Struts application.

Obtaining and Installing the Jakarta Struts Project

Before we can get started with our Struts development, we need to obtain the latest release of the Struts archive and all of its supporting archives. The following list contains all of the items you need to acquire:

- The latest−release Jakarta Struts binary for your operating system. For these examples, we are using Struts 1.1, which can be found at http://jakarta.apache.org/
- The latest Xerces Java parser. We are using Xerces 1.3, which you can find at http://xml.apache.org/

> Note For our example, we will use Tomcat 4, which comes packaged with a Xerces parser. If you choose to use another JSP/servlet container, you may need to acquire and install the latest Xerces parser.

Once you have the latest Struts release, you need to complete the following steps to prepare for the remainder of the text. You will have to complete these steps for each Struts Web application that you intend to deploy.

1. Uncompress the Struts archive to your local disk.
2. Create a new Web application, using the directory structure described in Chapter 1, “Introducing the Jakarta Struts Project and Its Supporting Components.” Make sure you substitute the name of your Web application for the value *wileyapp*. For our example, the name of our Web application is wileystruts.
3. Copy the following JAR files, extracted from the Jakarta Struts archive, to the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/lib directory:

   - struts.jar
   - commons−beanutils.jar
   - commons−collections.jar
   - commons−dbcp.jar
   - commons−digester.jar
   - commons−logging.jar
   - commons−pool.jar
   - commons−services.jar

- commons−validator.jar

4. Uncompress the Xerces archive to your local disk, if necessary.
5. Copy the xerces.jar file from the Xerces root directory to the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/lib/ directory.
6. Create an empty web.xml file, and copy it to the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/ directory. A sample web.xml file is shown in the following code snippet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
  2.3//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>

</web-app>
```

1. Create a basic strut−config.xml file, and copy it to the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/ directory. The struts−config.xml file is the deployment descriptor for Struts applications. It is the file that glues all of the MVC (Model−View−Controller) components together. Its normal location is in the <*CATALINA_HOME*>/webapps/ *webappname*/WEB−INF/ directory. We will be using this file extensively throughout the remainder of this text. An empty struts−config.xml file is listed here:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config
  PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

   <message-resources
     parameter="wiley.ApplicationResources"/>

</struts-config>
```

Note As of Struts 1.1 b1, you are required to have a <message−resources /> element defined in your struts−config.xml file. For now, you simply need to create the struts−config.xml file as shown previously. We will discuss this element's purpose in Chapter 6, "Internationalizing Your Struts Applications."

At this point, you have all of the necessary components to build the simplest of Struts applications. As you begin the design and development of your Struts application, you will need to install and configure further Struts components as necessary. In the next section, we take you through the steps that must be accomplished when developing a Struts application.

Creating Your First Struts Application

Now that you have Struts downloaded and installed, we can begin the development of our own sample Struts application. Our application consists of a simple set of JSP screens that queries a user for a stock symbol, performs a simple stock lookup, and returns the current price of the submitted stock. We will use this example

to describe the steps that must be performed when creating any Struts application.

Because Struts is modeled after the MVC design pattern, you can follow a standard development process for all of your Struts Web applications. This process begins with the identification of the application Views, the Controller objects that will service those Views, and the Model components being operated on. This process can be described using the following steps:

1. Define and create all of the Views, in relation to their purpose, that will represent the user interface of our application. Add all ActionForms used by the created Views to the struts−config.xml file.
2. Create the components of the application's Controller.
3. Define the relationships that exist between the Views and the Controllers (struts−config.xml).
4. Make the appropriate modifications to the web.xml file; describe the Struts components to the Web application.
5. Run the application.

These steps provide a high−level description of the Struts development process. In the sections that follow, we will describe each of these steps in much greater detail.

Creating the Views

When creating Views in a Struts application, you are most often creating JSPs that are a combination of JSP/HTML syntax and some conglomeration of prepackaged Struts tag libraries. The JSP/HTML syntax is similar to any other Web page and does not merit discussion, but the specialized Struts custom tag libraries do. Currently, there are three major Struts tag libraries: Bean, HTML, and Logic. We will focus on all of these libraries and more View details in Chapter 5, "The Views," but for now we will use some of the HTML tags in the Views we define in this section. For those tags that we do use, we will include a brief explanation.

To begin the development of our application, we need to first describe the Views that will represent the user interface of our application. Two Views are associated with our sample application: index.jsp and quote.jsp.

Note In our sample application, we do use a single image. This image file, hp_logo_wiley.gif, can be found in the images directory of our sample application's source tree.

The Index View

The Index View, which is represented by the file index.jsp, is our starting View. It is the first page our application users will see, and its purpose is to query the user for a stock symbol and submit the inputted symbol to the appropriate action. The source for index.jsp is found in Listing 3.1.

Listing 3.1: index.jsp.

```
<%@ page language="java" %>
<%@ taglib
  uri="/WEB-INF/struts-html.tld"
  prefix="html" %>

<html>
  <head>
    <title>Wiley Struts Application</title>
  </head>

  <body>
    <table width="500"
```

```
    border="0" cellspacing="0" cellpadding="0">
      <tr>
        <td> </td>
      </tr>
      <tr bgcolor="#36566E">
        <td height="68" width="48%">
          <div align="left">
            <img  src="images/hp_logo_wiley.gif"
             width="220"
         height="74">
          </div>
        </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
    </table>

    <html:form action="Lookup"
      name="lookupForm"
      type="wiley.LookupForm" >
      <table width="45%" border="0">
        <tr>
          <td>Symbol:</td>
          <td><html:text property="symbol" /></td>
        </tr>
        <tr>
          <td colspan="2" align="center"><html:submit /></td>
        </tr>
      </table>
    </html:form>

  </body>
</html>
```

As you look over the source for the Index View, you will notice that it looks much like any other HTML page containing a form used to gather data, with the exception of the actual form tags. Instead of using the standard HTML Form tag, like most HTML pages, the index.jsp uses a Struts−specific Form tag: <html:form />. This tag, with its child tags, encapsulates Struts form processing. The form tag attributes used in this example are described in Table 3.1.

Table 3.1: Attributes of the Form Tag Used in Our Example

Attribute	Description
action	Represents the URL to which this form will be submitted. This attribute is also used to find the appropriate ActionMapping in the Struts configuration file, which we will describe later in this section. The value used in our example is *Lookup*, which will map to an ActionMapping with a path attribute equal to Lookup.

name	Identifies the key that the ActionForm will be referenced by. We use the value *LookupForm*. An ActionForm is an object that is used by Struts to represent the form data as a JavaBean. It main purpose is to pass form data between View and Controller components. We will discuss LookupForm later in this section.
type	Names the fully qualified class name of the form bean to use in this request. For this example, we use the

	value *wiley.LookupForm*, which is an ActionForm object containing data members matching the inputs of this form.

This instance of the <html:form /> tag is also the parent to two other HTML tags. The first of the tags is the <html:text /> tag. This tag is synonymous with the HTML text input tag; the only difference is the property attribute, which names a unique data member found in the ActionForm bean class named by the form's type attribute. The named data member will be set to the text value of the corresponding input tag.

The second HTML tag that we use is the <html:submit /> tag. This tag simply emulates an HTML submit button. The net effect of these two tags is

- Upon submission, the ActionForm object named by the <html:form /> tag will be created, populated with the value of the <html:text /> tags, and stored in the session.
- Once the ActionForm object is populated with the appropriate values, the action referenced by the <html:form /> will be invoked and passed a reference to the populated ActionForm.

To use the previous two HTML tags, you must first add a taglib entry in the wileystruts application's web.xml file that references the URI /WEB−INF/struts−html.tld. This TLD describes all of the tags in the HTML tag library. The following snippet shows the <taglib> element that must be added to the web.xml file:

```
<taglib>
   <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
```

Second, you must copy the struts−html.tld from the lib directory of the extracted Struts archive to the <*CATALINA_HOME*>/webapps/wileystruts/ WEB_INF/ directory.

Note The previous two steps are used to deploy all of the Struts tag libraries. The only difference between each library's deployment is the name of the TLD. We will discuss additional Struts tag libraries in Chapter 5, "The Views."

The ActionForm

The ActionForm used in this example contains a single data member that maps directly to the symbol input parameter of the form defined in the Index View. As I stated in the previous section, when an <html:form /> is submitted, the Struts framework populates the matching data members of the ActionForm with the values entered into the <html:input /> tags. The Struts framework does this by using JavaBean reflection; therefore, the accessors of the ActionForm must follow the JavaBean standard naming convention. An example of this naming convention is shown here:

```
private String symbol;

public void setSymbol(String symbol);
public String getSymbol();
```

In this example, we have a single data member symbol. To satisfy the JavaBean standard, the accessors used to set the data member must be prefixed with *set* and *get*, followed by the data member name with its first letter capitalized. Listing 3.2 contains the source for our ActionForm.

Listing 3.2: The LookupForm implementation LookupForm.java.

```
package wiley;
```

```
import  javax.servlet.http.HttpServletRequest;
import  org.apache.struts.action.ActionForm;
import  org.apache.struts.action.ActionMapping;

public class LookupForm extends ActionForm {

  private String symbol = null;

  public String getSymbol() {

    return (symbol);
  }

  public void setSymbol(String symbol) {

    this.symbol = symbol;
  }

  public void reset(ActionMapping mapping,
    HttpServletRequest request) {

    this.symbol = null;
  }
}
```

There is really nothing special about this class. It is a simple bean that extends org.apache.struts.action.ActionForm, as must all ActionForm objects, with get and set accessors that match each of its data members. It does have one method that is specific to an ActionForm bean: the reset() method. The reset() method is called by the Struts framework with each request that uses the LookupForm. The purpose of this method is to reset all of the LookupForm's data members and allow the object to be pooled for reuse.

Note The reset() method is passed a reference to an ActionMapping class. At this point, you can ignore this class; we will fully describe it in Chapters 4 and 5.

To deploy the LookupForm to our Struts application, you need to compile this class, move it to the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/classes/wiley directory, and add the following line to the <form−beans> section of the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/struts−config.xml file:

```
<form-bean  name="lookupForm"  type="wiley.LookupForm"/>
```

This entry makes the Struts application aware of the LookupForm and how it should be referenced.

The Quote View

The last of our Views is the quote.jsp. This View is presented to the user upon successful stock symbol lookup. It is a very simple JSP with no Struts specific functionality. Listing 3.3 contains its source.

Listing 3.3: quote.jsp.

```
  <head>
    <title>Wiley  Struts  Application</title>
```

```
  </head>
  <body>

    <table width="500"
      border="0" cellspacing="0" cellpadding="0">
      <tr>
        <td> </td>
      </tr>
      <tr bgcolor="#36566E">
        <td height="68" width="48%">
          <div align="left">
            <img src="images/hp_logo_wiley.gif"
              width="220" height="74">
          </div>
        </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
      <tr>
        <td>
          Current Price : <%= request.getAttribute("PRICE") %>
        </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
    </table>
  </body>
</html>
```

As you look over this JSP, you will notice that it contains a single JSP functional line of code. This line of code retrieves the current price from the HttpServletRequest of the submitted stock symbol. This value is placed in the HttpServletRequest by the Action object that services this request, as shown in the next section.

Creating the Controller Components

In a Struts application, two components make up the Controller. These two components are the org.apache.struts.action.ActionServlet and the org.apache. struts.action.Action classes. In most Struts applications, there is one org. apache.struts.action.ActionServlet implementation and many org.apache. struts.action.Action implementations.

The org.apache.struts.action.ActionServlet is the Controller component that handles client requests and determines which org.apache.struts.action.Action will process the received request. When assembling simple applications, such as the one we are building, the default ActionServlet will satisfy your application needs, and therefore, you do not need to create a specialized org.apache.struts.action.ActionServlet implementation. When the need arises, however, it is a very simple process. For our example, we will stick with the ActionServlet as it is delivered in the Struts packages. We will cover the process of extending the org.apache.struts.action.ActionServlet in Chapter 4, “The Controller.”

The second component of a Struts Controller is the org.apache.struts. action.Action class. As opposed to the ActionServlet, the Action class must be extended for each specialized function in your application. This class is where your application's specific logic begins.

For our example, we have only one process to perform: looking up the value of the submitted stock symbol. Therefore, we are going to create a single org.apache.struts.action.Action bean named LookupAction. The source for our Action is shown in Listing 3.4. As you examine this listing, be sure to pay close attention to the execute() method.

Listing 3.4: The LookupAction bean.

```
package wiley;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class LookupAction extends Action {

  protected Double getQuote(String symbol) {

    if ( symbol.equalsIgnoreCase("SUNW") ) {

      return new Double(25.00);
    }
    return null;
  }

  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws IOException, ServletException {

    Double price = null;

    // Default target to success
    String target = new String("success");

    if ( form != null ) {

      // Use the LookupForm to get the request parameters
      LookupForm lookupForm = (LookupForm)form;

      String symbol = lookupForm.getSymbol();

      price = getQuote(symbol);
    }

    // Set the target to failure
    if ( price == null ) {
```

```
      target = new String("failure");
    }
    else {

      request.setAttribute("PRICE",  price);
    }
    // Forward to the appropriate View
    return  (mapping.findForward(target));
  }
}
```

After examining this class, you will notice that it extends the org.apache.struts.action.Action class and contains two methods: getQuote() and execute(). The getQuote() method is a simple method that will return a fixed price (if *SUNW* is the submitted symbol).

The second method is the execute() method, where the main functionality of the LookupAction is found. This is the method that must be defined by all Action class implementations. Before we can examine how the logic contained in the execute() method works, we need to examine the four parameters passed to it. These parameters are described in Table 3.2.

Table 3.2: The Parameters of the Action.execute() Method

Component	Description
ActionMapping	The ActionMapping class contains all of the deployment information for a particular Action bean. This class will be used to determine where the results of the LookupAction will be sent once its processing is complete.
ActionForm	The ActionForm represents the form inputs containing the request parameters from the View referencing this Action bean. The reference being passed to our LookupAction points to an instance of our LookupForm.
HttpServletRequest	The HttpServletRequest attribute is a reference to the current HTTP request object.
HttpServletResponse	The HttpServletResponse is a reference to the current HTTP response object.

Now that we have described the parameters passed to the execute() method, we can move on to describing the actual method body. The first notable action taken by this method is to create a String object named *target* with a value of *success*. This object will be used to determine the View that will present successful results of this action.

The next step performed by this method is to get the request parameters contained in the LookupForm. When the form was submitted, the ActionServlet used Java's reflection mechanisms to set the values stored in this object. You should note that the reference passed to the execute() method is an ActionForm that must be cast to the ActionForm implementation used by this action. The following code snippet contains the source used to access the request parameters:

```
// Use the LookupForm to get the request parameters
LookupForm lookupForm = (LookupForm)form;

String symbol = lookupForm.getSymbol();
```

Once we have references to the symbol parameters, we pass these values to the getQuote() method. This method is a simple user−defined method that will return the Double value *25.00*. If the symbol *String* contains any values other than *SUNW*, then null is returned, and we change the value of our target to *failure*. This will have the effect of changing the targeted View. If the value was not null, then we add the returned value to the request with a key of PRICE.

At this point, the value of target equals either *success* or *failure*. This value is then passed to the ActionMapping.findForward() method, which returns an ActionForward object referencing the physical View that will actually present the results of this action. The final step of the execute() method is to return the ActionForward object to the invoking ActionServlet, which will then forward the request to the referenced View for presentation. This step is completed using the following line of code:

```
return  (mapping.findForward(target));
```

To deploy the LookupAction to our Struts application, you need to compile the LookupAction class, move the class file to the <*CATALINA_HOME*>/webapps/ wileystruts/WEB−INF/classes/wiley directory, and add the following entry to the <action−mappings> section of the <*CATALINA_HOME*>/webapps/wileystruts/ WEB−INF/struts−config.xml file:

```
<action  path="/Lookup"
  type="wiley.LookupAction"
  name="lookupForm"
  input="/index.jsp">
  <forward  name="success"  path="/quote.jsp"/>
  <forward  name="failure"  path="/index.jsp"/>
</action>
```

This entry contains the data that will be stored in the ActionMapping object that is passed to the execute() method of the LookupAction. It contains all of the attributes required to use this instance of the LookupAction, including a collection of keyed <forward> subelements representing the possible Views that can present the results of the LookupAction.

Deploying Your Struts Application

Now we have all of the necessary Struts components deployed and modified. Next, we need to tell the Web application itself about our application components. To do this, we must make some simple changes to the web.xml file.

The first change we must make is to tell the Web application about our ActionServlet. This is accomplished by adding the following servlet definition to the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/web.xml file:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

This entry tells the Web application that we have a servlet named action that is implemented by the class

org.apache.struts.action.ActionServlet, which, as we stated earlier, is the default ActionServlet provided with Struts. The entry defines a single servlet initialization parameter, config, that tells the ActionServlet where to find the struts−config.xml file. It also includes a load−on−startup element that tells the JSP/servlet container that we want this servlet to be preloaded when the Web application starts. You must pre−load the ActionServlet, or your Struts Views will not load all of their necessary resources.

Once we have told the container about the ActionServlet, we need to tell it when the action should be executed. To do this, we have to add a <servlet−mapping> element to the <*CATALINA_HOME*>/webapps/wileystruts/WEB−INF/ web.xml file:

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Note You will notice in the previously listed index.jsp that our action does not include a .do at the end of the URL. We do not have to append the .do because it is automatically appended if we use the <html:form /> tag. If you do not use the <html:form /> tag, then you will need to append .do to the action's URL.

This mapping tells the Web application that whenever a request is received with .do appended to the URL, the servlet named action should service the request.

Walking through the wileystruts Web Application

At this point, you should have completed all of the steps described in the previous section and have a deployed wileystruts Web application. In this section, we will go through this sample application and discuss each of the steps performed by Struts along the way. The purpose of this section is to provide you with a walkthrough that ties together all of the previously assembled components.

To begin using this application, you need to restart Tomcat and open your Web browser to the following URL:

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

If everything went according to plan, you should see a page similar to Figure 3.1.

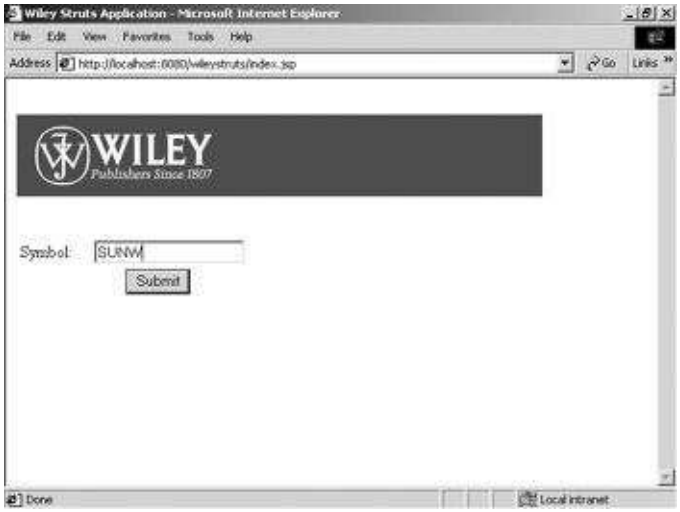


Figure 3.1: The wileystruts Index View.

