

Web Component  
Development With Java<sup>TM</sup>  
Technology  
SL-314

Student Guide



Sun Microsystems, Inc.  
UBRM05-104  
500 Eldorado Blvd.  
Broomfield, CO 80021  
U.S.A.

Revision A.1



Copyright 2002 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved.

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

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun Logo, the Duke Logo, EJB, Enterprise JavaBeans, JavaBeans, Javadoc, JDK, Java Naming and Directory Interface, JavaScript, Java Servlet, Java 2 Platform, Enterprise Edition, Java 2 Platform, Standard Edition, Java 2 SDK, JDBC, J2EE, J2SE, JSP, JVM, Solaris, and Sun Enterprise are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Netscape and Netscape Navigator is a trademark or registered trademark of Netscape Communications Corporation.

INTERACTIVE UNIX is a registered trademark of INTERACTIVE Systems Corporation. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015 (b)(6/95) and DFAR 227.7202-3(a).

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

**THIS MANUAL IS DESIGNED TO SUPPORT AN INSTRUCTOR-LED TRAINING (ILT) COURSE AND IS INTENDED TO BE USED FOR REFERENCE PURPOSES IN CONJUNCTION WITH THE ILT COURSE. THE MANUAL IS NOT A STANDALONE TRAINING TOOL. USE OF THE MANUAL FOR SELF-STUDY WITHOUT CLASS ATTENDANCE IS NOT RECOMMENDED.**



Please  
Recycle



Adobe PostScript

Copyright 2002 Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, le logo Duke, EJB, Enterprise JavaBeans, JavaBeans, Javadoc, JDK, Java Naming and Directory Interface, JavaScript, Java Servlet, Java 2 Platform, Enterprise Edition, Java 2 Platform, Standard Edition, Java 2 SDK, JDBC, J2EE, J2SE, JSP, JVM, Solaris, et Sun Enterprise sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Netscape et Netscape Navigator est une marque de Netscape Communications Corporation aux Etats-Unis et dans d'autres pays. in the United States and other countries.

INTERACTIVE est une marque de INTERACTIVE Systems Corporation. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



Please  
Recycle



Adobe PostScript™

# Table of Contents

---

<b>About This Course .....</b>	<b>xix</b>
Course Goal .....	xix
Learning Objectives .....	xx
Module-by-Module Overview .....	xi
Topics Not Covered .....	xxii
How Prepared Are You?.....	xxiii
How to Learn From This Course .....	xxiii
Introductions .....	xxiv
How to Use Course Materials .....	xxv
Conventions .....	xxvi
Icons .....	xxvi
Typographical Conventions .....	xxvii
Additional Conventions.....	xxviii
<b>Introduction to Web Application Technologies.....</b>	<b>1-1</b>
Objectives .....	1-1
Relevance.....	1-2
Additional Resources .....	1-2
Internet Services .....	1-3
The Internet Is a Network of Networks.....	1-3
Networking Protocol Stack.....	1-5
Client-Server Architecture .....	1-7
Hypertext Transfer Protocol.....	1-8
Web Browsers and Web Servers .....	1-8
HTTP Client-Server Architecture .....	1-9
The Structure of a Web Site .....	1-10
Web Applications.....	1-12
CGI Programs on the Web Server.....	1-12
Execution of CGI Programs .....	1-13
Advantages and Disadvantages of CGI Programs .....	1-14
Java Servlets .....	1-15
Servlets on the Web Server .....	1-15
Execution of Java Servlets.....	1-16
Advantages and Disadvantages of Java Servlets .....	1-18

---

Template Pages.....	1-19
Other Template Page Technologies.....	1-20
JavaServer Pages Technology.....	1-21
Advantages and Disadvantages of JavaServer Pages.....	1-22
The Model 2 Architecture .....	1-22
The J2EE Platform .....	1-24
An Example of J2EE Architecture.....	1-25
Job Roles .....	1-26
Web Application Migration.....	1-26
Summary .....	1-28
<b>Developing a Simple Servlet.....</b>	<b>2-1</b>
Objectives .....	2-1
Relevance.....	2-2
Generic Internet Services .....	2-3
The NetServer Architecture.....	2-3
The Generic Servlets API .....	2-5
The Generic HelloServlet Class .....	2-6
HTTP Servlets .....	2-7
Hypertext Transfer Protocol.....	2-7
HTTP GET Method .....	2-8
HTTP Request.....	2-8
The HttpServletRequest API .....	2-9
HTTP Response .....	2-11
The HttpServletResponse API .....	2-12
Web Container Architecture.....	2-14
The Web Container .....	2-14
Sequence Diagram of a HTTP GET Request .....	2-15
Request and Response Process .....	2-16
The HTTP Servlet API.....	2-19
The HTTP HelloServlet Class .....	2-20
Deploying a Servlet.....	2-21
Installing, Configuring, and Running the Web Container .....	2-21
Deploying the Servlet to the Web Container .....	2-21
Activating the Servlet in a Web Browser.....	2-22
Summary .....	2-23
Certification Exam Notes .....	2-24
<b>Developing a Simple Servlet That Uses HTML Forms.....</b>	<b>3-1</b>
Objectives .....	3-1
Relevance.....	3-2
Additional Resources .....	3-2
HTML Forms .....	3-3
The FORM Tag .....	3-4
HTML Form Components .....	3-5
Input Tags .....	3-6

---

Textfield Component.....	3-6
Submit Button Components .....	3-7
Reset Button Component.....	3-8
Checkbox Component.....	3-9
Radio Button Component .....	3-10
Password Component .....	3-11
Hidden Field Component.....	3-11
The SELECT Tag.....	3-12
The TEXTAREA Tag.....	3-13
Form Data in the HTTP Request.....	3-14
HTTP GET Method Request.....	3-15
HTTP POST Method Request.....	3-16
To GET or to POST? .....	3-18
How Servlets Access Form Data.....	3-19
The Servlet API.....	3-19
The FormBasedHello Servlet.....	3-20
Summary .....	3-22
Certification Exam Notes .....	3-23

<b>Developing a Web Application Using a Deployment Descriptor.....</b>	<b>4-1</b>
Objectives .....	4-1
Problems With Simple Servlets.....	4-2
Problems With Deploying in One Place .....	4-2
Multiple Web Applications .....	4-3
Web Application Context Name.....	4-4
Problems With Servlet Naming .....	4-5
Solutions to Servlet Naming Problems .....	4-6
Problems Using Common Services .....	4-7
Developing a Web Application Using a Deployment Descriptor.....	4-8
The Deployment Descriptor .....	4-8
A Development Environment .....	4-10
The Deployment Environment .....	4-11
The Web Archive (WAR) File Format.....	4-16
Summary .....	4-17
Certification Exam Notes .....	4-18

<b>Configuring Servlets .....</b>	<b>5-1</b>
Objectives .....	5-1
Relevance.....	5-2
Servlet Life Cycle Overview .....	5-3
The init Life Cycle Method .....	5-3
The service Life Cycle Method.....	5-4
The destroy Life Cycle Method.....	5-4
Servlet Configuration .....	5-5
The ServletConfig API .....	5-5
Initialization Parameters .....	5-6
Summary .....	5-9