When this page loads, the following actions occur:

1. The <html:form> creates the necessary HTML used to represent a form and then checks for an instance of the wiley.LookupForm in session scope. If there was an instance of the wiley.LookupForm, then the value stored in the ActionForm's data member will be mapped to the input element value on the form and the HTML form will be written to the response. This is a very handy technique that can be used to handle errors in form data. We will see examples of handling form errors in Chapter 7, "Managing Errors."
2. The Index View is then presented to the user.

To move on to the next step, enter the value *SUNW* into the Symbol text box, and click the Submit button. This will invoke the following functionality:

8. The Submit button will cause the browser to invoke the URL named in the <html:form /> tag's action attribute, which in this case is Lookup. When the JSP/servlet container receives this request, it looks in the web.xml file for a <servlet−mapping> with a <url−pattern> that ends with .do. It will find the following entry, which tells the container to send the request to a servlet that has been deployed with a <servlet−name> of action:

```
<!-- Standard Action Servlet Mapping -->
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

9. The container will find the following <servlet> entry with a <servlet−name> of action that points to the ActionServlet, which acts as the Controller for our Struts application:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
</servlet>
```


10. The ActionServlet then takes over the servicing of this request by retrieving the previously

created

LookupForm, populating its symbol data member with the value passed on the request, and adding the LookupForm to the session with a key of lookupForm.

11. At this point, the ActionServlet looks for an <ActionMapping> entry in the struts−config.xml file with a <path> element equal to Lookup. It finds the following entry:

```
<action
  path="/Lookup"
  type="wiley.Lookup
  Action"
  name="lookupForm"
  input="/index.jsp"
  >
  <forward name="success" path="/quote.jsp"/>
  <forward name="failure" path="/index.jsp"/>
</action>
```

12. It then creates an instance of the LookupAction class named by the type attribute. It also creates an ActionMapping class that contains all of the values in the <ActionMapping> element.

    Note The Struts framework does pool instances of Action classes; therefore, if the wiley.LookupAction had already been requested, then it will be retrieved from the instance pool as opposed to being created with every request.

13. It then invokes the LookupAction.execute() with the appropriate parameters. The LookupAction.execute() method performs its logic, and calls the ActionMapping.findForward() method with a String value of either *success* or *failure*.
14. The ActionMapping.findForward() method looks for a <forward> subelement with a name attribute matching the target value. It then returns an ActionForward object containing the results of the lookup, which is the value of the path attribute /quote.jsp (upon success) or /index.jsp (upon failure).
15. The LookupAction then returns the ActionForward object to the ActionServlet, which in turn forwards the request object to the targeted View for presentation. The results of a successful transaction are shown in Figure 3.2.

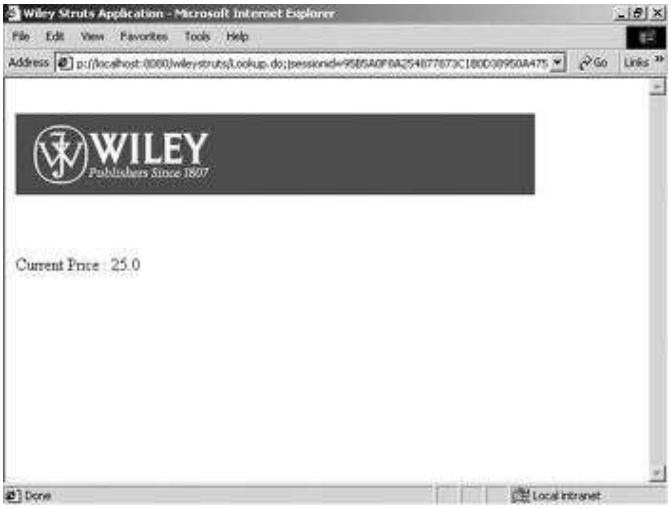


Figure 3.2: The wileystruts Quote View.

Note If you submit any value other than SUNW, you will be sent back index.jsp, which is

the failure path of the LookupAction. If this does happen, you will see that the input value on the index page is prepopulated with your originally submitted value. This is one of the handy error−handling techniques provided by the Struts application.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

Course Code : **18CAU304B** Course Title : **STRUTS FRAME WORK**

UNIT I – Multiple Choice Questions

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	Struts is a	Technology	Frame Work	Java development tool kit	Language	**Frame Work**
2	Which of the following feature is present in Struts 2?	POJO forms and POJO actions	Tag support	AJAX support	All the above	**All the above**
3	Who developed the Struts project?	Craig McClanahan	Dennis Ritchie	Dennis Ritchie	Charles	**Craig McClanahan**
4	Struts Framework is based on	Java, JSP,XML	Servelt, JSP,XML and Java	Servelet,Java, HTML,JSP	HTML,XML,Java,JSP	**Servelt, JSP,XML and Java**
5	To use Struts add____ file in our development environment	struts.war	struts.ini	struts.jar	Struts.xml	**Struts.jar**
6	____files are used for mapping between URL and action	Strut-cfg.xml	struts.xml	Struts-config.xml	web.xml	**struts.xml**
7	The limitation of creating Action Servlet instance for web applications	Two	Three	One	Four	**One**
8	____ method is necessary for ActionClass	service()	execute()	run()	destroy()	**execute()**
9	____ acts as a bridge between user invoked URI and a business method	HTTP Request	Action Class	Action Servlet	Request Processor	**Action Class**
10	Action Servlet, Request Processor and Action Classes are the components of	controller	Deployment	model	view	**Controller**
11	Return type of execute() method in Struts	void	string	int	Action Forward	**string**
12	_____ is the container that holds the components of a Web application.	Folder	Directory Structure	Files	Frame	**Directory Structure**
13	___file is used for storing JSP configuration information in struts	web.xml	struts-config.xml	struts-cfg.xml	web.cfg.xml	**web.xml**
14	The ___ method used to avoid duplicate form submissions	SaveToken	IsTokenValid()	Token()	Option 1 & 2	**Option 1 & 2**
15	___ is used to access JavaBeans and their properties	Row Library	Tag Library	Column Library	Table Library	**Tag Library**

16	____technology can be used at view layer.	J2EE	DHTML	XML/XSLT	Javascript	**XML/XSLT**
17	Method used to clear the values of a form before initiation of new request	unset	delete	submit	reset	**reset**
18	Which method is NOT used in Action Form	validate	reset	unset	set	**unset**
19	Which class is used to handle the request?	Action Servlet	Actice Forward Class	Action Class	Action Form Class	**Action Class**
20	How many xml files must have use in validate form	one	Two	three	four	**two**
21	Application module selection is done by	Action Class	Request processor	Action Servlet	All	**All**
22	The a root directory of the web application is	/wileyapp	/wileyapp/WEB−INF	/wileyapp/WEB−INF/classes	/wileyapp/WEB−INF/lib	**/wileyapp**
23	This directory is where servlet and utility classes are located.	/wileyapp	/wileyapp/WEB−INF	/wileyapp/WEB−INF/classes	/wileyapp/WEB−INF/lib	**/wileyapp/WEB−INF/classes**
24	This directory contains Java Archive (JAR) files that the Web application is dependent on.	/wileyapp	/wileyapp/WEB−INF	/wileyapp/WEB−INF/classes	/wileyapp/WEB−INF/lib	**/wileyapp/WEB−INF/lib**
25	This is where your Web application deployment descriptor is located.	/wileyapp	/wileyapp/WEB−INF	/wileyapp/WEB−INF/classes	/wileyapp/WEB−INF/lib	**/wileyapp/WEB−INF**
26	Which is the backbone of all Web applications?	Folder	Directory Structure	Deployment **Descriptor**	Frame	**Deployment Descriptor**
27	The standard packaging format for a Web application is a	WAR	JAR	JPEG	PNG	**WAR**
28	All _____ files are stored in /wileyapp.	JSP	HTML	JSP & HTML	PING	**JSP & HTML**
29	The _____file describes all of components in the Web application	Web.xml	HTML	JSP & HTML	JAR	**Web.xml**
30	The _____ performs its logic on the Model components.	Action Servlet	ActionClass	View	Actionservlet	**ActtionClass**
31	Which command is used to start the Tomcat server	\bin\startup.bat	\bin\tc\startup.bat	\bin\begin.bat	\bin\start.bat	**\bin\startup.bat**
32	The _____ method services all requests received from a client using a simple request/response pattern.	Perorm()	Service()	Destroy()	Init()	**Service()**
33	A ServletContext is an object that is defined in the package.	Java.awt	Java.util	Java.io	Javax.servlet	**Javax.servlet**
34	____are the JSP components that bring all the JSP elements together.	Servlets	Scripts	Java Script	Scriplets	**Scriplets**
35	The directive is used to insert text	include	taglib	session	info	**include**

	and/or code at JSP translation time.					
36	Which tag enables a JSP author to generate the required HTML, using the appropriate client−browser independent constructs?	<jsp:setProperty>	<jsp:forward>	<jsp.plugin>	<jsp.param>	**<jsp.plugin>**
37	Which sets the value of a bean's property?	<jsp:setProperty>	<jsp:forward>	<jsp.plugin>	<jsp.param>	**<jsp:setProperty>**
38	Which is the Controller component?	org.apache.struts.action.ActionServlet	org.apache.struts.action.Action	org.apache.struts.action.Servlet	org.apache.struts.action.Activeservlet	**org.apache.struts.action.ActionServlet**
39	All JSPs should be deployed to a Struts application by using a element.	<setProperty>	<forward>	<plugin>	<param>	**<setProperty>**
40	MVC	Model View Controller	ModernVirtual Control	Model View Controller	Module Virtual Control	**Model View Controller**
41	The _____is being manipulated and presented to the user.	Model	View	Controller	JSP	**Model**
42	_____ presents current state of the data objects	Model	View	Action Class	Controller	**View**
43	_____ defines the way the user interface reacts to the user's input.	Model	View	Action Server	Controlle	**Controller**
44	The benefits of MVC	Reliability	High reuse and adaptability	Maintability	All the above	**All the above**
	The syntax for a JSP declaration is	<%! declaration %>	<% declaration %>	<%! jsp.declaration %>	<<%! declaration !%>	**<%! declaration %>**
45	WAP allow users to access information via wireless communication.	Wireless Application Protocol	Wireless Access point	Wireless Active Point	Wireless Attach Point	**Wireless Application Protocol**
46	The backbone of the struts Framework is	Directory Structure	The Controller	The model	The View	**The Controller**
47	A collection of servlets, HTML pages, Classes and other resources called	Web Browser	Web Server	Web Applications	Website	**Web Applications**
48	WAR	Web Archive File	Web Application Resource	Web Access Resource	Both a & b	**Both a & b**
49	____ are JSP elements that provide global information about a JSP page.	JSP directives	JSP Classes	Java	Java Class	**JSP directives**
50	The ___ method is used to create and initialize the resources.	service()	init()	doget()	dopost()	**init()**
51	The method signifies the end of servlets life is____	service()	init()	destroy()	delete()	**destroy()**
52	___ is used to generate dynamic HTML on the server side.	JSP	JAVA	Servlet	Javaserver	**JSP**
53	JSP stands for	JavaServer Pages	Java Servlet Port	Jarkata Struts Page	Jarkata Struts Port	**JavaServer Pages**
54	Java Servlet is a platform____ web application component.	independent	dependent	free	oriented	**independent**

55	____ method is used to create and initialize the resources while handling requests.	init()	service()	execute()	destroy()	**init()**
56	____ elements that provide global information about a JSP Page	directives	script	objects	actions	**directives**
57	The request processor contains ___ methods	2	1	n	3	**n**
58	ServletRequest, ServletResponse are the parameters of ___ method	init()	destroy()	execute()	service()	**service()**
59	The doGet() method will be executed when ____ request is sent to the container	GET	POST	START	STOP	**GET**
60	The doPost() method will service the request when ____ received.	GET	POST	START	STOP	**POST**

Chapter 4: The Controller

In this chapter, we dig further into the Controller components of the Struts framework. We begin by looking at three distinct Struts Controller components, including the ActionServlet class, the Action class, Plugins, and the RequestProcesser.

The goal of this chapter is to provide you with a solid understanding of the Struts Controller components, and how they can be used and extended to create a robust and easily extended Web application.

The ActionServlet Class

The org.apache.struts.action.ActionServlet is the backbone of all Struts applications. It is the main Controller component that handles client requests and determines which org.apache.struts.action.Action will process each received request. It acts as an Action factory by creating specific Action classes based on the user's request.

While the ActionServlet sounds as if it might perform some extraordinary magic, it is a simple servlet. Just like any other HTTP servlet, it extends the class javax.servlet.http.HttpServlet and implements each of the HttpServlet's life−cycle methods, including the init(), doGet(), doPost(), and destroy() methods.

The special behavior begins with the ActionServlet's process() method. The process() method is the method that handles all requests. It has the following method signature:

```
protected void process(HttpServletRequest request,
  HttpServletResponse response);
```

When the ActionServlet receives a request, it completes the following steps:

1. The doPost() or doGet() methods receive a request and invoke the process() method.
2. The process() method gets the current RequestProcessor, which is discussed in the final section of this chapter, and invokes its process() method.

   Note If you intend to extend the ActionServlet, the most logical place for customization is in the RequestProcessor object. It contains the logic that the Struts controller performs with each request from the container. We will discuss the RequestProcessor in the final section of this chapter.
3. The RequestProcessor.process() method is where the current request is actually serviced. The RequestProcessor.process() method retrieves, from the struts−config.xml file, the <action> element that matches the path submitted on the request. It does this by matching the path passed in the <html:form /> tag's action element to the <action> element with the same path value. An example of this match is shown below:

```
<html:form action="/Lookup"
  name="lookupForm"
  type="wiley.LookupForm" >

<action path="/Lookup"
  type="wiley.LookupAction"
  name="lookupForm" >
  <forward name="success" path="/quote.jsp"/>
  <forward name="failure" path="/index.jsp"/>
```

```
</action>
```

4. When the RequestProcessor.process() method has a matching <action>, it looks for a <form−bean> entry that has a name attribute that matches the <action> element's name attribute. The following code snippet contains a sample match:

```
<form-beans>
  <form-bean  name="lookupForm"
    type="wiley.LookupForm"/>
</form-beans>

<action  path="/Lookup"
  type="wiley.LookupAction"
  name="lookupForm" >  <forward  name="success"  path="/quote.jsp"/>
  <forward  name="failure"  path="/index.jsp"/>
</action>
```

5. When the RequestProcessor.process() method knows the fully qualified name of the FormBean, it creates or retrieves a pooled instance of the ActionForm named by the <form−bean> element's type attribute, and populates its data members with the values submitted on the request.
6. After the ActionForm's data members are populated, the RequestProcessor.process() method calls the ActionForm.validate() method, which checks the validity of the submitted values.

   Note There is more to the validate() method than we are discussing in this chapter. We will see exactly how this method is configured and performs in Chapter 7, "Managing Errors."
7. At this point, the RequestProcessor.process() method knows all that it needs to know, and it is time to actually service the request. It does this by retrieving the fully qualified name of the Action class from the <action> element's type attribute, creating or retrieving the named class, and calling the Action.execute() method. We will look at this method in the section titled "The Action Class," later in this chapter.
8. When the Action class returns from its processing, its execute() method returns an ActionForward object that is used to determine the target of this transaction. The RequestProcessor.process() method resumes control, and the request is then forwarded to the determined target.
9. At this point, the ActionServlet instance has completed its processing for this request and is ready to service future requests.

Extending the ActionServlet

Now that you have seen what the ActionServlet is and how it is configured, let's look at how it can be extended to provide additional functionality. As you might have guessed, there are several different ways in which the ActionServlet can be extended, and we are going to examine just one of them. This examination, however, should provide the foundation you need to extend the ActionServlet for your own uses.

To develop your own ActionServlet, you must complete the following four steps. We will perform each of these steps when creating our custom ActionServlet.

1. Create a class that extends the org.apache.struts.action.ActionServlet class.
2. Implement the methods specific to your business logic.
3. Compile the new ActionServlet and move it into the Web application's classpath.
4. Add a <servlet> element to the application's web.xml file; name the new ActionServlet as the mapping to the .do extension.

In the 1.0x version of Struts, this was very common method of extending the ActionServlet. As of Struts 1.1,

it is more appropriate to extend a RequestProcessor to modify the default ActionServlet processing. We will

discuss extending these components later in this chapter.

Configuring the ActionServlet

Now that you have a solid understanding of how the ActionServlet performs its duties, let's take a look at how it is deployed and configured. The ActionServlet is like any other servlet and is configured using a web.xml <serlvet> element.

You can take many approaches when setting up an ActionServlet. You can go with a bare−bones approach, as we did in Chapter 3, "Getting Started with Struts," or you can get more serious and include any combination of the available initialization parameters described in Table 4.1.

Table 4.1: The Initialization Parameters of the ActionServlet (continues)

Parameter	Description
bufferSize	Names the size of the input buffer used when uploading files. The default value is 4096 bytes. (optional)
config	Names the context−relative path to the struts−config.xml file. The default location is in the /WEB−INF/struts−config.xml directory. (optional)
content	Names the content type and character encoding to be set on each response. The default value is *text/html*. (optional)
debug	Determines the debugging level for the ActionServlet. The default value is 0, which turns debugging off. (optional)
detail	Sets the debug level for the Digester object, which is used during ActionServlet initialization. The default value is 0. (optional)
factory	Names the fully qualified class name of the object used to create the application's MessageResources object. The default value is *org.apache.struts.util.PropertyMessageResourcesFactory*. In most cases, the default class will handle your application needs. (optional)
locale	If set to true and the requesting client has a valid session, then the Locale object is stored in the user's session bound to the key Action.LOCALE_KEY. The default value is true. (optional)
mapping	Names the fully qualified class name of the ActionMapping implementation used to describe each Action deployed to this application. The default value is *org.apache.struts.action.ActionMapping*. We will create our own ActionMapping extension in Chapter 8, "Creating Custom ActionMappings." (optional)
maxFileSize	Names the maximum file size (in bytes) of a file to be uploaded to a Struts application. This value can be expressed using *K*, *M*, or *G*, understood as kilobytes, megabytes, or gigabytes, respectively. The default size is 250M. (optional)

multipartClass	Names the fully qualified class of the MultipartRequestHandler implementation to be used when file uploads are being processed. The default value is

	org.apache.struts.upload.DiskMultipartRequestHandler. (optional)
nocache	If set to true, will add the appropriate HTTP headers to every response, turning off browser caching. This parameter is very useful when the client browser is not reflecting your application changes. The default value is false. (optional)
null	If set to true, will cause the Struts application resources to return null, as opposed to an error message, if it cannot find the requested key in the application resource bundle. The default value for this parameter is true. (optional)
tempDir	Names a directory to use as a temporary data store when file uploads are being processed. The default value is determined by the container hosting the application. (optional)
validate	If set to true, tells the ActionServlet that we are using the configuration file format defined as of Struts 1.0. The default value is true. (optional)
validating	If set to true, tells the ActionServlet that we want to validate the strut−config.xml file against its DTD. While this parameter is optional, it is highly recommended, and therefore the default is set to true.