---

Certification Exam Notes .....	5-10
<b>Sharing Resources Using the Servlet Context .....</b>	<b>6-1</b>
Objectives .....	6-1
Relevance.....	6-2
The Web Application.....	6-3
Duke's Store Web Application.....	6-3
The ServletContext API.....	6-4
Context Initialization Parameters .....	6-5
Access to File Resources.....	6-6
Writing to the Web Application Log File .....	6-6
Accessing Shared Runtime Attributes .....	6-7
The Web Application Life Cycle .....	6-8
Duke's Store Example .....	6-9
Configuring Servlet Context Listeners.....	6-11
Summary .....	6-12
Certification Exam Notes .....	6-13
<b>Developing Web Applications Using the MVC Pattern.....</b>	<b>7-1</b>
Objectives .....	7-1
Activities of a Web Application .....	7-2
The Soccer League Example .....	7-3
Page Flow of the Soccer League Example .....	7-4
Architecture of the Soccer League Example.....	7-6
Activity Diagram of the Soccer League Example.....	7-6
Discussion of the Simple Web Application.....	7-8
Model-View-Controller for a Web Application.....	7-9
Sequence Diagram of MVC in the Web Tier .....	7-10
Soccer League Application: The Domain Model.....	7-11
Soccer League Application: The Services Model.....	7-11
Soccer League Application: The Big Picture .....	7-13
Soccer League Application: The Controller .....	7-14
Soccer League Application: The Views.....	7-15
The Request Scope .....	7-16
Summary .....	7-17
Certification Exam Notes .....	7-18
<b>Developing Web Applications Using Session Management.....</b>	<b>8-1</b>
Objectives .....	8-1
Relevance.....	8-2
Additional Resources .....	8-2
HTTP and Session Management.....	8-3
Sessions in a Web Container .....	8-3
Web Application Design Using Session Management .....	8-4
Example: Registration Use Case .....	8-4
Example: Multiple Views for Registration .....	8-5
Example: Enter League Form .....	8-6

---

Web Application Development Using Session Management .....	8-8
The Session API.....	8-8
Retrieving the Session Object .....	8-9
Storing Session Attributes.....	8-10
Accessing Session Attributes.....	8-12
Destroying the Session .....	8-15
Session Management Using Cookies .....	8-17
The Cookie API .....	8-17
Using Cookies.....	8-18
Using Cookies for Session Management .....	8-19
Session Management Using URL-Rewriting .....	8-21
Implications of Using URL-Rewriting .....	8-22
Guidelines for Working With Sessions .....	8-23
Summary .....	8-24
Certification Exam Notes .....	8-25
<b>Handling Errors in Web Applications .....</b>	<b>9-1</b>
Objectives .....	9-1
Additional Resources .....	9-2
The Types of Web Application Errors .....	9-3
HTTP Error Codes .....	9-3
Generic HTTP Error Page .....	9-4
Servlet Exceptions.....	9-5
Generic Servlet Error Page.....	9-6
Using Custom Error Pages .....	9-7
Creating Error Pages.....	9-7
Declaring HTTP Error Pages .....	9-8
Example HTTP Error Page .....	9-8
Declaring Servlet Exception Error Pages.....	9-9
Example Servlet Error Page .....	9-10
Developing an Error Handling Servlet.....	9-11
Programmatic Exception Handling.....	9-14
Exception Handling Servlet Declarations .....	9-16
Trade-offs for Declarative Exception Handling .....	9-17
Trade-offs for Programmatic Exception Handling .....	9-18
Logging Exceptions .....	9-19
Summary .....	9-20
Certification Exam Notes .....	9-21
<b>Configuring Web Application Security.....</b>	<b>10-1</b>
Objectives .....	10-1
Relevance.....	10-2
Additional Resources .....	10-2
Web Security Issues .....	10-3
Authentication.....	10-3
Authorization .....	10-4

---

Maintaining Data Integrity .....	10-5
Access Tracking.....	10-5
Dealing With Malicious Code .....	10-6
Dealing With Web Attacks .....	10-6
Declarative Authorization .....	10-7
Web Resource Collection .....	10-7
Declaring Security Roles .....	10-8
Security Realms .....	10-9
Declarative Authentication.....	10-10
BASIC Authentication.....	10-11
Summary .....	10-15
Certification Exam Notes .....	10-16
<b>Understanding Web Application Concurrency Issues .....</b>	<b>11-1</b>
Objectives .....	11-1
Additional Resources .....	11-2
The Need for Servlet Concurrency Management.....	11-3
Concurrency Management Example.....	11-4
Attributes and Scope .....	11-8
Local Variables .....	11-9
Instance Variables .....	11-10
Class Variables .....	11-11
Request Scope .....	11-12
Session Scope .....	11-13
Application Scope .....	11-14
The Single Threaded Model .....	11-15
The SingleThreadModel Interface .....	11-15
How the Web Container Might Implement the Single Threaded Model .....	11-16
STM and Concurrency Management .....	11-17
Recommended Approaches to Concurrency Management .....	11-19
Summary .....	11-20
Certification Exam Notes .....	11-21
<b>Integrating Web Applications With Databases .....</b>	<b>12-1</b>
Objectives .....	12-1
Relevance.....	12-2
Additional Resources .....	12-2
Database Overview.....	12-3
The JDBC API .....	12-4
Designing a Web Application That Integrates With a Database .....	12-5
Domain Objects .....	12-5
Database Tables.....	12-6
Data Access Object Pattern .....	12-7
Advantages of the DAO Pattern.....	12-9

---

Developing a Web Application That Uses a Connection Pool.....	12-10
Connection Pool .....	12-10
Storing the Connection Pool in a Global Name Space .....	12-12
Accessing the Connection Pool.....	12-13
Initializing the Connection Pool .....	12-16
Developing a Web Application That Uses a Data Source.....	12-17
Summary .....	12-18
Certification Exam Notes .....	12-19
<b>Developing JSP™ Pages .....</b>	<b>13-1</b>
Objectives .....	13-1
Relevance.....	13-2
Additional Resources .....	13-2
JavaServer Page Technology .....	13-3
How a JSP Page Is Processed.....	13-5
Developing and Deploying JSP Pages .....	13-7
JSP Scripting Elements .....	13-8
Comments .....	13-9
Directive Tag.....	13-10
Declaration Tag .....	13-11
Scriptlet Tag .....	13-12
Expression Tag .....	13-13
Implicit Variables .....	13-14
The page Directive.....	13-15
JSP Page Exception Handling.....	13-17
Declaring an Error Page .....	13-18
Developing an Error Page.....	13-19
Behind the Scenes.....	13-20
Debugging a JSP Page .....	13-23
Summary .....	13-26
Certification Exam Notes .....	13-27
<b>Developing Web Applications Using the Model 1 Architecture.....</b>	<b>14-1</b>
Objectives .....	14-1
Additional Resources .....	14-2
Designing With Model 1 Architecture .....	14-3
Guest Book Form.....	14-4
Guest Book Components .....	14-5
Guest Book Page Flow.....	14-6
What Is a JavaBeans Component? .....	14-7
The GuestBookService JavaBeans Component.....	14-8
Developing With Model 1 Architecture .....	14-9
The Guest Book HTML Form.....	14-9
JSP Standard Actions .....	14-10

---

Creating a JavaBeans Component in a JSP Page .....	14-10
Initializing the JavaBean Component .....	14-11
Control Logic in the Guest Book JSP Page .....	14-13
Accessing a JavaBeans Component in a JSP Page .....	14-14
Beans and Scope .....	14-15
Review of the <code>jsp:useBean</code> Action .....	14-15
Summary .....	14-17
Certification Exam Notes .....	14-18
<b>Developing Web Applications Using the Model 2 Architecture .....</b>	<b>15-1</b>
Objectives .....	15-1
Relevance.....	15-2
Designing With Model 2 Architecture .....	15-3
The Soccer League Example Using Model 2 Architecture .....	15-4
Sequence Diagram of Model 2 Architecture .....	15-6
Developing With Model 2 Architecture .....	15-8
Controller Details .....	15-9
Request Dispatchers .....	15-14
View Details.....	15-15
Summary .....	15-16
Certification Exam Notes .....	15-17
<b>Building Reusable Web Presentation Components .....</b>	<b>16-1</b>
Objectives .....	16-1
Complex Page Layouts.....	16-2
What Does a Fragment Look Like? .....	16-3
Organizing Your Presentation Fragments.....	16-4
Including JSP Page Fragments .....	16-5
Using the <code>include</code> Directive.....	16-5
Using the <code>jsp:include</code> Standard Action .....	16-6
Using the <code>jsp:param</code> Standard Action.....	16-7
Summary .....	16-9
Certification Exam Notes .....	16-10
<b>Developing JSP Pages Using Custom Tags .....</b>	<b>17-1</b>
Objectives .....	17-1
Relevance.....	17-2
Additional Resources .....	17-2
Job Roles Revisited.....	17-3
Introducing Custom Tag Libraries .....	17-4
Contrasting Custom Tags and Scriptlet Code .....	17-4
Developing JSP Pages Using Custom Tags.....	17-5
What Is a Custom Tag Library? .....	17-6
Custom Tag Syntax Rules .....	17-6
Example Tag Library: Soccer League .....	17-8