While none of these initialization parameters are required, the most common ones include the config, application, and mapping parameters. It is also common practice to use a <load−on−startup> element to ensure that the ActionServlet is started when the container starts the Web application. An example <serlvet> entry, describing an ActionServlet, is shown in the following code snippet:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>mapping</param-name>
    <param-value>wiley.WileyActionMapping</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

We will use all of these <init−param> elements in subsequent chapters of this book.

The Action Class

The second component of a Struts Controller is the org.apache.struts.action. Action class. As we stated in Chapter 3, the Action class must and will be extended for each specialized Struts function in your application. The collection of the Action classes that belong to your Struts application is what defines your Web application.

To begin our discussion of the Action class, we must first look at some of the Action methods that are more commonly overridden or leveraged when creating an extended Action class. The following sections describe five of these methods.

The execute() Method

The execute() method is where your application logic begins. It is the method that you need to override when defining your own Actions. The Struts framework defines two execute() methods.

The first execute() implementation is used when you are defining custom Actions that are not HTTP−specific. This implementation of the execute() method would be analogous to the javax.serlvet.GenericServlet class. The signature of this execute() method is

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             ServletRequest request,
                             ServletResponse response)
        throws IOException, ServletException
```

You will notice that this method receives, as its third and fourth parameter, a ServletRequest and a ServletResponse object, as opposed to the HTTP−specific equivalents HttpServletRequest and HttpServletResponse.

The second execute() implementation is used when you are defining HTTP− specific custom Actions. This implementation of the execute() method would be analogous to the javax.serlvet.http.HttpServlet class. The signature of this execute() method is

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response)
        throws IOException, ServletException
```

You will notice that this method receives, as its third and fourth parameter, a HttpServletRequest and a HttpServletResponse object, as opposed to the previously listed execute() method. This implementation of the execute() method is the implementation that you will most often extend. Table 4.2 describes all of the parameters of the Action.execute() method.

Table 4.2: The Parameters of the Action.execute() Method

Component	Description
ActionMapping	Contains all of the deployment information for a particular Action bean. This class will be used to determine where the results of the LoginAction will be sent after its processing is complete.
ActionForm	Represents the Form inputs containing the request parameters from the View referencing this Action bean. The reference being passed to our LoginAction points to an instance of our LoginForm.

The execute() Method

HttpServletRequest	Is a reference to the current HTTP request object.

HttpServletResponse	Is a reference to the current HTTP response object.

Extending the Action Class

Now that you have seen the Action class and some of its configuration options, let's see how we can create our own Action class.

To develop your own Action class, you must complete the following steps. These steps describe the minimum actions that must be completed when creating a new Action:

1. Create a class that extends the org.apache.struts.action.Action class.
2. Implement the appropriate execute() method and your specific to your business logic.
3. Compile the new Action and move it into the Web application's classpath.
4. Add an <action> element to the application's struts−config.xml file describing the new Action.

An example execute() implementation is listed in the following snippet. We will be extending the Action class throughout the remainder of this text.

```
public ActionForward execute(ActionMapping mapping,
  ActionForm form,
  HttpServletRequest request,
  HttpServletResponse response)
  throws IOException, ServletException {

  Double price = null;

  // Default target to success
  String target = new String("success");

  if ( form != null ) {

    // Use the LookupForm to get the request parameters
    LookupForm lookupForm = (LookupForm)form;

    String symbol = lookupForm.getSymbol();

    price = getQuote(symbol);
  }

  // Set the target to failure
  if ( price == null ) {

    target = new String("failure");
  }
  else {

    request.setAttribute("PRICE", price);
  }
  // Forward to the appropriate View
  return (mapping.findForward(target));
}
```

Configuring the Action Class

Now that you have seen the major methods of the Action class, let's examine its configuration options. The

Extending the Action Class

Action class is a Struts−specific object, and therefore must be configured using the struts−config.xml file.

The element that is used to configure a Struts action is an <action> element. The class that defines the <action> element's attributes is the org.apache. struts.action.ActionMappings class. We will look at how this class can be extended to define additional <action> attributes in Chapter 8, "Creating Custom ActionMappings." Table 4.3 describes the attributes of an <action> element as they are defined by the default ActionMappings class.

Note When using an <action> element to describe an Action class, you are describing only one instance of the named Action class. There is nothing stopping you from using n−number of <action> elements that describe the same Action class. The only restriction is that the path attribute must be unique for each <action> element.

Table 4.3: Attributes of an <action> Element

Attribute	Description
path	Represents the context−relative path of the submitted request. The path must be unique and start with a / character. (required)
type	Names the fully qualified class name of the Action class being described by this ActionMapping. The type attribute is valid only if no include or forward attribute is specified. (optional)
name	Identifies the name of the form bean, if any, that is coupled with the Action being defined. (optional)
scope	Names the scope of the form bean that is bound to the described Action. The default value is session. (optional)
input	Names the context−relative path of the input form to which control should be returned if a validation error is encountered. The input attribute is where control will be returned if ActionErrors are returned from the ActionForm or Action objects. (optional)
className	Names the fully qualified class name of the ActionMapping implementation class to use in when invoking this Action class. If the className attribute is not included, the ActionMapping defined in the ActionServlet's mapping initialization parameter is used. (optional)
forward	Represents the context−relative path of the servlet or JSP resource that will process this request. This attribute is used if you do not want an Action to service the request to this path. The forward attribute is valid only if no include or type attribute is specified. (optional)

include	Represents the context−relative path of the servlet or JSP resource that will process this request. This attribute is used if you do not want an Action to service the request to this path. The include attribute is valid only if no forward or type attribute is

	specified. (optional)
validate	If set to true, causes the ActionForm.validate() method to be called on the form bean associated to the Action being described. If the validate attribute is set to false, then the ActionForm.validate() method is not called. The default value is true. (optional)

A sample <action> subelement using some of the previous attributes is shown here:

```
<action-mappings>

  <action path="/Lookup"
    type="wiley.LookupAction"
    name="lookupForm"
    input="/index.jsp">
    <forward name="success" path="/quote.jsp"/>
    <forward name="failure" path="/index.jsp"/>
  </action>

</action-mappings>
```

This <action> element tells the ActionServlet the following things about this Action instance:

- The Action class is implemented by the wiley.LookupAction class.
- This Action should be invoked when the URL ends with the path /Lookup.
- This Action class will use the <form−bean> with the name lookupForm.
- The originating resource that submitted the request to this Action is the JSP index.jsp.
- This Action class will forward the results of its processing to either the quote.jsp or the index.jsp.

The previous <action> element uses only a subset of the possible <action> element attributes, but the attributes that it does use are some of the more common.

Struts Plugins

Struts Plugins are modular extensions to the Struts Controller. They have been introduced in Struts 1.1, and are defined by the org.apache.struts.action.Plugin interface. Struts Plugins are useful when allocating resources or preparing connections to databases or even JNDI resources. We will look at an example of loading application properties on startup later in this section.

This interface, like the Java Servlet architecture, defines two methods that must be implemented by all used−defined Plugins: init() and destroy(). These are the life−cycle methods of a Struts Plugin.

init()

The init() method of a Struts Plugin is called whenever the JSP/Servlet container starts the Struts Web application containing the Plugin. It has a method signature as follows:

```
public void init(ApplicationConfig config)
  throws ServletException;
```

This method is convenient when initializing resources that are important to their hosting applications. As you will have noticed, the init() method receives an ApplicationConfig parameter when invoked. This object provides access to the configuration information describing a Struts application. The init() method marks the beginning of a Plugin's life.

destroy()

The destroy() method of a Struts Plugin is called whenever the JSP/Servlet container stops the Struts Web application containing the Plugin. It has a method signature as follows:

```
public void destroy();
```

This method is convenient when reclaiming or closing resources that were allocated in the Plugin.init() method. This method marks the end of a Plugin's life.

Creating a Plugin

Now that we have discussed what a Plugin is, let's look at an example Plugin implementation. As we stated earlier, all Plugins must implement the two Plugin methods init() and destroy(). To develop your own Plugin, you must complete the following steps. These steps describe the minimum actions that must be completed when creating a new Plugin:

1. Create a class that implements the org.apache.struts.action.Plugin interface.
2. Add a default empty constructor to the Plugin implementation. **You must have a default constructor to ensure that your Plugin is properly created by the ActionServlet.**
3. Implement both the init() and destroy() methods and your implementation.
4. Compile the new Plugin and move it into the Web application's classpath.
5. Add an <plug−in> element to the application's struts−config.xml file describing the new Plugin. We will look at this step in the next section.

An example Plugin implementation is listed in the following snippet.

```
package wiley;

import java.util.Properties;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.ServletContext;

import org.apache.struts.action.PlugIn;
import org.apache.struts.config.ApplicationConfig;
import org.apache.struts.action.ActionServlet;

public class WileyPlugin implements PlugIn {

  public static final String PROPERTIES = "PROPERTIES";

  public WileyPlugin() {

  }
```

```
  public void init(ActionServlet servlet,
    ApplicationConfig applicationConfig)
    throws javax.servlet.ServletException {

    System.err.println("---->The Plugin is starting<----");
    Properties properties = new Properties();

    try {

      // Build a file object referening the properties file
      // to be loaded
      File file =
        new File("PATH TO PROPERTIES FILE");

      // Create an input stream
      FileInputStream fis =
        new FileInputStream(file);

      // load the properties
      properties.load(fis);

      // Get a reference to the ServletContext
      ServletContext context =
        servlet.getServletContext();

      // Add the loaded properties to the ServletContext
      // for retrieval throughout the rest of the Application
      context.setAttribute(PROPERTIES, properties);
    }
    catch (FileNotFoundException fnfe) {

      throw new ServletException(fnfe.getMessage());
    }
    catch (IOException ioe) {

      throw new ServletException(ioe.getMessage());
    }
  }

  public void destroy() {

    // We don't have anything to clean up, so
    // just log the fact that the Plugin is shutting down
    System.err.println("---->The Plugin is stopping<----");
  }
}
```

As you look over our example Plugin, you will see just how straightforward Plugin development can be. In this example, we create a simple Plugin that extends the init() method, which contains the property loading logic, and the destroy() method, which contains no specialized implementation. The purpose of this Plugin is to make a set of properties available upon application startup. To make the wiley.WileyPlugin available to your Struts application, you need to move on to the following section on Plugin configuration.

Configuring a Plugin

Now that you have seen a Plugin and understand how they can be used, let’s take a look at how a Plugin is deployed and configured. To deploy and configure our wiley.WileyPlugin, you must

1. Compile and move the Plugin class file into the classpath.
2. Add a <plug−in> element to your struts−config.xml file. An example <plug−in> entry, describing the previously defined Plugin, is shown in the following code snippet:

```
<plug-in  className="wiley.WileyPlugin"/>
```

Note The <plug−in> element must follow all <message−resources /> elements in the struts−config.xml.

3. Restart the Struts Web application.

When this deployment is complete, this Plugin will begin its life when the hosting application restarts.

The RequestProcessor

As we stated previously, the org.apache.struts.action.RequestProcessor contains the logic that the Struts controller performs with each servlet request from the container. The RequestProcessor is the class that you will want to override when you want to customize the processing of the ActionServlet.

Creating a New RequestProcessor

Now that we have discussed what the RequestProcessor is, let's look at an example Plugin implementation. The RequestProcessor contains n−number of methods that you can override to change the behavior of the ActionServlet.

To create your own RequestProcessor, you must follow the steps described in the following list:

1. Create a class that extends the org.apache.struts.action.RequestProcessor class.
2. Add a default empty constructor to the RequestProcessor implementation.
3. Implement the method that you want to override. Our example overrides the processPreprocess() method.

In our example, we are going to override one of the more useful RequestProcessor methods, the processPreprocess() method, to log information about every request being made to our application.

The processPreprocess() method is executed prior to the execution of every Action.execute() method. It allows you to perform application−specific business logic before every Action. The method prototype for the processPreprocess() method is shown below:

```
protected boolean processPreprocess(HttpServletRequest request,
  HttpServletResponse response)
```

The default implementation of the processPreprocess() method simply returns true, which tells the framework to continue its normal processing. You must return true from your overridden processPreprocess() method if you want to continue processing the request.

> Note If you do choose to return false from the processPreprocess() method, then the RequestProcessor will stop processing the request and return control back to the doGet() or doPost() of the ActionServlet.

To see how all of this really works, take a look at our example RequestProcessor implementation, which is listed in the following snippet.

```
package wiley;

import  javax.servlet.http.HttpServletRequest;
import  javax.servlet.http.HttpServletResponse;
import  javax.servlet.http.HttpServlet;
import  javax.servlet.ServletException;
import  javax.servlet.http.Cookie;

import  java.io.IOException;
import  java.util.Enumeration;

import  org.apache.struts.action.RequestProcessor;

public class WileyRequestProcessor extends RequestProcessor {

  public WileyRequestProcessor() {
  }

  public boolean processPreprocess(HttpServletRequest request,
    HttpServletResponse response) {

    log("----------processPreprocess  Logging--------------");
    log("Request URI = " + request.getRequestURI());
    log("Context Path = " + request.getContextPath());

    Cookie cookies[] = request.getCookies();
    if (cookies != null) {

      for (int i = 0; i < cookies.length; i++) {

        log("Cookie = " + cookies[i].getName() + " = " +
        cookies[i].getValue());
      }
    }

    Enumeration headerNames = request.getHeaderNames();

    while (headerNames.hasMoreElements()) {

      String headerName =
        (String)  headerNames.nextElement();

      Enumeration headerValues =
        request.getHeaders(headerName);

      while (headerValues.hasMoreElements()) {

        String headerValue =
          (String)  headerValues.nextElement();

        log("Header = " + headerName + " = " + headerValue);
      }
    }
    log("Locale = " + request.getLocale());
    log("Method = " + request.getMethod());
    log("Path Info = " + request.getPathInfo());
    log("Protocol = " + request.getProtocol());
    log("Remote Address = " + request.getRemoteAddr());
```

```
    log("Remote Host = " + request.getRemoteHost());
    log("Remote User = " + request.getRemoteUser());
    log("Requested Session Id = "
      +  request.getRequestedSessionId());
    log("Scheme = " + request.getScheme());
    log("Server Name = " + request.getServerName());
    log("Server Port = " + request.getServerPort());
    log("Servlet Path = " + request.getServletPath());
    log("Secure = " + request.isSecure());
    log("-----------------------------------------------");

    return true;
  }
}
```

In our processPreprocess() method, we are retrieving the information stored in the request and logging it to the ServletContext log. Once the logging is complete, the processPreprocess() method returns the Boolean value true, and normal processing continues. If the processPreprocess() method had returned false, then the ActionServlet would have terminated processing, and the Action would never have been performed.

Configuring an Extended RequestProcessor

Now that you have seen a Plugin and understand how it can be used, let's take a look at how a Plugin is deployed and configured. To deploy and configure our wiley.WileyPlugin, you must

1. Compile the new RequestProcessor and move it into the Web application's classpath.
2. Add a <controller> element to the application's struts−config.xml file describing the new RequestProcessor. An example <controller> entry, describing the our new RequestProcessor, is shown in the following code snippet:

   ```
   <controller
     processorClass="wiley.WileyRequestProcessor"  />
   ```

   Note The <controller> element must follow the <action−mappings> element and precede the <message−resources /> elements in the struts−config.xml. A full description of the <controller> element and its attributes is included in Chapter 12, "The struts−config.xml File."

3. Restart the Struts Web application.

When this deployment is complete, the new RequestProcessor will take effect. To see the results of these log statements, open the *<CATALINA_HOME>*/ logs/localhost_log.*todaysdate*.txt file, and you will see the logged request at the bottom of the log file.

Summary

In this chapter, we described the different Controller components, and discussed how and when they should be extended. In the next chapter, we will discuss the presentation layer of the Struts framework. We will describe the major components of the Struts View, including ActionForm beans and the Struts tag libraries, and how each of these components fit into the Struts framework.

Chapter 5: The Views

In this chapter, we examine the View component of the Struts framework. Some of the topics that we discuss are using tags from Struts tag libraries, using ActionForms, and deploying Views to a Struts application.

The goal of this chapter is to give you an understanding of the Struts View and the components that can be leveraged to construct the View.

Building a Struts View

As we discussed in Chapter 1, "Introducing the Jakarta Struts Project and Its Supporting Components," the Struts View is represented by a combination of JSPs, custom tag libraries, and optional ActionForm objects. In the sections that follow, we examine each of these components and how they can be leveraged.

At this point, you should have a pretty good understanding of what JSPs are and how they can be used. We can now focus on how JSPs are leveraged in a Struts application.

JSPs in the Struts framework serve two main functions. The first of these functions is to act as the presentation layer of a previously executed Controller Action. This is most often accomplished using a set of custom tags that are focused around iterating and retrieving data forwarded to the target JSP by the Controller Action. This type of View is not Struts−specific, and does not warrant special attention.

The second of these functions, which is very much Struts−specific, is to gather data that is required to perform a particular Controller Action. This is done most often with a combination of tag libraries and ActionForm objects. This type of View contains several Struts−specific tags and classes, and is therefore the focus of this chapter.

Deploying JSPs to a Struts Application

Before we can begin looking at the role of a JSP in the Struts framework, we must take a look at how JSPs are deployed to the framework. JSPs are most often the target of a previous request; whether they are gathering or presenting data usually makes no difference as to how they are deployed. All JSPs should be deployed to a Struts application by using a <forward> element. This element is used to define the targets of Struts Actions, as shown in the following code snippet:

```
<forward name="login" path="/login.jsp"/>
```

In this example, the <forward> element defines a View named login with a path of /login.jsp.