---

Developing JSP Pages Using a Custom Tag Library .....	17-10
Using an Empty Custom Tag .....	17-11
Using a Conditional Custom Tag .....	17-12
Using an Iterative Custom Tag .....	17-13
Summary .....	17-14
Certification Exam Notes .....	17-15
<b>Developing a Simple Custom Tag.....</b>	<b>18-1</b>
Objectives .....	18-1
Overview of Tag Handlers .....	18-2
Fundamental Tag Handler API.....	18-2
Tag Handler Life Cycle .....	18-3
Tag Library Relationships.....	18-6
Developing a Tag Handler Class .....	18-8
The getReqParam Tag .....	18-8
The getReqParam Tag Handler Class .....	18-9
Configuring the Tag Library Descriptor.....	18-11
Tag Declaration Element.....	18-12
Custom Tag Body Content .....	18-13
Custom Tag Attributes.....	18-13
Custom Tag That Includes the Body .....	18-14
The heading Tag.....	18-15
The heading Tag Handler Class.....	18-16
The heading Tag Descriptor .....	18-18
Summary .....	18-19
Certification Exam Notes .....	18-20
<b>Developing Advanced Custom Tags .....</b>	<b>19-1</b>
Objectives .....	19-1
Writing a Conditional Custom Tag .....	19-2
Example: The checkStatus Tag .....	19-2
The checkStatus Tag Handler .....	19-3
The checkStatus Tag Life Cycle .....	19-4
Writing an Iterator Custom Tag.....	19-5
Iteration Tag API.....	19-6
Iteration Tag Life Cycle .....	19-7
Example: The iterateOverErrors Tag .....	19-9
The iterateOverErrors Tag Handler .....	19-10
Using the Page Scope to Communicate .....	19-13
Summary .....	19-15
Certification Exam Notes .....	19-16
<b>Integrating Web Applications With Enterprise JavaBeans Components.....</b>	<b>20-1</b>
Objectives .....	20-1
Relevance.....	20-2
Additional Resources .....	20-3
Distributing the Business Logic .....	20-4
Java 2 Platform, Enterprise Edition .....	20-5

---

Enterprise JavaBeans Technology .....	20-6
Integrating the Web Tier With the EJB Tier .....	20-7
Enterprise JavaBeans Interfaces .....	20-8
Java Naming and Directory Service .....	20-9
Creating the Business Delegate.....	20-10
Declaring the Business Delegate Class .....	20-11
Finding the Home interface Using JNDI .....	20-12
Delegating Business Methods to the EJB Tier.....	20-13
Using the Business Delegate in a Servlet Controller.....	20-14
Using a Delegate in a Servlet.....	20-15
Summary .....	20-17
Certification Exam Notes .....	20-18
<b>Quick Reference for HTML.....</b>	<b>A-1</b>
Objectives .....	A-1
Additional Resources .....	A-2
HTML and Markup Languages .....	A-3
Definition.....	A-3
Types of Markup .....	A-3
Simple Example.....	A-4
Basic Structure of HTML.....	A-5
Tag Syntax.....	A-5
Comments .....	A-5
Spaces, Tabs, and Newlines Within Text.....	A-6
Character and Entity References.....	A-6
Links and Media Tags .....	A-7
The HREF Attribute and the A Element.....	A-7
The IMG Element and the SRC Attribute .....	A-8
The APPLET Element .....	A-8
The OBJECT Element .....	A-9
Text Structure and Highlighting.....	A-10
Text Structure Tags .....	A-10
Text Highlighting.....	A-13
HTML Forms .....	A-14
The FORM Tag .....	A-14
HTML Form Components .....	A-15
Input Tags .....	A-16
Text Fields .....	A-17
Submit Buttons .....	A-18
Reset Button .....	A-19
Checkboxes .....	A-20
Radio Buttons .....	A-21
Password .....	A-22
Hidden Fields .....	A-22
The SELECT Tag.....	A-23
The TEXTAREA Tag.....	A-24

---

Table Elements .....	A-25
Advanced HTML .....	A-28
The JavaScript™ Language .....	A-28
CSS .....	A-30
Frames.....	A-33
<b>Quick Reference for HTTP .....</b>	<b>B-1</b>
Objectives .....	B-1
Additional Resources .....	B-2
Introduction to HTTP .....	B-3
Definition.....	B-3
Structure of Requests.....	B-4
HTTP Methods .....	B-5
Request Headers .....	B-6
Structure of Responses .....	B-8
Response Headers.....	B-9
Status Codes.....	B-10
CGI .....	B-12
Set of Environment Variables.....	B-12
Data Formatting .....	B-14
<b>Quick Reference for the Tomcat Server .....</b>	<b>C-1</b>
Objectives .....	C-1
Additional Resources .....	C-2
Definition of the Tomcat Server.....	C-3
Installation Instructions .....	C-3
Starting and Stopping the Tomcat Server Execution .....	C-4
Starting the Tomcat Server .....	C-4
Stopping the Tomcat Server .....	C-5
Configuration .....	C-6
Logging and Log Files.....	C-10
<b>Quick Reference for the Ant Tool .....</b>	<b>D-1</b>
Objectives .....	D-1
Additional Resources .....	D-2
Introduction to Ant .....	D-3
Build File Structure .....	D-4
Projects.....	D-4
Targets .....	D-4
Tasks .....	D-5
Properties .....	D-5
Ant Special Features .....	D-6
Patterns .....	D-6
The <code>fileset</code> Element.....	D-6
Filtering .....	D-7
Basic Built-in Ant Tasks .....	D-8
The <code>copy</code> Task.....	D-8

---

The delete Task .....	D-9
The mkdir Task .....	D-10
The echo Task.....	D-10
The javac Task .....	D-11
The javadoc Task.....	D-11
The jar Task.....	D-12
Complete Ant Build File .....	D-13
Executing Ant .....	D-15
Install Instructions .....	D-16
System Requirements .....	D-16
Environment Setup for Ant .....	D-16
<b>Quick Reference for XML .....</b>	<b>E-1</b>
Objectives .....	E-1
Additional Resources .....	E-2
Introduction to XML.....	E-3
Simple Example.....	E-3
Basic Syntax .....	E-4
Well-Formed XML Documents.....	E-4
Validity and DTDs .....	E-5
DTD-specific Information .....	E-7
Schemas .....	E-9
<b>Quick Reference for UML .....</b>	<b>F-1</b>
Additional Resources .....	F-1
What Is UML? .....	F-2
Modeling in UML .....	F-2
User View .....	F-3
Structural View.....	F-3
Behavioral View .....	F-3
Implementation View.....	F-3
Environment View .....	F-4
General Elements .....	F-4
Packages .....	F-5
Stereotypes .....	F-6
Annotation .....	F-7
Constraints .....	F-7
Tagged Values .....	F-8
Use Case Diagrams .....	F-9
Class Diagrams .....	F-10
Class Nodes.....	F-10
Inheritance.....	F-13
Interface Implementation.....	F-13
Association, Roles, and Multiplicity .....	F-14
Aggregation and Composition .....	F-15
Association Classes .....	F-16
Other Association Elements .....	F-18