To make this <forward> element available to a Struts application, we must nest it within one of two possible Struts elements. You can make a JSP available globally to the entire application. This type of JSP deployment is useful for error pages and login pages. You perform this type of deployment by adding the JSP to the <global−forwards> section of the struts−config.xml file. An example of this is shown in the following code snippet:

```
<global-forwards>
  <forward name="login" path="/login.jsp"/>
</global-forwards>
```

The previous <forward> element states that /login.jsp will be the target of all Struts Actions that return an ActionForward instance with the name login, as shown here:

```
return  (mapping.findForward("login"));
```

Note The only time that a global forward is not used is when an <action> element has a <forward> declaration with the same name. In this instance, the <action> element's <forward> will take precedence.

The second type of <forward> declaration is defined as an Action <forward>. These types of <forward> elements are defined as subelements of an <action> definition, and are accessible only from within that <action>. The following code snippet shows an example of this type of <forward> declaration:

```
<action  path="/Login"
  type="com.wiley.LoginAction"
  validate="true"
  input="/login.jsp"
  name="loginForm"
  scope="request" >
  <forward  name="success"  path="/employeeList.jsp"/>
  <forward  name="failure"  path="/login.jsp"/>
</action>
```

This <forward> definition states that /login.jsp will be the target of com.wiley.LoginAction when this Action returns an ActionForward instance with the name "failure", as shown here:

```
return  (mapping.findForward("failure"));
```

JSPs that Gather Data

Now that you know how JSPs are deployed in a Struts application, let's take a look at one of the two most common uses of a JSP in a Struts application: using JSPs to gather data.

There are several methods that can be leveraged when gathering data using a JSP. The most common of these methods includes using the HTML <form> element and any combination of <input> subelements to build a form. While Struts uses this exact methodology, it does so with a set of JSP custom tags that emulate the HTML <form> and <input> elements, but also includes special Struts functionality. The following code snippet contains a JSP that uses the Struts tags to gather data:

```
<html:form  action="/Login"
  name="loginForm"
  type="com.wiley.LoginForm"  >

  <table  width="45%"  border="0">
    <tr>
      <td>Username:</td>
      <td><html:text  property="username"  /></td>
    </tr>
    <tr>
      <td>Password:</td>
      <td><html:password  property="password"  /></td>
    </tr>
    <tr>
      <td  colspan="2"  align="center"><html:submit  /></td>
    </tr>
  </table>
</html:form>
```

If we break this JSP into logical sections, you will first take notice of the four Struts HTML tags: <html:form />, <html:text />, <html:password />, and <html:submit />. These tags include special Struts functionality that is used to gather HTML form data. We look at each of these tags in the sections that follow.

Note The Struts library that includes the HTML tags listed in our JSP is named the HTML Tag Library. It includes tags that closely mimic the same functionality common to HTML form elements. In our example, we saw only a small fraction of the entire HTML tag library. The remaining tags are discussed in Chapter 14, "HTML Tag Library."

The <html:form /> Tag

The first of these tags is the <html:form /> tag. This tag serves as the container for all other Struts HTML input tags. It renders an HTML <form> element, containing all of the child elements associated with this HTML form. While the <html:form /> tag does serve as an HTML input container, it is also used to store and retrieve the data members of the named ActionForm bean. This tag, with its children, encapsulates the presentation layer of Struts form processing. The form tag attributes used in this example are described in Table 5.1.

Table 5.1: The Attributes of the Form Tag Used in this Example

Attribute	Description
action	Represents the URL to which this form will be submitted. This attribute is also used to find the appropriate ActionMapping in the Struts configuration file, which we describe later in this section. The value used in our example is /Login, which will map to an ActionMapping with a path attribute equal to /Login.
name	Identifies the key that the ActionForm that we will be using in this request to identify the FormBean associated with this Form. We use the value loginForm. ActionForms are described in the following section.
type	Provides the fully qualified class name of the ActionForm bean used in this request. For this example, we use the value com.wiley.LoginForm, which is described following.

ActionForm Beans

Before we can move on to examining the rest of this form, we must discuss the org.apache.struts.action.ActionForm object. ActionForms are JavaBeans that are used to encapsulate and validate the request parameters submitted by an HTTP request. A sample ActionForm, named LoginForm, is listed in the following code snippet:

```
package com.wiley;

import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionError;
```

```
public class LoginForm extends ActionForm {

  private String password = null;

  private String username = null;

  public String getPassword() {

    return (this.password);
  }

  public void setPassword(String password) {

    this.password = password;
  }

  public String getUsername() {

    return (this.username);
  }

  public void setUsername(String username) {

    this.username = username;
  }

  public void reset(ActionMapping mapping,
    HttpServletRequest request) {

    this.password = null;
    this.username = null;
  }

  public ActionErrors validate(ActionMapping mapping,
    HttpServletRequest request) {

    ActionErrors errors = new ActionErrors();

    if ( (username == null ) || (username.length() == 0) ) {

      errors.add("username",
        new  ActionError("errors.username.required"));
    }
    if ( (password == null ) || (password.length() == 0) ) {

      errors.add("password",
        new  ActionError("errors.password.required"));
    }
    return errors;
  }
}
```

As you look over this class, you should first notice that it extends the org.apache.struts.action.ActionForm class; all ActionForm beans must extend this class. After this, you will notice that the LoginForm definition itself contains two data members−−username and password−−as well as six methods.

The first four of these methods are simple setters and getters used to access and modify the two data members. Each of these setter methods is called by the Struts framework when a request is submitted with a parameter

matching the data member's name. This is accomplished using JavaBean reflection; therefore, the accessors of the ActionForm must follow the JavaBean standard naming convention. In the next section, we learn how these data members are mapped to request parameters.

The last two methods of this ActionForm are probably the most important. These methods are defined by the ActionForm object, and are used to perform request−time processing.

The reset() method is called by the Struts framework with each request that uses the defined ActionForm. The purpose of this method is to reset all of the LoginForm's data members prior to the new request values being set. You should implement this method to reset your form's data members to their original values; otherwise, the default implementation will do nothing.

As you look over our reset() method, you will note that it sets both of our data members back to null. This method guarantees that our data members are not holding stale data.

The last method defined by our LoginForm is validate(). This method should be overridden when you are interested in testing the validity of the submitted data prior to the invocation of the Action.execute() method.

The proper use of this method is to test the values of the data members, which have been set to the matching request parameters. If there are no problems with the submitted values, then the validate() method should return null or an empty ActionErrors object, and the execution of the request will continue with normal operation.

If the values of the data members are invalid, then it should create a collection of ActionErrors containing an ActionError object for each invalid parameter and then return this ActionErrors instance to the Controller. If the Controller receives a valid ActionErrors collection, it will forward the request to the path identified by the <action> element's input attribute. If your ActionForm does not implement the validate() method, then the default implementation will simply return null, and processing will continue normally.

Note We will discuss the error management process in much more detail in Chapter 7, "Managing Errors."

After looking at the LoginForm's validate() method, you will see that the request using this ActionForm must contain a username and password that is not null or a 0 length string.

The Input Tags

Once you get past the attributes of this instance of the <html:form /> tag, you will see that it also acts as a parent to three other HTML tags. These tags are synonymous with the HTML input elements.

The <html:text /> Tag

The first of the HTML input tags is the <html:text /> tag. This tag is equivalent to the HTML text input tag, with the only difference being the property attribute, which names a unique data member found in the ActionForm bean class named by the form's type attribute. The following code snippet contains our <html:text /> tag.

```
<html:text property="username" />
```

As you can see, the property attribute of this instance is set to the value *username;* therefore, when the form is submitted, the value of this input tag will be stored in the LoginForm's username data member.

The <html:password /> Tag

The second of the HTML tags is the <html:password /> tag. This tag is equivalent to the HTML password input tag. It functions in the same way as <html:text />; the only difference is that its value is not displayed to the client. The following code snippet contains our <html:password /> tag:

```
<html:password property="password" />
```

As you can see, the property attribute of this instance is set to the value *password*, which results in the LoginForm's password data member being set to the value of this input parameter.

The <html:submit />Tag

The last HTML tag that we use is the <html:submit /> tag. This tag simply emulates an HTML submit button by submitting the request to the targeted action:

```
<html:submit />
```

The Steps of a Struts Form Submission

When a View containing this type of <html:form /> is requested, it will be evaluated and the resulting HTML will look similar to this:

```
<form name="loginForm"
  method="POST"
  action="/employees/Login.do">

      <table width="45%" border="0">
        <tr>
          <td>User Name:</td>
          <td>
            <input type="text"
              name="username"
              value=""></td>
        </tr>
        <tr>
          <td>Password:</td>
          <td>
            <input type="password"
              name="password"
              value=""></td>
        </tr>
        <tr>
          <td colspan="2" align="center">
            <input type="submit"
              name="submit"
              value="Submit"></td>
        </tr>
      </table>
    </form>
```

Note As you examine the evaluated form, you will notice that the value of the <input> elements is an empty string. This will not always be the case. If the session already includes an instance of the ActionForm named by the <form> element's name attribute, then the values stored in its data members will be used to prepopulate the input values. We will see an example of this in Chapter 7.

Once the user of this form has entered the appropriate values and clicked the Submit button, the following actions take place:

1. The Controller creates or retrieves (if it already exists) an instance of the com.wiley.LoginForm object, and stores the instance in the appropriate scope. The default scope is session. To change the scope of the ActionForm, you use the <html:form /> attribute scope.
2. The Controller then calls the com.wiley.LoginForm.reset() method to set the form's data members back to their default values.
3. The Controller next populates the com.wiley.LoginForm username and password data members with the values of the <html:text /> and <html:password /> tags, respectively.
4. Once the data members of the com.wiley.LoginForm have been set, the Controller invokes the com.wiley.LoginForm.validate() method.
5. If the validate() method does not encounter problems with the data, then the Action referenced by the <html:form />'s action attribute is invoked and passes a reference to the populated ActionForm. Processing then continues normally.

That's about it. There is almost no limit to the type of Views that can exist in a Struts application, but this type of View is most tightly bound to the Struts framework. This is also the type of View that you will see evolve throughout the remainder of this text.

Summary

In this chapter, we discussed the View component of the Struts framework, and examined how to use tags from Struts tag libraries, use ActionForms, and deploy Views to a Struts application. The next chapter focuses on how to make use of the internationalization (i18n) features in Struts.

Chapter 6: Internationalizing Your Struts Applications

Overview

In this chapter, we look at the internationalization (i18n) features of the Struts Framework. We begin by defining each Struts i18n component and how it is used and configured. We then examine the steps involved when internationalizing our existing stock lookup application.

The goal of this chapter is to cover all of the required components and processes involved when internationalizing a Struts application. At the end of this chapter, you should feel comfortable with internationalizing your own Struts applications.

> Note In this chapter, you will notice that I use the terms *i18n* and *internationalization* interchangeably. While i18n looks like an acronym, we use it simply to represent "Internationalization," because 18 is the number of letters between the alphabetical characters *i* and *n* in the word *internationalization*.

I18N Components of a Struts Application

Two i18n components are packaged with the Struts Framework. The first of these components, which is managed by the application Controller, is a Message class that references a resource bundle containing Locale−dependent strings. The second i18n component is a JSP custom tag, <bean:message />, which is used in the View layer to present the actual strings managed by the Controller. We examine each of these components in the following sections.

The Controller

The standard method used when internationalizing a Struts application begins with the creation of a set of simple Java properties files. Each file contains a key/value pair for each message that you expect your application to present, in the language appropriate for the requesting client.

Defining the Resource Bundles

The first of these files is one that contains the key/value pairs for the default language of your application. The naming format for this file is *ResourceBundleName*.properties. An example of this default file, using English as the default language, would be

```
ApplicationResources.properties
```

A sample entry in this file would be

```
app.symbol=Symbol
```

This combination tells Struts that when the client has a default Locale that uses English as the language, and the key for app.symbol exists in the requested resource, then the value *Symbol* will be substituted for every occurrence of the app.symbol key.

Note In the following section, which describes the i18n View component, we see how these keys are

requested.

Once you have defined the default properties file, you must define a properties file for each language that your application will use. This file must follow the same naming convention as the default properties file, except that it must include the two−letter ISO language code of the language that it represents. An example of this naming convention for an Italian−speaking client would be

```
ApplicationResources_it.properties
```

And a sample entry in this file would be

```
app.symbol=Simbolo
```

Note You can find all of the two−letter ISO language codes at www−old.ics.uci.edu/pub/ietf/http/related/iso639.txt.

This combination tells Struts that when the client has a Locale that uses the Italian language, and the key for app.symbol exists in the requested resource, then the value *Simbolo* will be substituted for every occurrence of the app.symbol key.

Note The ApplicationResources.properties files are loaded upon application startup. If you make changes to this file, you must reload the properties file, either by restarting the entire container or by restarting the Web application referencing the properties files.

Deploying the Resource Bundles

Once you have defined all of the properties files for your application, you need to make Struts aware of them. Prior to 1.1, this was accomplished by setting the application servlet <init−parameter> of the org.apache.struts.actions.ActionServlet. As of Struts 1.1, this is achieved by adding a <message−resources> sub−element to the struts−config.xml file. The Struts 1.1 method of configuring a resource bundle will be the focus of the following section.

To make the Struts Framework aware of your application resource bundles, you must copy all of your resource bundles into the application classpath, which in this case is <*CATALINA−HOME*>/webapps/*webapplicationname*/ WEB−INF/classes/wiley, and then use the package path plus the base file name as the value of the <message−resources> subelement. The following snippet shows an example of using the <message−resources> subelement to configure a resource bundle, using the properties files described in the previous section:

```
<message-resources
  parameter="wiley.ApplicationResources"/>
```

This <message−resource> subelement tells the Struts Controller that all of our properties' files exist in the <*CATALINA_HOME*>/webapps/*web applicationname*/WEB−INF/classes/wiley directory, and are named ApplicationResources_*xx*.properties.

Note You will notice that the <param−value> contains only the default filename. This is because Struts will get the Locale of the client and append it to the filename, if it uses a language other than the default language. The behavior is the default method used when loading resource bundles.

The View

The second i18n component defined by the Struts Framework is a JSP custom tag, <bean:message />, which is used to present the actual strings that have been loaded by the Controller. This section will describe the <bean:message /> tag and how it is configured for use.

Deploying the bean Tag Library

Before we can use <bean:message />, we must first deploy the bean tag library, which contains the <bean:message /> tag. Deploying a tag library is a very simple process that requires only the addition of a new <taglib> entry in the web.xml file of the Web application using the bean library. Here is an example of this entry:

```
<taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
```

This entry simply tells the JSP/servlet container that this Web application uses a tag library, which exists in the classpath and is described by the TLD located in the <*CATALINA_HOME*>/webapps/*webappname*/WEB−INF/struts−bean.tld file. To make this a true statement, you need to copy this TLD from the Struts archive to this directory, and make sure the struts.jar file exists in the <*CATALINA_HOME*>/webapps/*webapplicationname*/WEB−INF/lib directory.

That's all there is to deploying the bean tag library. To make this change effective, you must restart Tomcat or the Web application that contains the newly deployed bean tag library.

Using the <bean:message /> Tag

The <bean:message /> tag is a useful tag that retrieves keyed values from a previously defined resource bundle—specifically, the properties files defined in the <message−resources> subelement—and displays them in a JSP. The <bean:message /> tag defines nine attributes and has no body. Of these nine attributes, we are interested in only three: key, bundle, and locale:

> **key**—The key attribute is the unique value that is used to retrieve a message from the previously defined resource bundle. The key attribute is a request time attribute that is required.
>
> **bundle**—The bundle attribute is the name of the bean under which messages are stored. This bean is stored in the ServletContext. If the bundle is not included, the default value of *Action.MESSAGES_KEY* is used. This attribute is an optional request time attribute. If you use the ActionServlet to manage your resource bundles, you can ignore this attribute.
>
> **locale**—The locale attribute names the session bean that references the requesting client's Locale. If the bundle is not included, the default value of *Action.LOCALE_KEY* is used.

Now that we have described the <bean:message /> tag, it is time to take a look at how it is used. The following code snippet contains a simple example of using the <bean:message /> tag:

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

<html>
```

```
  <head>
    <title><bean:message  key="app.title"/></title>
  </head>
  <body>

  </body>
</html>
```

As you look over the previous snippet, you will see two lines in bold. We need to focus on these two areas. The first of these lines is a JSP taglib directive that must be included by all JSPs that will use the <bean:message /> tag.

Note The URI defined in the previous taglib directive should match the <taglib−uri> defined in the previously defined web.xml file.

The second line that we need to look at is the actual <bean:message /> tag. The <bean:message /> instance that we use in this snippet contains only the key attribute; it retrieves the value stored in the resource bundle that is referenced by the key app.title, and substitutes it for the occurrence of the <bean:message /> tag. The result of this is a JSP that will have an HTML <title> that matches the Locale of the requesting client.

Internationalizing the wileystruts Application

Now that we have seen all of the components involved in internationalizing a Struts application, we can apply them to our wileystruts application. In this section, we take you through the step−by−step process that is required when internationalizing a Struts Web application. Each of these steps is described as follows:

1. Create the resource bundles that will contain the key/value pairs used in your application. For our application, we will have two properties files that contain our resource bundles. These properties files appear in Listings 6.1 and 6.2.

Listing 6.1: The Italian ApplicationResources_it.properties file.

```
app.symbol=Simbolo
app.price=Prezzo  Corrente
```

Listing 6.2: The English ApplicationResources.properties file.

```
app.symbol=Symbol
app.price=Current  Price
```

2. Copy all of the properties files to the <*CATALINA_HOME*>/webapps/*webappname*/WEB−INF/classes/wiley directory.
3. Add an application <message−resources /> subelement, naming the wiley. ApplicationResources to the struts−config.xml file, as shown in Listing 6.3.

```
<?xml  version="1.0"  encoding="ISO-8859-1"  ?>

<!DOCTYPE  struts-config  PUBLIC
  "-//Apache  Software  Foundation//DTD  Struts  Configuration  1.1//EN"
  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
```

```
<struts-config>

  <form-beans>
    <form-bean  name="lookupForm"
      type="wiley.LookupForm"/>
  </form-beans>

  <action-mappings>

    <action  path="/Lookup"
      type="wiley.LookupAction"
      name="lookupForm"  >
      <forward  name="success"  path="/quote.jsp"/>
      <forward  name="failure"  path="/index.jsp"/>
    </action>

  </action-mappings>

  <message-resources
    parameter="wiley.ApplicationResources"/>

</struts-config>
```

4. Add a <taglib> entry, describing the bean tag library to the application's web.xml file, as shown in Listing 6.4.

Listing 6.3: The Modified web.xml file.

```
<?xml  version="1.0"  encoding="ISO-8859-1"?>

<!DOCTYPE  web-app
  PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
      org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
       <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Standard Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- The Usual Welcome File List -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
```

```
  <!-- Struts Tag Library Descriptors -->
  <taglib>
    <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
  </taglib>

  <taglib>
    <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
    <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
  </taglib>

</web-app>
```

Note Make sure that you are using the <load−on−startup> element when describing the ActionServlet. This will ensure that all of the key/value pairs are loaded prior to any requests.

5. Modify your JSP files to include a taglib directive referencing the bean tag library, and replace all text strings presented to the user with matching <bean:message /> tags. Listings 6.4 and 6.5 show our modified JSPs. You will notice that all of the formerly presented strings have been placed in the properties files, listed earlier, and are now referenced using a <bean:message /> tag with the appropriate key.

Listing 6.4: The Internationalized index.jsp.

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<html>
  <head>
    <title>Wiley Struts Application</title>
  </head>

  <body>
    <table width="500"
      border="0" cellspacing="0" cellpadding="0">
      <tr>
        <td> </td>
      </tr>
      <tr bgcolor="#36566E">
        <td height="68" width="48%">
          <div align="left">
            <img src="images/hp_logo_wiley.gif"
              width="220"
            height="74">
          </div>
        </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
    </table>

    <html:form action="Lookup"
      name="lookupForm"
      type="wiley.LookupForm" >
      <table width="45%" border="0">
        <tr>
```

```
          <td><bean:message key="app.symbol" />:</td>
          <td><html:text property="symbol" /></td>
        </tr>
        <tr>
          <td colspan="2" align="center"><html:submit /></td>
        </tr>
      </table>
    </html:form>

  </body>
</html>
```

Listing 6.5: The Internationalized quote.jsp.

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

<html>
  <head>
    <title>Wiley Struts Application</title>
  </head>
  <body>

    <table width="500"
      border="0" cellspacing="0" cellpadding="0">
      <tr>
        <td> </td>
      </tr>
      <tr bgcolor="#36566E">
        <td height="68" width="48%">
          <div align="left">
            <img src="images/hp_logo_wiley.gif"
              width="220" height="74">
          </div>
        </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
      <tr>
        <td>
          <bean:message key="app.price" />:
            <%= request.getAttribute("PRICE") %>
        </td>
      </tr>
      <tr>
        <td> </td>
      </tr>
    </table>
  </body>
</html>
```

That's all there is to it. To see these changes take effect, restart Tomcat, and open the following URL:

```
file:///M:/For_pouellette/Text/outputImages
```

You should see results that look exactly like your previous encounters with the wileystruts application, except that now all user−presented strings are retrieved from the ApplicationResources.properties file that matched the requesting client's Locale.

Summary

In this chapter, we took a look at the internationalization (i18n) features of the Struts Framework. We began by defining each Struts i18n component, and discussed how it is used and configured. We then went through the steps involved when internationalizing an existing Struts application.

In the next chapter, we will discuss how errors are managed by the Struts framework. We will be discussing how errors are both managed and presented to the user.

Chapter 7: Managing Errors

In this chapter, we look at some of the methods available when managing errors in a Struts application. We begin by discussing the various error classes provided by the Struts Framework. We also examine how errors are managed in both the Controller and Views of a Struts application by adding error handling to our wileystruts stock quote application.

The goal of this chapter is to show you how errors can be managed in a Struts application. At the end of this chapter, you will know how and where the Struts error−management component can be leveraged.

Struts Error Management

The Struts Framework is packaged with two main classes that are intended for error management. The first of these classes is the ActionError class, which represents an encapsulation of an error message. The second error management class is the ActionErrors class, which acts as a container for a collection of ActionError instances. We look at both of these classes in this section.

ActionError

The first of our error−management classes, the org.apache.struts.action.ActionError class, represents a single error message. This message—most often created in either an Action or an ActionForm instance—is composed of a message key, which is used to look up a resource from the application resource bundle, and up to four replacement values, which can be used to dynamically modify an error message.

Note The methods of the ActionError class are used by the Struts Framework to assemble the human−readable message and are not often used by the Struts developer; therefore, we will focus only on the constructors of this object.

The ActionError class can be instantiated using one of five different constructors. The method signatures for each of these constructors are shown here:

```
public ActionError(java.lang.String key)
public ActionError(java.lang.String key,
                   java.lang.Object value0)
public ActionError(java.lang.String key,
                   java.lang.Object value0,
                   java.lang.Object value1)
public ActionError(java.lang.String key,
                   java.lang.Object value0,
                   java.lang.Object value1,
                   java.lang.Object value2)
public ActionError(java.lang.String key,
                   java.lang.Object value0,
                   java.lang.Object value1,
                   java.lang.Object value2,
                   java.lang.Object value3)
```

The key attribute of the ActionError class is used to look up a resource from the application resource bundle described in Chapter 6, “Internationalizing Your Struts Applications.” This allows you to provide error messages that are i18n− enabled. We will see examples of this when we add error management to our wileystruts application.

The value0..3 attributes allow you to pass up to four replacement objects that can be used to dynamically modify messages. This allows you to parameterize an internationalized message.

Here's an example of constructing an ActionError:

```
ActionError error = new ActionError("errors.lookup.unknown",
                                    symbol);
```

This ActionError instance would look up the resource bundle string with the key errors.lookup.unknown, and substitute the value of the symbol object as the retrieved resource's first parameter. If we were to assume our resource bundle contained the entry

```
errors.lookup.unknown=Unknown Symbol : {0}and the symbol object was a String containing the val
Unknown Symbol : BOBCO
```

Note The placeholders used by the ActionError class are formatted according the standard JDK's java.text.MessageFormat, using the replacement symbols of {0}, {1}, {2}, and {3}.

ActionErrors

The second of our error−management classes, the org.apache.struts.action. ActionErrors class, represents a collection of ActionError classes. This class contains an internal HashMap of ActionError objects that are keyed to a property or the global application.

The ActionErrors class is composed of a single default constructor and eight methods that are used to query and manipulate the contained ActionError instances. Table 7.1 describes the methods of the ActionErrors class.

Table 7.1: The Methods of the ActionErrors Class

Method	Description
add()	Adds an ActionError instance, associated with a property, to the internal ActionErrors HashMap. You should note that the internal HashMap contains an ArrayList of ActionErrors. This allows you to add multiple ActionError objects bound to the same property.
clear()	Removes all of the ActionError instances currently stored in the ActionErrors object.
empty()	Returns true if no ActionError objects are currently stored in the ActionErrors collection; otherwise, returns false.
get()	Returns a Java Iterator referencing all of the current ActionError objects, without regard to the property they are bound to.
get(java.lang.String)	Returns a Java Iterator referencing all of the current ActionError objects bound to the property represented by the String value passed to this method.
properties()	Returns a Java Iterator referencing all of the current properties bound to ActionError objects.

size()	Returns the number of ActionError objects, without regard to the property they are bound to.
size(java.lang.String)	Returns the number of ActionError objects bound to the property represented by the String value passed to this method.

The add() method is the method most often used when managing collections of errors. The following code snippet contains two add() methods, and shows how ActionError objects can be added to the ActionErrors collection:

```
ActionErrors errors = new ActionErrors();

errors.add("propertyname",
           new ActionError("key");

errors.add(ActionErrors.GLOBAL_ERROR,
           new ActionError("key");
```

As you can see, the only difference between these two add()s is the first parameter. This parameter represents the property to which the ActionError being added should be bound. The first add() example uses a String as the property value. This tells Struts that this error is bound to an input property from the HTML form that submitted this request. This method is most often used to report errors that have occurred when validating the form in the ActionForm.validate() method.

The second add() example uses the value *ActionErrors.GLOBAL_ERROR* as the property value. This tells Struts that this error is not bound to any input property. This method is most often used to report errors that have occurred in an Action.perform() method. We will see examples of both of these methods when we modify the wileystruts application.

Adding Error Handling to the wileystruts Application

Now that we have seen the classes involved in Struts error management, let's look at how they are actually used. We will do this by adding the Struts error−management components to our wileystruts Web application.

Before you can leverage the Struts error−management classes, you must add two attributes to the <action> element that describe the wiley.LookupAction. The following code snippet shows the changes to the struts−config.xml file:

```
<action path="/Lookup"
  type="wiley.LookupAction"
  name="lookupForm"
  validate="true"
  input="/index.jsp">
  <forward name="success" path="/quote.jsp"/>
  <forward name="failure" path="/index.jsp"/>
</action>
```

The new attributes are the validate and input attributes. The first attribute, validate, when set to true tells the Struts framework that validation should be performed. The second attribute tells the Struts frame where the error originated and where the action should be redirected, if any errors have occurred. You must add these attributes to all <action> elements that will use the ActionForm.validate() mechanism described in the following section.

The ActionForm.validate() Method

The first area where we are going to apply error−management techniques is in the ActionForm object. This is probably the best place to begin, because it is the first chance you will have to test the incoming request for errors. The errors that we are checking for are validation errors that occur when the user submitting an HTML form enters incorrect data. The Struts Framework allows us to do this by simply overriding the ActionForm.validate() method. The signature of this method is as follows:

```
public ActionErrors validate(ActionMapping mapping,
  HttpServletRequest request)
```

The ActionForm.validate() method is called by the ActionServlet after the matching HTML input properties have been set. It provides you with the opportunity to test the values of the input properties before the targeted Action.perform() method is invoked. If the validate() method finds no errors in the submitted data, then it returns either an empty ActionErrors object or null, and processing continues normally.

If the validate() method does encounter errors, then it should add an ActionError instance describing each encountered error to an ActionErrors collection, and return the ActionErrors instance. When the ActionServlet receives the returned ActionErrors, it will forward the collection to the JSP that is referenced by the input attribute described previously, which in our case is the index.jsp. We will see what the index.jsp View will do with the ActionErrors collection later in this section. Listing 7.1 contains the changes we have made to our LookupForm to perform input validation.

Listing 7.1: The Modified LookupForm.java.

```
package wiley;

import  javax.servlet.http.HttpServletRequest;
import  org.apache.struts.action.ActionForm;
import  org.apache.struts.action.ActionMapping;
import  org.apache.struts.action.ActionError;
import  org.apache.struts.action.ActionErrors;

public class LookupForm extends ActionForm {

  private String symbol = null;

  public String getSymbol() {

    return (symbol);
  }

  public void setSymbol(String symbol) {

    this.symbol = symbol;
  }

  public void reset(ActionMapping mapping,
    HttpServletRequest request) {

    this.symbol = null;
  }

  public ActionErrors validate(ActionMapping mapping,
    HttpServletRequest request) {
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

STRUTS FRAME WORK – UNIT II

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	___ defines the availability of struts JSP custom tag libraries	Struts-taglib	JSP-taglib	taglib	tagLibrary	**taglib**
2	___ file is used by controller to get mapping information for request routing	Struts.xml	Struts-cfg.xml	Struts-config.xml	config.xml	**Struts-config.xml**
3	____is used to present the actual strings that have been loaded by the controller	<message/>	<bean:text/>	<bean:message/>	<text/>	**<bean:message/>**
4	<bean:message/> tag has __ attributes	9	2	8	4	**9**
5	The Struts Framework packaged with __ main classes for error management.	1	2	3	4	**2**
6	____ represents an encapsulation of an error message	ActionErrorClass	ActionErrorsClass	ActionSupport	ActionClass	**ActionErrorClass**
7	____ acts as a container for collection of ActionError instances	ActionErrorsClass	ActionErrorClass	ActionClass	ActionSupport	**ActionErrorsClass**
8	____ contains an internal HashMap of ActionError objects.	ActionClass	ActionSupport	ActionErrorClass	ActionErrorsClass	**ActionErrorClass**
9	The ActionForm.validate() method is called by the ActionServlet	after matching HTML input properties	Before targeted Action.perform() invoked	Option 1 only	Option 1 & 2	**Option 1 & 2**
10	To way to display any errors resulting from our validation.	<html:errors/>	html error	html text	html header	**<html:error/>**
11	The header and footer values are identified using	header	footer	header & footer	text keys	**text keys**
12	An ___object describes an Action instance to the ActionServlet.	ActionErrorClass	ActionInstance	ActionServlet	ActionMapping	**ActionMapping**
13	The <action/> subelement is used to describe an Action instance to the ___	action Class	action-mappings	actionServlet	application	**actionServlet**
14	____ is designed to provide functionalities required by action	request processor	interceptor	stack	actionserver	**interceptor**

15	Interceptor approach helps in	modularizing code into reusable classes	Used to provide required functionalities	Provide all preprocessing of the request	All the above	**All the above**
16	Interceptor is	unpluggable	portable	pluggable	unportable	**pluggable**
11	Interceptor must be declared in	struts.config..xml	struts.xml	struts.cfg.xml	struts-default.xml	**struts.xml**
13	The <action/> subelement is used to describe an Action instance to the ___	action Class	action-mappings	actionServlet	application	**actionServlet**
14	____ is designed to provide functionalities required by action	request processor	interceptor	stack	actionserver	**interceptor**
15	Interceptor approach helps in	modularizing code into reusable classes	Used to provide required functionalities	Provide all preprocessing of the request	All the above	**All the above**
16	Interceptor is	unpluggable	portable	pluggable	unportable	**pluggable**
18	Interceptor must be declared in	struts.config..xml	struts.xml	struts.cfg.xml	struts-default.xml	**struts.xml**
19	The tag enables developers to call actions directly from a JSPpage	generator tag	Action tag	Include tag	Bean tag	**Action Tag**
20	The _____ tag assigns a value to a variable in a specified scope	set tag	Text tag	url tag	Push tag	**settag**
21	Types of Validator in struts	date Validator	double validator	email validator	all the above	**all the above**
	Tag to get the property of a value	date tag	param tag	property tag	push tag	**property tag**
22.	_____ container that holds the components of a web aplication	Sub directory	Class	API	Directory Structure	**Directory Structure**
23	All the web applications are packaged into standard	Directory Structure	browser	interceptor	server	**Directory Structure**
24.	Web application invoked by	Web Browser	Web Server	Web component	Web application	**Web Browser**
25	Web application executed by	Web Browser	Web Server	Web component	Web application	**Web Server**
26	___ oriented web application contains static and dynamic web pages	Presentation	Service	Server	browser	**Presentation**
27	____ oriented web applications are implemented as an end point of web service	Presentation	Service	Server	browser	**Service**
28	___ method requests the server to provide the information.	POST	PUT	GET	POST	**GET**
29	____ submits the server the data to be processed by the resource	POST	PUT	GET	POST	**POST**
30	The web server recognizes the request for Java Server Pages by the ___ file extension.	.java	.jsp	.jdk	.js	**.jsp**
31	The JSP engine translates the JSP page into a	Java class	JSP class	JDK class	JSF class	**Java Class**

32	To bind values into views, the ___ technology used	ServletRequest	DefaultStack	ValueStack	InterceptorStack	**ValueStack**
33	OGNL	Object Graphics Net Language	Object Graph Notation Language	Object Graphical Notation Language	Graphics Oriented Net Language	**Object Graph Notation Language**
34	__ file is used to generate input screen.	error.jsp	web.xml	struts.xml	index.jsp	**index.jsp**
35	__ file is used to display an error message.	error.jsp	web.xml	struts.xml	index.jsp	**error.jsp**
36	___containing action mappings	error.jsp	web.xml	struts.xml	index.jsp	**struts.xml**
37	____ is the deployment descriptor of web application.	error.jsp	web.xml	struts.xml	index.jsp	**web.xml**
38	Interceptors are used to ____ a request.	pre-process	post-process	pre-process and post-process	process	**pre-process and post-process**
39	The __ attribute of package acts as the key	extends	namespace	name	abstract	**name**
40	__ gets message based on a message key	String getText	Stringkey	ResourceBundle getTexts	Stringobj	**String getText**
41	____ tags are used to control the behavior of data in a page.	control	bean	data	html	**control**
42	___ tags are used for creating and manipulating data.	control	bean	data	html	**data**
43	___ tag is used to include the Servlet.	insert	include	infix	includeServlet	**include**
44	The ___ configurations are the basic unit of struts	action				
45	The element of action attribute is	class	name	result	name & class	**name and class**
46	The subelement of action attribute is	class	name	result	name & class	**result**
47	___ acts as an action factory by creating action classes.	Action servlet	Action class	JSP Servlet	HTTP Servlet	**Action Servlet**
48	The process method gets the current	browser	processor	Request processor	server	**Request processor**
49	The input buffersize is	4096 bytes	4096 KB	4096 MB	4096 GB	**4096 bytes**
50	Struts ___ are modular extensions to the Struts controller.	actionservlet	action	plugin	actionclass	**plugin**
51	Struts ____ are useful when allocating resources to databases.	actionservlet	action	plugin	actionclass	**plugin**
52	The description property of a datasource instance is_____	text	sting	character	instance	**text**
53	The add method of action error class add ___	Action error instance	Action error object	Action error	Arraylist	**Action error instance**

54	___ returns a java iterator referencing all of the current actionerror without regard to the property	get()	get(java.lang.string)	properties()	add()	**get()**
55	___ method returns the number of Actionerror objects.	get()	add()	size()	properties()	**size()**
56	The first area to apply error management technique is ___	Actionform object	ActionServlet object	ActionError object	Actionmapping object	Actionform object
57	The Actionform validate() method is called by	ActionServlet	ActionError	Validate	Actionmapping	**ActionServlet**
58	The ___ tag used to display the Actionerror objects	html	logic	control	bean	**html**
59	The ___method is used to report errors	report	errors	add	perform	**perform**
60	Action mapping extension allows us to turn on and off action debug on ___	action	debug	class	extension	**action**