---

Object Diagrams .....	F-19
Sequence Diagrams.....	F-20
Collaboration Diagrams .....	F-22
State Diagrams.....	F-23
Transitions.....	F-24
Activity Diagrams.....	F-25
Component Diagrams .....	F-27
Deployment Diagrams .....	F-29



# About This Course

---

## Course Goal

The *Web Component Development With Java™ Technology* course provides students with the skills to analyze, design, develop, test, and deploy a Web application. This course describes how to create dynamic Web content using Java™ technology servlets and JavaServer Pages™ (JSP™) technology, including custom tag library development. The course describes how to construct small to medium scale Web applications and deploy them onto the Tomcat server, which is the reference implementation for the servlet and JSP specifications. The course also summarizes best practices for integrating the Web tier with other tiers, such as a database server and an Enterprise JavaBeans™ components (EJB™) server.

This course is designed for developers who use the Java programming language to create components, such as servlets and custom tags, for Web applications.

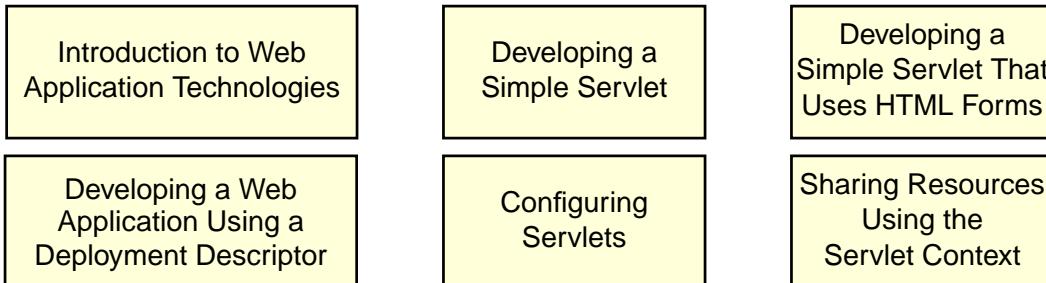
## Learning Objectives

When you have completed this course, you should be able to do the following:

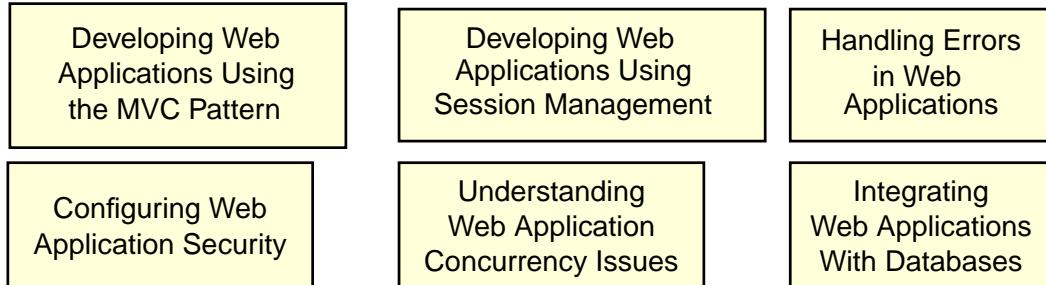
- Develop a Web application using Java servlets
- Develop a robust Web application using the Model-View-Controller (MVC) software pattern, session management, exception handling, declarative security, and proper concurrency control
- Develop a Web application using JavaServer Pages technology
- Develop a custom tag library
- Develop a Web application that integrates with an n-tiered architecture using a relational database management system (RDBMS) or EJB server