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

STRUTS FRAME WORK – UNIT III

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	The major subelements of Stuts componenets.	<icon/>,<display-name/>, <set-property /> <bean:cookie/>	<icon/>,<bean:define/> <set-property /> <description/>	<icon/>,<display-name/>, <bean:stuts /> <description/>	<icon/>,<display-name/>, <set-property /> <description/>	**<icon/>,<display-name/>, <set-property /> <description/>**
2	The ____ subelement used to graphically represent its parent element	<display-name/>	<icon/>	<description/>	<set-property/>	**<icon/>**
3	The ____ subelement contains a short texual description.	<display-name/>	<icon/>	<description/>	<set-property/>	**<display-name>**
4	The ____subelement contains a full length textual description.	<display-name/>	<icon/>	<description/>	<set-property/>	**<description/>**
5	The ____ element contain-number if <data-source> subelements.	<data-source/>	<set-property>	<data-sources/>	<icon/>	**<data-sources/>**
6	JDBC	Java DataBase Connectivity	Java Driver Class	Java Data Beans Connector	Java Data Base Connector	**Java DataBase Connectivity**
7	JDBC is a	Data Source	Data Base	Driver Class	Data Connector	**Data Source**
8	Types of datasource implementation	2	3	4	5	**3**
9	____data source generates standard JDBC connection objects	basic	pooled	distributed	connector	**basic**
10	The <form-bean/> subelement is used to describe _____	an element	a form	an instance	a subelement	**an instance**
11	<global-forwards> acts as a container for public ____ subelements.	<global/>	<forward/>	<global-forward>	<global-forwards>	**<forward/>**
12	The container for <form-bean/> subelements is	<form-bean>	<global-forwards>	<form-beans>	<form:bean>	**<form-beans>**
13	<action-mapping> subelement is used to define n number of	<data-source/>	<action/>	<forward/>	<controller/>	**<action/>**
14	____ is used to modify the default behavior of the Struts Controller.	<data-source/>	<action/>	<forward/>	<controller/>	**<controller/>**

15	___ subelement is used to define the collection of messages.	<data-source/>	<message-resources/>	<resources-msg/>	<msg-resources/>	**<message-resources/>**
16	<controller/> subelement define a	m	request Processor	plug-in	action	**Request processor**
11	____ provides a group of tags encapsulate and manipulate JavaBeans.	Bean tags	Control tag	General tag	Html tag	**Bean tag**
12	All bean tags should be prefixed with a string ____	action	html	bean	bean tag	**bean**
13	The ___ tag is used to retrieve the value of an HTTP Cookie.	<bean:cookie/>	<bean||cookie/>	<bean::cookie/>	<bean//cookie/>	**<bean:cookie>**
14	The ____ tag is used to retrieve the value of named bean property.	<bean:name/>	<bean:property/>	<bean:define/>	<bean:value/>	**<bean:define/>**
15	<bean:header/> tag functions like	<bean:include/>	<bean:define/>	<bean:message>	<bean:cookie/>	<bean:cookie/>
16	<bean:header/> tag stores header values in	Message	Header Page	PageContext	PageHeader	**PageContext**
17	___tag is used to evaluate and retrieve the results of a web application resource.	<bean:name>	<bean:include/>	<bean:message>	<bean:cookie/>	<bean:include/>
18	The <bean:message/> tag has no	body	attributes	header	files	**body**
19	The ___ tag is used to retrieve the value of JSP object stores in page context.	<bean:message>	<bean:parameter>	<bean:resoure>	<bean:page>	**<bean:page>**
20	<bean:parameter/> tag retrieve the value of a	request parameter	response parameter	resource parameter	retrieve parameter	**request parameter**
21	<bean:resource> tag is used to retrieve the value of web application resource by__ attribute	name	input	id	property	**name**
22	<bean:size/> tag is used to retrieve the number of elements in	an array	a collection	map	all the above	**all the above**
23	____ tag is used to copy a specified struts internal component	<bean:struts/>	<beans:formbean/>	<bean:component/>	<bean:write/>	**<bean:struts/>**
24	____ tag is used to retrieve and print the value of a named bean property	<beans:formbean/>	<bean:struts/>	<bean:write/>	<bean:cookie>	**<bean:write/>**
25						
26						
27						

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

STRUTS FRAME WORK – UNIT III

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	The major subelements of Stuts componenets.	<icon/>,<display-name/>, <set-property /> <bean:cookie/>	<icon/>,<bean:define/> <set-property /> <description/>	<icon/>,<display-name/>, <bean:stuts /> <description/>	<icon/>,<display-name/>, <set-property /> <description/>	**<icon/>,<display-name/>, <set-property /> <description/>**
2	The ____ subelement used to graphically represent its parent element	<display-name/>	<icon/>	<description/>	<set-property/>	**<icon/>**
3	The ____ subelement contains a short textual description.	<display-name/>	<icon/>	<description/>	<set-property/>	**<display-name>**
4	The ____subelement contains a full length textual description.	<display-name/>	<icon/>	<description/>	<set-property/>	**<description/>**
5	The ____ element contain-number of <data-source> subelements.	<data-source/>	<set-property>	<data-sources/>	<icon/>	**<data-sources/>**
6	JDBC	Java DataBase Connectivity	Java Driver Class	Java Data Beans Connector	Java Data Base Connector	**Java DataBase Connectivity**
7	JDBC is a	Data Source	Data Base	Driver Class	Data Connector	**Data Source**
8	Types of data source implementation	2	3	4	5	**3**
9	____data source generates standard JDBC connection objects	basic	pooled	distributed	connector	**basic**
10	The <form-bean/> subelement is used to describe _____	an element	a form	an instance	a subelement	**an instance**
11	<global-forwards> acts as a container for public ____ subelements.	<global/>	<forward/>	<global-forward>	<global-forwards>	**<forward/>**
12	The container for <form-bean/> subelements is	<form-bean>	<global-forwards>	<form-beans>	<form:bean>	**<form-beans>**
13	<action-mapping> subelement is used to define n number of	<data-source/>	<action/>	<forward/>	<controller/>	**<action/>**
14	____ is used to modify the default behavior of the Struts Controller.	<data-source/>	<action/>	<forward/>	<controller/>	**<controller/>**

15	___ subelement is used to define the collection of messages.	<data-source/>	<message-resources/>	<resources-msg/>	<msg-resources/>	**<message-resources/>**
16	<controller/> subelement define a	m	request Processor	plug-in	action	**Request processor**
11	_provides a group of tags encapsulate and manipulate JavaBeans.	Bean tags	Control tag	General tag	Html tag	**Bean tag**
12	All bean tags should be prefixed with a string ____	action	html	bean	bean tag	**bean**
13	The ___ tag is used to retrieve the value of an HTTP Cookie.	<bean:cookie/>	<bean||cookie/>	<bean::cookie/>	<bean//cookie/>	**<bean:cookie>**
14	The ____ tag is used to retrieve the value of named bean property.	<bean:name/>	<bean:property/>	<bean:define/>	<bean:value/>	**<bean:define/>**
15	<bean:header/> tag functions like	<bean:include/>	<bean:define/>	<bean:message>	<bean:cookie/>	<bean:cookie/>
16	<bean:header/> tag stores header values in	Message	Header Page	PageContext	PageHeader	**PageContext**
17	___tag is used to evaluate and retrieve the results of a web application resource.	<bean:name>	<bean:include/>	<bean:message>	<bean:cookie/>	<bean:include/>
18	The <bean:message/> tag has no	body	attributes	header	files	**body**
19	The ___ tag is used to retrieve the value of JSP object stores in page context.	<bean:message>	<bean:parameter>	<bean:resoure>	<bean:page>	**<bean:page>**
20	<bean:parameter/> tag retrieve the value of a	request parameter	response parameter	resource parameter	retrieve parameter	**request parameter**
21	<bean:resource> tag is used to retrieve the value of web application resource by_____ attribute	name	input	id	property	**name**
22	<bean:size/> tag is used to retrieve the number of elements in	an array	a collection	map	all the above	**all the above**
23	____ tag is used to copy a specified struts internal component	<bean:struts/>	<beans:formbean/>	<bean:component/>	<bean:write/>	**<bean:struts/>**
24	____ tag is used to retrieve and print the value of a named bean property	<beans:formbean/>	<bean:struts/>	<bean:write/>	<bean:cookie>	**<bean:write/>**
25	The <display-name> subelement contains a____ textual description.	short	long	brief	label	**short**
26	The <description> subelement contains a____ textual description.	short	long	brief	label	**long**
27	The <icon> subelement used to graphically represent its ___ element.	child	sub	parent	icon	**parent**
28	Basic data source generates standard _____connection objects.	database	JDBC	data	java	**JDBC**

29	The ____ subelement is used to describe instance.	<forward>	<form-instance>	<form-bean/>	<icon>	**<form-bean/>**
30	<global-forwards> acts as a container for ___subelements.	public	private	global	all	**<forward/>**
31	The <form-beans> is the container for ____ subelements	<form-bean>	<global-forwards>	<form-beans>	<form:bean>	**<form-bean>**
32	<action-mapping> is used as the container for ___ subelements.	<data-source/>	<action/>	<forward/>	<controller/>	**<action/>**
33	____ is used to modify the default behavior of the Struts Controller.	<data-source/>	<action/>	<forward/>	<controller/>	**<controller/>**
34	Small icon subelement names a graphics file that contains a _____ pixel iconic image.	32 x 32 pixel	16 x 16 pixel	8 x 8 pixel	24 x 24 pixel	**16 x 16 pixel**
35	Large-icon subelement names a graphics file that contains a _____ pixel iconic image	32 x 32	16 x 16	8 x 8	24 x 24	**32 x 32** pixel
36	The images referenced in configuration files are NOT intended for ___ display	Client- side	Server- side	View side	Model side	**Client side**
37	The ___ attribute of data-source entry is used to set minimum number of connections open at any time	mincount	maxcount	maximumcount	minimumcount	**mincount**
38	The ___ attribute of data-source entry is used to set maximum number of connections open at any time	mincount	maxcount	maximumcount	minimumcount	**maxcount**
39	The unique key bound the datasource instance will be ____	data-key	data-source key	database key	data-snippet	**data-source key**
40	The data-source key bound in the	servlet	server	serveletcontext	servletcontent	**serveletcontext**
41	The original values stored in the httpservletrequest will be lost if the ___ method used.	path	name	redirect	type	**redirect**
42	The maximum file size to be uploaded can be expressed by __ bytes.	K, M or G	M,G or T	G,T or P	K,G or T	**K, M or G**
43	The default value of file size uploaded is	250 GB	250 PB	250 MB	250 TB	**250 MB**
44	The ___ attribute used to store file being uploaded.	tempDir	temporary directory	tempdirectory	Tempdir	**tempDir**

45	The ___ attribute returns null string for unknown message keys.	null	string	empty	zero	**null**
46	The ___ attribute identifies the scope of newly defined bean.	scope	toscope	newscope	define	**toscope**
47	The ___ attribute identifies the scope of the bean specified by name attribute.	scope	toscope	namescope	define	**scope**
48	___ projects are used to manage application information	Application information	Class path	Source path	All the above	**All the above**
49	____ are like stop signs to the IDE debugger.	Jbuilder	Breakpoints	debugger	Startpoint	**breakpoint**
50	The toscope attribute identifies the scope of _____.	newly defined bean	specified by name bean	existing bean	defined bean	**newly defined bean**
51	The mincount attribute of data-source entry is used to set ____ number of connections open at any time	minimum	maximum	limited	restricted	**minimum**
52	The <description> subelement contains a_long textual description.	<display-name>	<description>	<text>	<longtext>	**<description>**
53	The ____ bound the datasource instance will be data source key.___	data	unique key	key	primary	**Unique key**
54	Controller is used to modify the ___behavior of the Struts Controller.	dynamic	present	default	static	**default**
55	_____ acts as a container for forward subelements.	<forwards>	<global-forward>	<global-forwards>	<global>	**<global-forwards>**
56	Breakpoints are like _____ signs to the IDE debugger.	restart	start	break	stop	**stop**
57	The ___ subelement used to graphically represent its parent element.	<child>	<graphic>	<icon>		
58	<bean-write> tag is used to name the value of a named bean property	write	print	retrieve	retrieve and print	**retrieve and print**
59	____tag retrieve the value of a request parameter.	<bean:parameter/>	<bean:parameter/>	<bean:retrieveparameter/>	<bean:requestparameter/>	**<bean:parameter/>**
60	<bean:resource> tag is used to retrieve the value of web application ____ by name attribute.	response	request	retrieve	resource	**resource**

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

STRUTS FRAME WORK – UNIT IV

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	To include html tags we should add taglib subelement to the	web.xml	web.html	config.xml	Cfg.xml	**web.xml**
2	HTML Tags should be prefixed with the string	htmltag	web	xml	html	**html**
3	The ____ tag is used to insert an HTML base element	<html:base/>	<html:base-element/>	<html:element/>	<base/>	**<html:base/>**
4	The <html:button/> tag is used to render an input element with type of ____	input	button	body	form	**button**
5	The <html:button/> tag nested inside the body of an ___ tag.	<html:base/>	<html:buttons/>	<html:form/>	<html:base/>	**<html:form/>**
6	The <html:cancel> tag is used to render an input element with the type of	input	cancel	button	element	**cancel**
7	The _____ tag has a body type of JSP.	<html:body/>	<html:write/>	<html:cancel/>	<html:base/>	**<html:cancel/>**
8	The ___ is used to render an HTML input element with an input type of checkbox.	<html:input/>	<html:checkbox/>	<html:inputelement/>	<html::checkbox/>	<html:checkbox/>
9	The <html:errors/> tag is used to display the ActionError ____.	objects	classes	collections	errors	**objects**
10	To create an HTML form	<html:forms/>	<html:formbean/>	<html:form/>	<html:form-create/>	**<html:form/>**
11	The <html:hidden> tag is used to render the input element with input type of	hide	hidden	input-hide	input-hidden	**hidden**
12	To render the top level element ___ tag is used.	<html:toplevel/>	<html:html/>	<html:hidden/>	<html:element/>	**<html:html>**
13	To render an html input element with image ___ tag is used	<html:image>	<html:input-image>	<html-image>	<html:img>	**<html:image>**
14	The image tag must specify __ attrubutes	url	src or page	altkey	accesskey	src or page

15.	The <html:image/> tag nested inside the body of an	<html:html/>	<html:image/>	<html:form/>	<html:forms>	**<html:form/>**
15	___ tag is used to render an html <img> element.	<html:image>	<html:img>	<html:images>	<html:form/>	**<html:img>**
16	<html:link/> tag is used to generate an HTML ____	url	link	hyperlink	href	**hyperlink**
11	<html:multibox/> tag is used to generate type of __	multibox	option button	checkbox	forms	**multibox**
12	Access key attribute identifies ___ character.	keyboard	ASCII	unicode	string	**keyboard**
13	___ attribute defines an alternate text string	alt	altkey	accesskey	name	**alt**
14	__ attribute specify a javascript function that will be executed when this element is under the mouse pointer and button is pressed	onmousedown	onmousemove	onmouseout	onmouseover	**onmousedown**
15	__ attribute specify a javascript function that will be executed when this element is under the mouse pointer and pointer is moved.	onmousedown	onmousemove	onmouseout	onmouseover	**onmousemove**
16	__ attribute specify a javascript function that will be executed when this element is under the mouse pointer and pointer is moved outside the element.	onmousedown	onmousemove	onmouseout	onmouseover	**onmouseout**
17	_____ attribute identify a data member of the bean	property	style	styleclass	title	**property**
18	<html:select> is used to render an html input element with a type of	Select-input	property	elements	select	**select**
19	The ____attribute used to identify the tab order of the elements in the form	tab	taborder	tabindex	tabform	**tabindex**
20	The attribute ___ specifies a cascading style sheet to apply to the HTML element.	style sheet	style	styleld	stylecss	**style**
21	The tag is used to generate an html input element of type option	<html:option/>	<html::option/>	<html:type/>	<html:type-option/>	**<html:option/>**
22	The parent element for html input option element	<html:option/>	<html:input/>	<html:select/>	<select/>	**<select/>**
23	<html:options> is used to list the HTML ____ elements	option	select	input	value	**option**

24	The ____ tag is used to render the html input element with the type of password.	<html:pwd/>	<html:password/>	<html:pwd/>	<html:paswd/>	**<html:password/>**
25	___ tag is used to render an html with input type of radio.	<html::radio/>	<html||radio/>	<html//reset/>	<html:radio/>	**<html:radio/>**
26	____ tag is used to render an html with an input type of reset.	<html::radio/>	<html:reset/>	<html:resetting/>	<html:rewrite/>	**<html:reset/>**
27	___ tag is used to create a URI request.	<html::radio/>	<html:reset/>	<html:request/>	<html:rewrite/>	**<html:rewrite/>**
28	____ tag is used to render the html input with a type of submit	<html:submit/>	<html:rewrite/>	<html:resubmit/>	<html:submission/>	**<html:submit/>**
29	____ tag is used to render the html input element with the type of text.	<html:textarea/>	<html:character>	<html:text>	<html:string>	**<html:text>**
30	___tag is used to render an html input element with type of texarea.	<html:textarea/>	<html:character>	<html:text>	<html:str>	**<html:textarea>**
31	To include html tags we should add __ subelement to the web.xml	taglib	request processor	global forward	struts-tag	**taglib**
32	____ should be prefixed with the string html	html tag	bean tag	control tag	logic	**Html tag**
33	The <html:base/> tag is used to insert an HTML ____ element	input	sub	base	output	**base**
34	The ___ tag is used to render an input element with type of button.	<html:button>	<html:optionbutton>	<html:checkbox>	<html:html>	**<html:button>**
35	The <html:button/> tag nested inside the ___ of an <html:form/> tag.	title	body	head	form	body
36	___attribute specifies an HTML identifier to be associated with this HTML element.	styleId	style	styleclass	stylesheet	**styleid**
37	__ attribute specifies the label to be placed on this button.	label	name	value	display	**value**
38	alt attribute defines an alternate ___ string for this element.	text	image	video	audio	**text**
39	___ attribute defines the width, in pixels, of the image border.	height	width	border	image	border
40	<img src="/webappname/images/ add.gif" alt="Add to Basket"> What will be the output when the image cannot be found?	add.gif image will be display	any image will be display	image will not display	Add to Basket	**Add to Basket**

41	____ specifies a JavaScript function that will be executed when the containing element loses its focus	onblur	onfocus	onclick	click	**onblur**
42	The ____ tag is used to render the input element with input type of hidden.	<html:hide>	<html:hidden>	<html:input-hide>	<html:input-hidden>	**<html:hidden>**
43	___ tag is used to associate the array of strings	<html:multibox/>	<html:checkbox/>	<html:option/>	<html:array/>	**<html:multibox>**
44	___ property is used to determine the currently selected <option> of the <select> element,	checkbox	option	image	select	**select**
45	It the ___attribute set to true, the input field generated by this tag sets to uneditable.	write	enable	edit	readonly	readonly
46	Anchor attribute is used to ___ an HTML anchor to the end of a generated hyperlink.	Insert	add	append	remove	**append**
47	HTML tags acts as a bridge between	JSP & model	Client & server	View & model	Controller & model	**Jsp & model**
48	Which of the following tags in struts configuration file defines the JSPavailability of necessary Struts JSP custom tag libraries?	jsplib	taglib	JSP-lib	Struts-taglib	**taglib**
49	HTML is a subset of	linux	SGML	Unix	XML	**SGML**
50	_____ attribute identifies keyboard character.	Access key	Console	Keyboard	Char	Access key
51	Alt attribute defines an alternate text __.	char	special character	text	numeric	**text**
52	Onmousedown attribute specify a javascript function that will be executed when this element is _____ and button is pressed	under the mouse pointer	under the mouse pointer and pointer is moved	under the mouse pointer and pointer is moved outside the element	on the mouse button	**under the mouse pointer**
53	Onmousemove attribute specify a javascript function that will be executed when this element is	under the mouse pointer	under the mouse pointer and pointer is moved	under the mouse pointer and pointer is moved outside the element	on the mouse button	**under the mouse pointer and pointer is moved**
54	Onmouseout attribute specify a javascript function that will be executed when this element is	under the mouse pointer	under the mouse pointer and pointer is moved	under the mouse pointer and pointer is moved outside the element	on the mouse button	**under the mouse pointer and pointer is moved outside the element.**
55	Property attribute identify a ___ of the bean	data	data member	class	object	**data member**
56	____ is used to render an html input element with a type of select	<html:element>	<html:html>	<html:select>	<html:input>	<html:select>

57	The tabindex attribute used to identify the tab order of the elements in the form	tab	tab order	tab index	tab form	**Tab order**
58	CSS ____	Continuous Style Sheet	Cascading Sheet Style	Cascading String Style	Cascading Style Sheet	**Cascading Style Sheet**
59	The <html:option>tag is used to generate an html ____ element of type option	input	optional	process	output	**input**
60	___ attribute specifies a JavaScript function that will be executed when this element loses input focus and its value has changed.	onmouseout	onfocus	onchange	onclick	**onchange**

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

STRUTS FRAME WORK – UNIT IV

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	To include html tags we should add taglib subelement to the	web.xml	web.html	config.xml	Cfg.xml	**web.xml**
2	HTML Tags should be prefixed with the string	htmltag	web	xml	html	**html**
3	The ____ tag is used to insert an HTML base element	<html:base/>	<html:base-element/>	<html:element/>	<base/>	**<html:base/>**
4	The <html:button/> tag is used to render an input element with type of ____	input	button	body	form	**button**
5	The <html:button/> tag nested inside the body of an ___ tag.	<html:base/>	<html:buttons/>	<html:form/>	<html:base/>	**<html:form/>**
6	The <html:cancel> tag is used to render an input element with the type of	input	cancel	button	element	**cancel**
7	The _____ tag has a body type of JSP.	<html:body/>	<html:write/>	<html:cancel/>	<html:base/>	**<html:cancel/>**
8	The ___ is used to render an HTML input element with an input type of checkbox.	<html:input/>	<html:checkbox/>	<html:inputelement/>	<html::checkbox/>	<html:checkbox/>
9	The <html:errors/> tag is used to display the ActionError ____.	objects	classes	collections	errors	**objects**
10	To create an HTML form	<html:forms/>	<html:formbean/>	<html:form/>	<html:form-create/>	**<html:form/>**
11	The <html:hidden> tag is used to render the input element with input type of	hide	hidden	input-hide	input-hidden	**hidden**
12	To render the top level element ___ tag is used.	<html:toplevel/>	<html:html/>	<html:hidden/>	<html:element/>	**<html:html>**
13	To render an html input element with image ___ tag is used	<html:image>	<html:input-image>	<html-image>	<html:img>	**<html:image>**
14	The image tag must specify __ attrubutes	url	src or page	altkey	accesskey	src or page

15.	The <html:image/> tag nested inside the body of an	<html:html/>	<html:image/>	<html:form/>	<html:forms>	**<html:form/>**
15	___ tag is used to render an html <img> element.	<html:image>	<html:img>	<html:images>	<html:form/>	**<html:img>**
16	<html:link/> tag is used to generate an HTML ____	url	link	hyperlink	href	**hyperlink**
11	<html:multibox/> tag is used to generate type of __	multibox	option button	checkbox	forms	**multibox**
12	Access key attribute identifies ___ character.	keyboard	ASCII	unicode	string	**keyboard**
13	___ attribute defines an alternate text string	alt	altkey	accesskey	name	**alt**
14	__ attribute specify a javascript function that will be executed when this element is under the mouse pointer and button is pressed	onmousedown	onmousemove	onmouseout	onmouseover	**onmousedown**
15	__ attribute specify a javascript function that will be executed when this element is under the mouse pointer and pointer is moved.	onmousedown	onmousemove	onmouseout	onmouseover	**onmousemove**
16	__ attribute specify a javascript function that will be executed when this element is under the mouse pointer and pointer is moved outside the element.	onmousedown	onmousemove	onmouseout	onmouseover	**onmouseout**
17	_____ attribute identify a data member of the bean	property	style	styleclass	title	**property**
18	<html:select> is used to render an html input element with a type of	Select-input	property	elements	select	**select**
19	The attribute used to identify the tab order of the elements in the form	tab	taborder	tabindex	tabform	**tabindex**
20	The attribute ___ specifies a cascading style sheet to apply to the HTML element.	style sheet	style	styleld	stylecss	**style**
21	The tag is used to generate an html input element of type option	<html:option/>	<html::option/>	<html:type/>	<html:type-option/>	**<html:option/>**
22	The parent element for html input option element	<html:option/>	<html:input/>	<html:select/>	<select/>	**<select/>**
23	<html:options> is used to list the HTML ____ elements	option	select	input	value	**option**

24	The ____ tag is used to render the html input element with the type of password.	<html:pwd/>	<html:password/>	<html:pwd/>	<html:paswd/>	**<html:password/>**
25	___ tag is used to render an html with input type of radio.	<html::radio/>	<html||radio/>	<html//reset/>	<html:radio/>	**<html:radio/>**
26	____ tag is used to render an html with an input type of reset.	<html::radio/>	<html:reset/>	<html:resetting/>	<html:rewrite/>	**<html:reset/>**
27	___ tag is used to create a URI request.	<html::radio/>	<html:reset/>	<html:request/>	<html:rewrite/>	**<html:rewrite/>**
28	____ tag is used to render the html input with a type of submit	<html:submit/>	<html:rewrite/>	<html:resubmit/>	<html:submission/>	**<html:submit/>**
29	____ tag is used to render the html input element with the type of text.	<html:textarea/>	<html:character>	<html:text>	<html:string>	**<html:text>**
30	___tag is used to render an html input element with type of texarea.	<html:textarea/>	<html:character>	<html:text>	<html:str>	**<html:textarea>**

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

STRUTS FRAME WORK – UNIT V

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	The ___ is on decision making and object evaluation	Html tag	Logic tag	Control tag	Generic tag	**Logic tag**
2	logic Tags should be prefixed with the string	htmltag	logic	xml	html	**logic**
3	The logic taglib contains ____ tags.	12	13	14	15	**14**
4	The _____ tag evaluates the named scripting variable is equal to null or an empty string.	<logic:empty/>	<logic:null/>	<logic:emptystring/>	<logic:nullstring/>	**<logic:empty/>**
5	The ___ tag evaluates if the named scripting variable is not equal to null or does not contain an empty string	<logic:null/>	<logic:notEmpty/>	<logic:empty/>	<logic:nullstring>	**<logic:notEmpty>**
6	The <logic:equal/> tag evaluates the variable equals the ___ value	constant	Integer	double	String	**constant**
7	The ____ tag evaluates the variable not equals the constant value	<logic:notEqual/>	<logic:nonequal>	<logic:notEqual/>	<logic:equal/>	**<logic:notEqual/>**
8	<logic:forward> tag is used to ____ control of the current request to previously identified element.	next	previous	forward	globalforward	forward
9	<logic:redirect> tag uses to redirect the current ____ to a resource.	response	request	action	element	**request**
10	The ___ tag evaluates if the variable is greater than the constant value	<logic:isgreaterThan>	<logic:greater>	<logic:isgreaterThan>	<logic:greaterThan>	<logic:greaterThan>
11	___ attribute specifies an HTTP Cookie to be used as a variable.	variable	cookie	property	value	**cookie**
12	___ attribute indicates the iteration begin.	iteration	offset	begin	scope	**offset**
13	<logic:iterate> tag is used to iterate over a named collection which contains	Enumerator,iterator,map, array	Enumerator,map, array	iterator,map, array	Enumerator,iterator, array	**Enumerator,iterator,map, array**

14	The ____ tag evaluates if the variable is less than or equal to the constant value.	<logic:lessThan>	<logic:islessEqual>	<logic:lessEqual>	<logic:islessThan>	<logic:lessEqual>
15.	<logic:lessThan> tag evaluates if the is less than the constant value.	<logic:lessEqual>	<logic:lessThan>	<logic:islessThan>	<logic:islessEqual>	<logic:lessThan>
15	___ attribute specify the data member of scripting variable.	property	scope	parameter	value	**property**
16	The ____ tag evaluates if the variable contains the specified constant values.	<logic:mismatch>	<logic:notMatch>	<logic:contain>	<logic:match>	**<logic:match>**
11	The ____ tag evaluates if the variable not contains the specified constant values.	<logic:mismatch>	<logic:notMatch>	<logic:contain>	<logic:match>	**<logic:notMatch>**
12	The <logic:present> tag evaluates if the variable present in the applicable	Constant	attribute	scope	property	**scope**
13	The <logic:notPresent> tag evaluates if the variable ____ in the applicable scope.	not present	present	match	not match	**not present**
14	___ attribute is used to determine if the currently authenticated user has the specified name.	author	name	user	currentuser	**user**
15	Eg: Google Search Engine If the match tag with an attribute location= “start”, the output is	Goog	gine	error	search	**Goog**
16	Eg: Google Search Engine If the match tag with an attribute location= “end”, the output is	Goog	gine	error	search	**gine**
17	The logic tag is used for____ and object evaluation	enhancement	logic	control	decision making	**decision making**
18	Logic tags are used to manage conditional checking of _____.	output text	input text	tiles	form	**output text**
19	____ tag is used to iterate over a named collection which contains enumerator,iterator,map, array.	<logic:map>	<logic:iterate>	<logic:array>	<logic:enumerator>	<logic:iterate>
20	The <logic:empty/>tag evaluates the named scripting variable is __ or an empty string.	present	equal to null	notpresent	equal to zero	**equal to null**
21	The ___ tag evaluates if the named	<logic:null/>	<logic:notEmpty/>	<logic:empty/>	<logic:nullstring>	**<logic:notEmpty>**

	scripting variable is neither null or nor an empty string.					
22	The <logic:equal/> tag evaluates the __ variable equals the specified value	required	numeric	requested	string	**requested**
23	The <logic:notEqual> tag evaluates the variable ____ the constant value	equal	empty	notEqual	null	**notEqual**
24	<logic:forward> tag is used to forward control of the current request to previously identified element.	action	request	element	member	element
25	<logic:redirect> tag uses to redirect the current request to a ____	response	request	resource	element	**resource**
26	The <logic:greaterThan>tag evaluates if the variable is ____the constant value	equal to	greater than or equal to	less than	greater than	greater than
27	Offset attribute indicates the iteration ____	iteration	offset	begin	scope	**begin**
28	The <logic:lessEqual> tag evaluates if the variable is ___the constant value.	lessThan	lessEqual	less than or equal to	greater than or equal to	less than or equal to
29	The property attribute specify the data member of scripting variable.	property	class	variable	value	variable
30	The<logic:match>tag evaluates if the variable contains the specified constant ____	values	property	class	variable	values
31	___ attribute is used to append an HTML anchor to the end of a generated resource.	scope	name	value	anchor	**anchor**
32						**scope**
33						**not present**
34						**user**
35						**Goog**
36						**gine**

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act, 1956)

COIMBATORE – 641 021.

DEPARTMENT OF COMPUTER APPLICATIONS

STRUTS FRAME WORK – UNIT V

S.No.	Question	Option1	Option2	Option3	Option4	Answer
1	The ___ is on decision making and object evaluation	Html tag	Logic tag	Control tag	Generic tag	**Logic tag**
2	logic Tags should be prefixed with the string	htmltag	logic	xml	html	**logic**
3	The logic taglib contains ____ tags.	12	13	14	15	**14**
4	The _____ tag evaluates the named scripting variable is equal to null or an empty string.	<logic:empty/>	<logic:null/>	<logic:emptystring/>	<logic:nullstring/>	**<logic:empty/>**
5	The ___ tag evaluates if the named scripting variable is not equal to null or does not contain an empty string	<logic:null/>	<logic:notEmpty/>	<logic:empty/>	<logic:nullstring>	**<logic:notEmpty>**
6	The <logic:equal/> tag evaluates the variable equals the ___ value	constant	Integer	double	String	**constant**
7	The ____ tag evaluates the variable not equals the constant value	<logic:notEqual/>	<logic:nonequal>	<logic:notEqual/>	<logic:equal/>	**<logic:notEqual/>**
8	<logic:forward> tag is used to ____ control of the current request to previously identified element.	next	previous	forward	globalforward	forward
9	<logic:redirect> tag uses to redirect the current ____ to a resource.	response	request	action	element	**request**
10	The ___ tag evaluates if the variable is greater than the constant value	<logic:isgreaterThan>	<logic:greater>	<logic:isgreaterThan>	<logic:greaterThan>	<logic:greaterThan>
11	___ attribute specifies an HTTP Cookie to be used as a variable.	variable	cookie	property	value	**cookie**
12	___ attribute indicates the iteration begin.	iteration	offset	begin	scope	**offset**
13	<logic:iterate> tag is used to iterate over a named collection which contains	Enumerator,iterator,map, array	Enumerator,map, array	iterator,map, array	Enumerator,iterator, array	**Enumerator,iterator,map, array**

14	The ____ tag evaluates if the variable is less than or equal to the constant value.	<logic:lessThan>	<logic:islessEqual>	<logic:lessEqual>	<logic:islessThan>	<logic:lessEqual>
15.	<logic:lessThan> tag evaluates if the is less than the constant value.	<logic:lessEqual>	<logic:lessThan>	<logic:islessThan>	<logic:islessEqual>	<logic:lessThan>
15	___ attribute specify the data member of scripting variable.	property	scope	parameter	value	**property**
16	The ____ tag evaluates if the variable contains the specified constant values.	<logic:mismatch>	<logic:notMatch>	<logic:contain>	<logic:match>	**<logic:match>**
11	The ____ tag evaluates if the variable not contains the specified constant values.	<logic:mismatch>	<logic:notMatch>	<logic:contain>	<logic:match>	**<logic:notMatch>**
12	The <logic:present> tag evaluates if the variable present in the applicable	Constant	attribute	scope	property	**scope**
13	The <logic:notPresent> tag evaluates if the variable ____ in the applicable scope.	not present	present	match	not match	**not present**
14	___ attribute is used to determine if the currently authenticated user has the specified name.	author	name	user	currentuser	**user**
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						

Reg No. ……

[**18CAU301]**

Karpagam Academy Of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 64 021
BCA Degree Examination
(For the candidates admitted from 2018 onwards)
THIRD SEMESTER
First Internal Exam July 2019
DATA STRUCTURES

Time: 2 Hours **Maximum: 50 Marks**
Date&Session: **CLASS: II BCA(A&B)**

Part – A (20 X 1 = 20 Marks)

Answer ALL Questions

1.Who developed the struts project?
a) Craig McClanahan b) Dennis Ritchie c) Tim Bernerslee d) Charles

2. _____ is the container that holds the components of a Web application.

a) Folder **b) Directory Structure** c) Files d) Frame

3. Which is a root directory of Tim Bernerslee the web application?
a) /wileyapp b) /wileyapp/WEB−INF
c) / wileyapp/WEB−INF/classes d) / wileyapp/WEB−INF/lib

4. This directory is where servlet and utility classes are located.
a) /wileyapp b) /wileyapp/WEB−INF
c) / wileyapp/WEB−INF/classes d) / wileyapp/WEB−INF/lib

5. This directory contains Java Archive (JAR) files that the Web application is dependent on.
a) /wileyapp b) /wileyapp/WEB−INF
c) / wileyapp/WEB−INF/classes **d) / wileyapp/WEB−INF/lib**

6. This is where your Web application deployment descriptor is located.
a) /wileyapp **b) /wileyapp/WEB−INF**
c) / wileyapp/WEB−INF/classes d) / wileyapp/WEB−INF/lib

7. Which is the backbone of all Web applications? is its
a) Folder b) Directory Structure **c) Deployment Descriptor** d) Frame

8. The standard packaging format for a Web application is a
a) WAR b) JAR **c) JPEG** d) PNG

9. All _____ files are stored /wileyapp.
a) JSP b) HTML **c) JSP & HTML** d) JAR
10. The _____file describes all of the components in the Web application.
a) web.xml b) HTML c) JSP & HTML d) JAR
11. The performs its logic on the Model components.
a) Action Servlet b) Action class c) View d) ActiveServlet

12. Which command is used to start the Tomcat server
a) \bin\startup.bat b) \bin\tc\startup.bat c) \bin\begin.bat d) \bin\start.bat

13. The _____ method services all requests received from a client using a simple request/response pattern.
a) Perform() **b) Service()** c) Destroy() d) init()

14. A ServletContext is an object that is defined in the package.
a) Java.awt b) java.util c) java.io **d) javax.servlet**

15. _____are the JSP components that bring all the JSP elements together.
a) Servlets b) Scripts c) Java Script **d) Scriptlets**

16. The directive is used to insert text and/or code at JSP translation time.
a) include b) taglib c) session d) info

17. Which tag enables a JSP author to generate the required HTML, using the appropriate client−browser independent constructs?
a) <jsp:setProperty> b) <jsp:forward> **c) <jsp:plugin>** d) <jsp:param>

18. Which sets the value of a bean's property?
a) <jsp:setProperty> b) <jsp:forward> c) <jsp:plugin> d) <jsp:param>

19. Which is the Controller component?
a) org.apache.struts.action.ActionServlet b) org.apache.struts.action.Action
c) org.apache.struts.action. Servlet d) org.apache.struts.action.Activeservlet

20. All JSPs should be deployed to a Struts application by using a element.
a) <setProperty> **b) <forward>** c) <plugin> d) <param>

Part – B (3X 2= 6 Marks)
Answer ALL Questions

21. Name the common components of Web Application.

Java Servlets, JSP (Java Server Pages), Custom Tags and Message Resources

22. Define Directory Structure.

Directory	Contains
/wileyapp	This is the root directory of the Web application. All JSP and HTML files are stored here.
/wileyapp/WEB−INF	This directory contains all resources related to the application that are not in the document root of the application. This is where your Web application deployment descriptor is located. You should note that the WEB−INF directory is not part of the public document. No files contained in this directory can be served directly to a client.
/ wileyapp/WEB−INF/classes	This directory is where servlet and utility classes are located.
/ wileyapp/WEB−INF/lib	This directory contains Java Archive (JAR) files that the Web application is dependent on.

23. Write any 2 usages of JSP.

JavaServer Pages, or JSPs, are a simple but powerful technology used most often to generate dynamic HTML on the server side. JSPs are a direct extension of Java servlets designed to let the developer embed Java logic directly into a requested document.

.

Part – C(3X 8= 24 Marks)
Answer ALL Questions

24.(a) Explain the struts implementation of the MVC.

The Struts Framework models its server−side implementation of the MVC using a combination of JSPs, custom JSP tags, and Java servlets. In this section, we briefly describe how the Struts Framework maps to each component of the MVC. When we have completed this discussion, we will have drawn a portrait similar to Figure 1.1.

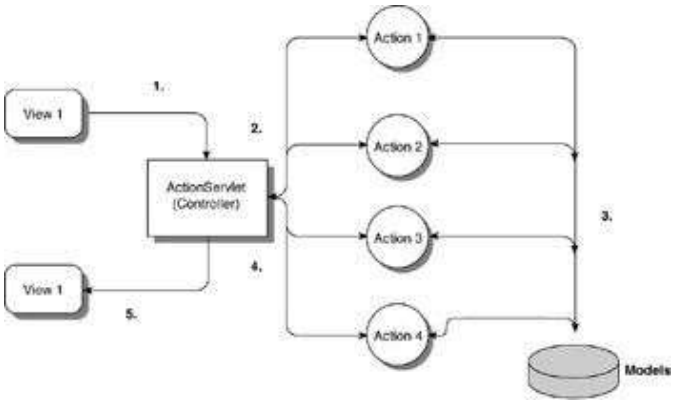


Figure 1.1: The Struts implementation of the MVC.

Figure 1.1 depicts the route that most Struts application requests follow. This process can be broken down into five basic steps. Following these steps is a description of the ActionServlet and Action classes.

1. A request is made from a previously displayed View.

2. The request is received by the ActionServlet, which acts as the Controller, and the ActionServlet looks up the requested URI in an XML file (described in Chapter 3, "Getting Started with Struts"), and determines the name of the Action class that will perform the necessary business logic.
3. The Action class performs its logic on the Model components associated with the application.
4. Once the Action has completed its processing, it returns control to the ActionServlet. As part of the return, the Action class provides a key that indicates the results of its processing. The ActionServlet uses this key to determine where the results should be forwarded for presentation.
5. The request is complete when the ActionServlet responds by forwarding the request to the View that was linked to the returned key, and this View presents the results of the Action.

The Model

The Struts Framework does not provide any specialized Model components; therefore, we will not dedicate an entire chapter to the Model component. Instead, we will reference Model components as they fit into each example.

The View

Each View component in the Struts Framework is mapped to a single JSP that can contain any combination of Struts custom tags.

The Controller

The Controller component of the Struts Framework is the backbone of all Struts Web applications. It is implemented using a servlet named org.apache.struts.action.ActionServlet. This servlet receives all requests from clients, and delegates control of each request to a user−defined org.apache.struts.action.Action class. The ActionServlet delegates control based on the URI of the incoming request. Once the Action class has completed its processing, it returns a key to the ActionServlet, which is then used by the ActionServlet to determine the View that will present the results of the Action's processing. The ActionServlet is similar to a factory that creates Action objects to perform the actual business logic of the application. The Controller of the Struts Framework is the most important component of the Struts MVC.

(OR)

(b) Discuss about the lifecycle of a Servlet.

The life cycle of a Java servlet follows a very logical sequence. The interface that declares the life−cycle methods is the javax.servlet.Servlet interface. These methods are the init(), the service(), and the destroy() methods. This sequence can be described in a simple three−step process:

1. A servlet is loaded and initialized using the init() method. This method will be called when a servlet is preloaded or upon the first request to this servlet.
2. The servlet then services zero or more requests. The servlet services each request using the service() method.
3. The servlet is then destroyed and garbage collected when the Web application containing the servlet shuts down. The method that is called upon shutdown is the destroy() method.

init() Method

The init() method is where the servlet begins its life. This method is called immediately after the servlet is instantiated. It is called only once. The init() method should be used to create and initialize the resources that it will be using while handling requests. The init() method's signature is defined as follows:

```
public void init(ServletConfig config) throws ServletException;
```

The init() method takes a ServletConfig object as a parameter. This reference should be stored in a member variable so that it can be used later. A common way of doing this is to have the init() method call super.init() and pass it the ServletConfig object.

The init() method also declares that it can throw a ServletException. If for some reason the servlet cannot initialize the resources necessary to handle requests, it should throw a ServletException with an error message signifying the problem.

service() Method

The service() method services all requests received from a client using a simple request/response pattern. The service() method's signature is shown here:

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

The service() method takes two parameters:

- A ServletRequest object, which contains information about the service request and encapsulates information provided by the client
- A ServletResponse object, which contains the information returned to the client

You will not usually implement this method directly, unless you extend the GenericServlet abstract class. The most common implementation of the service() method is in the HttpServlet class. The HttpServlet class implements the Servlet interface by extending GenericServlet. Its service() method supports standard HTTP/1.1 requests by determining the request type and calling the appropriate method.

destroy() Method

This method signifies the end of a servlet's life. When a Web application is shut down, the servlet's destroy() method is called. This is where all resources that were created in the init() method should be cleaned up. The following code snippet contains the signature of the destroy() method:

```
public void destroy();
```

25.(a) Explain the components of JavaServer Page.

The Components of a JavaServer Page

This section discusses the components of a JSP, including directives, scripting, implicit objects, and standard actions.

JSP Directives

JSP directives are JSP elements that provide global information about a JSP page. An example would be a directive that included a list of Java classes to be imported into a JSP. The syntax of a JSP directive follows:

```
<%@ directive {attribute="value"} %>
```

Three possible directives are currently defined by the JSP specification v1.2: page, include, and taglib. These directives are defined in the following sections.

The page Directive

The page directive defines information that will globally affect the JSP containing the directive. The syntax of a JSP page directive is

```
<%@ page {attribute="value"} %>

<%@ page import="java.util.*" %>
```

The include Directive

The include directive is used to insert text and/or code at JSP translation time. The syntax of the include directive is shown in the following code snippet:

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

The file attribute can reference a normal text HTML file or a JSP file, which will be evaluated at translation time. This resource referenced by the file attribute must be local to the Web application that contains the include directive. Here's a sample include directive:

```
<%@ include file="header.jsp" %>
```

The taglib Directive

The taglib directive states that the including page uses a custom tag library, uniquely identified by a URI and associated with a prefix that will distinguish each set of custom tags to be used in the page.

The syntax of the taglib directive is as follows:

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

The taglib attributes are described in Table 2.3.

JSP Scripting

Scripting is a JSP mechanism for directly embedding Java code fragments into an HTML page. Three scripting language components are involved in JSP scripting. Each component has its appropriate location in the generated servlet. This section examines these components.

Declarations

JSP declarations are used to define Java variables and methods in a JSP. A JSP declaration must be a complete declarative statement.

JSP declarations are initialized when the JSP page is first loaded. After the declarations have been initialized, they are available to other declarations, expressions, and scriptlets within the same JSP.

When this document is initially loaded, the JSP code is converted to servlet code and the name declaration is placed in the declaration section of the generated servlet. It is now available to all other JSP components in the JSP.

Expressions

JSP expressions are JSP components whose text, upon evaluation by the container, is replaced with the resulting value of the container evaluation. JSP expressions are evaluated at request time, and the result is inserted at the expression's referenced position in the JSP file. If the resulting expression cannot be converted to a string, then a translation−time error will occur. If the conversion to a string cannot be detected during translation, a ClassCastException will be thrown at request time.

destroy()

The syntax of a JSP expression is as follows:

```
<%= expression %>
```

A code snippet containing a JSP expression is shown here:

```
Hello <B><%= getName() %></B>
```

Here is a sample JSP document containing a JSP expression:

```
<HTML>

<BODY>

<%! public String getName() { return "Bob"; } %>

Hello <B><%= getName() %></B>


</BODY>

</HTML>
```

Scriptlets

Scriptlets are the JSP components that bring all the JSP elements together. They can contain almost any coding statements that are valid for the language referenced in the language directive. They are executed at request time, and they can make use of all the JSP components. The syntax for a scriptlet is as follows:

```
<% scriptlet source %>
```

When JSP scriptlet code is converted into servlet code, it is placed into the generated servlet's service() method. The following code snippet contains a simple JSP that uses a scripting element to print the text "Hello Bob" to the requesting client:

```
<HTML>
<BODY>
<% out.println("Hello Bob"); %>
</BODY>
</HTML>
```

You should note that while JSP scriplet code can be very powerful, composing all your JSP logic using scriptlet code can make your application difficult to manage. This problem led to the creation of custom tag libraries.

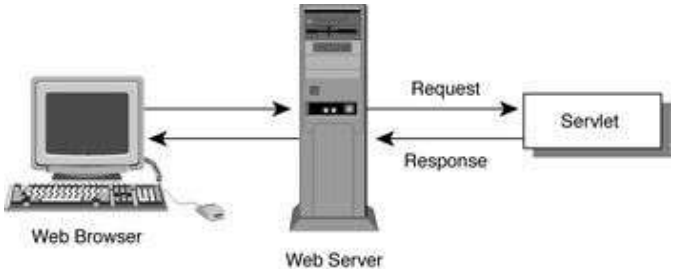
(OR)

(b) Explain the Java Servlet Architecture.

The Java Servlet Architecture

A *Java servlet* is a platform−independent Web application component that is hosted in a JSP/servlet container. Servlets cooperate with Web clients by means of a request/response model managed by a JSP/servlet container. Figure 2.1 depicts the execution of a Java servlet.

destroy()



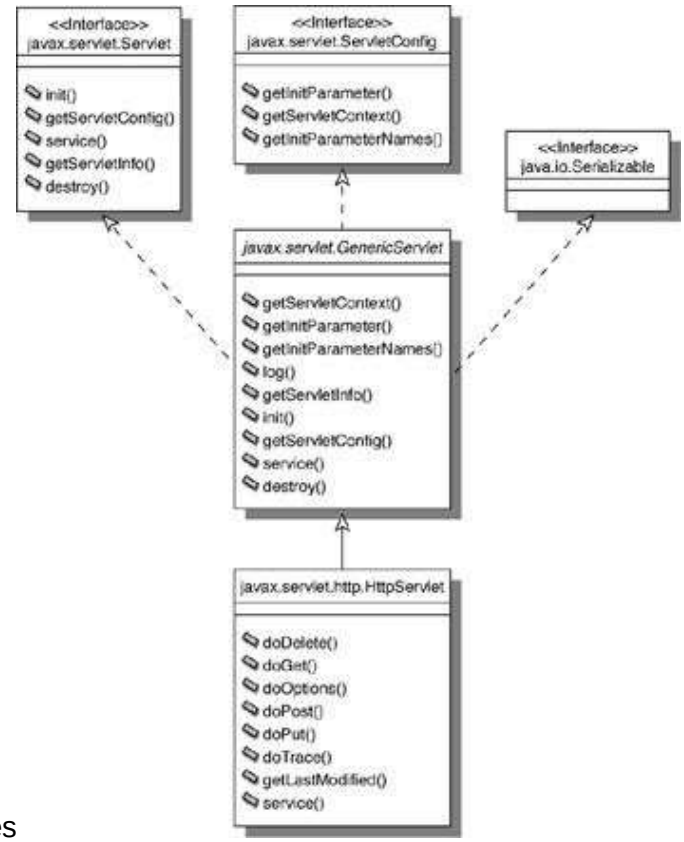
The execution of a Java servlet.

Two packages make up the servlet architecture: javax.servlet and javax. servlet.http. The first of these, the javax.servlet package, contains the generic interfaces and classes that are implemented and extended by all servlets. The second, the javax.servlet.http package, contains all servlet classes that are HTTP protocol−specific. An example of this would be a simple servlet that responds using HTML.

At the heart of this architecture is the interface javax.servlet.Servlet. It is the base class interface for all servlets. The Servlet interface defines five methods. The three most important of these methods are the

- init() method, which initializes a servlet
- service() method, which receives and responds to client requests
- destroy() method, which performs cleanup

These are the servlet life−cycle methods. We will describe these methods in a subsequent section. All servlets must implement the Servlet interface, either directly or through inheritance. Figure 2.2 is an object model that gives you a very high−level view of the servlet framework.



The GenericServlet and HttpServlet Classes

destroy()
: A simple object model showing the servlet framework.

26.(a) Explain the controller in JSP.

In this chapter, we dig further into the Controller components of the Struts framework. We begin by looking at three distinct Struts Controller components, including the ActionServlet class, the Action class, Plugins, and the RequestProcesser.

The goal of this chapter is to provide you with a solid understanding of the Struts Controller components, and how they can be used and extended to create a robust and easily extended Web application.

The ActionServlet Class

The org.apache.struts.action.ActionServlet is the backbone of all Struts applications. It is the main Controller component that handles client requests and determines which org.apache.struts.action.Action will process each received request. It acts as an Action factory by creating specific Action classes based on the user's request.

While the ActionServlet sounds as if it might perform some extraordinary magic, it is a simple servlet. Just like any other HTTP servlet, it extends the class javax.servlet.http.HttpServlet and implements each of the HttpServlet's life−cycle methods, including the init(), doGet(), doPost(), and destroy() methods.

The special behavior begins with the ActionServlet's process() method. The process() method is the method that handles all requests. It has the following method signature:

```
protected void process(HttpServletRequest request,
  HttpServletResponse response);
```

When the ActionServlet receives a request, it completes the following steps:

1. The doPost() or doGet() methods receive a request and invoke the process() method.
2. The process() method gets the current RequestProcessor, which is discussed in the final section of this chapter, and invokes its process() method.
3. The RequestProcessor.process() method is where the current request is actually serviced. The RequestProcessor.process() method retrieves, from the struts−config.xml file, the <action> element that matches the path submitted on the request. It does this by matching the path passed in the <html:form /> tag's action element to the <action> element with the same path value. An example of this match is shown below:
4. When the RequestProcessor.process() method has a matching <action>, it looks for a <form−bean> entry that has a name attribute that matches the <action> element's name attribute. The following code snippet contains a sample match:
5. When the RequestProcessor.process() method knows the fully qualified name of the FormBean, it creates or retrieves a pooled instance of the ActionForm named by the <form−bean> element's type attribute, and populates its data members with the values submitted on the request.
6. After the ActionForm's data members are populated, the RequestProcessor.process() method calls the ActionForm.validate() method, which checks the validity of the submitted values.
7. At this point, the RequestProcessor.process() method knows all that it needs to know, and it is time to actually service the request. It does this by retrieving the fully qualified name of the Action class from the <action> element's type attribute, creating or retrieving the named class, and calling the

Action.execute() method. We will look at this method in the section titled "The Action Class," later in this chapter.

8. When the Action class returns from its processing, its execute() method returns an ActionForward object that is used to determine the target of this transaction. The RequestProcessor.process() method resumes control, and the request is then forwarded to the determined target.
9. At this point, the ActionServlet instance has completed its processing for this request and is ready to service future requests.

The Action Class

The second component of a Struts Controller is the org.apache.struts.action. Action class. As we stated in Chapter 3, the Action class must and will be extended for each specialized Struts function in your application. The collection of the Action classes that belong to your Struts application is what defines your Web application.

To begin our discussion of the Action class, we must first look at some of the Action methods that are more commonly overridden or leveraged when creating an extended Action class. The following sections describe five of these methods.

The execute() Method

The execute() method is where your application logic begins. It is the method that you need to override when defining your own Actions. The Struts framework defines two execute() methods.
The first execute() implementation is used when you are defining custom Actions that are not HTTP−specific. This implementation of the execute() method would be analogous to the javax.serlvet.GenericServlet class.

You will notice that this method receives, as its third and fourth parameter, a HttpServletRequest and a HttpServletResponse object, as opposed to the previously listed execute() method. This implementation of the execute() method is the implementation that you will most often extend. Table 4.2 describes all of the parameters of the Action.execute() method.

Struts Plugins

Struts Plugins are modular extensions to the Struts Controller. They have been introduced in Struts 1.1, and are defined by the org.apache.struts.action.Plugin interface. Struts Plugins are useful when allocating resources or preparing connections to databases or even JNDI resources. We will look at an example of loading application properties on startup later in this section.
This interface, like the Java Servlet architecture, defines two methods that must be implemented by all used−defined Plugins: init() and destroy(). These are the life−cycle methods of a Struts Plugin.

init()

The init() method of a Struts Plugin is called whenever the JSP/Servlet container starts the Struts Web application containing the Plugin. This method is convenient when initializing resources that are important to their hosting applications. As you will have noticed, the init() method receives an ApplicationConfig parameter when invoked. This object provides access to the configuration information describing a Struts application. The init() method marks the beginning of a Plugin's life.

destroy()

The destroy() method of a Struts Plugin is called whenever the JSP/Servlet container stops the Struts Web application containing the Plugin. It has a method signature as follows:

```
public void destroy();
```

This method is convenient when reclaiming or closing resources that were allocated in the Plugin.init() method. This method marks the end of a Plugin's life.

The RequestProcessor

As we stated previously, the org.apache.struts.action.RequestProcessor contains the logic that the Struts controller performs with each servlet request from the container. The RequestProcessor is the class that you will want to override when you want to customize the processing of the ActionServlet.

(OR)

(b) Explain the view in JSP.

The Views

In this chapter, we examine the View component of the Struts framework. Some of the topics that we discuss are using tags from Struts tag libraries, using ActionForms, and deploying Views to a Struts application.

The goal of this chapter is to give you an understanding of the Struts View and the components that can be leveraged to construct the View.

Building a Struts View

As we discussed in Chapter 1, "Introducing the Jakarta Struts Project and Its Supporting Components," the Struts View is represented by a combination of JSPs, custom tag libraries, and optional ActionForm objects. In the sections that follow, we examine each of these components and how they can be leveraged.

At this point, you should have a pretty good understanding of what JSPs are and how they can be used. We can now focus on how JSPs are leveraged in a Struts application.

JSPs in the Struts framework serve two main functions. The first of these functions is to act as the presentation layer of a previously executed Controller Action. This is most often accomplished using a set of custom tags that are focused around iterating and retrieving data forwarded to the target JSP by the Controller Action. This type of View is not Struts−specific, and does not warrant special attention.

The second of these functions, which is very much Struts−specific, is to gather data that is required to perform a particular Controller Action. This is done most often with a combination of tag libraries and ActionForm objects. This type of View contains several Struts−specific tags and classes, and is therefore the focus of this chapter.