# Module-by-Module Overview

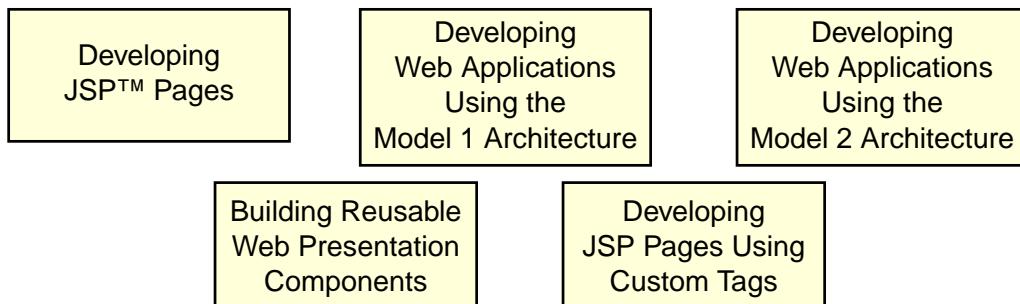
## Design and Development of N-Tier Web Applications



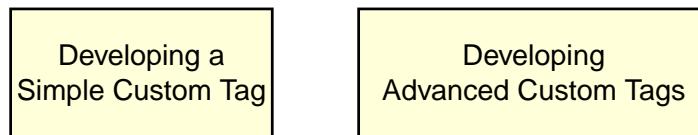
## Java Servlet Application Strategies



## JSP™ Application Strategies



## Developing Custom JSP Tag Libraries



## Java 2 Platform, Enterprise Edition



## Topics Not Covered

This course does not cover the following topics. Many of these topics are covered in other courses offered by Sun Educational Services:

- Java technology programming – Covered in SL-275: *The Java<sup>TM</sup> Programming Language*
- Object-oriented design and analysis – Covered in OO-226: *Object-Oriented Analysis and Design for Java Technology (UML)*
- Java 2 Platform, Enterprise Edition – Covered in SEM-SL-345: *Java<sup>TM</sup> 2 Platform, Enterprise Edition: Technology Overview Seminar*
- Enterprise JavaBeans – Covered in SL-351: *Enterprise JavaBeans<sup>TM</sup> Programming*

Refer to the Sun Educational Services catalog for specific information and registration.

## How Prepared Are You?

To be sure you are prepared to take this course, can you answer yes to the following questions?

- Can you create Java technology applications?
- Can you read and use a Java technology application programming interface (API)?
- Can you analyze and design a software system using a modeling language like Unified Modeling Language (UML)?
- Can you develop applications using a component/container framework?

## How to Learn From This Course

To get the most out of the course, you should:

- Ask questions
- Participate in the discussions and exercises
- Use the online documentation for Java™ 2, Standard Edition (J2SE™), servlet, and JSP APIs
- Read the servlet and JSP specifications

## Introductions

Now that you have been introduced to the course, introduce yourself to the other students and the instructor, addressing the items shown on the overhead.

# How to Use Course Materials

To enable you to succeed in this course, these course materials use a learning module that is composed of the following components:

- **Goals** – You should be able to accomplish the goals after finishing this course and meeting all of its objectives.
- **Objectives** – You should be able to accomplish the objectives after completing a portion of instructional content. Objectives support goals and can support other higher-level objectives.
- **Lecture** – The instructor will present information specific to the objective of the module. This information should help you learn the knowledge and skills necessary to succeed with the activities.
- **Activities** – The activities take on various forms, such as an exercise, self-check, discussion, and demonstration. Activities help facilitate mastery of an objective.
- **Visual aids** – The instructor might use several visual aids to convey a concept, such as a process, in a visual form. Visual aids commonly contain graphics, animation, and video.

## Conventions

The following conventions are used in this course to represent various training elements and alternative learning resources.

### Icons



**Additional resources** – Indicates other references that provide additional information on the topics described in the module.



**Demonstration** - Indicates a demonstration of the current topic is recommended at this time.



**Discussion** – Indicates a small-group or class discussion on the current topic is recommended at this time.



**Note** – Indicates additional information that can help students but is not crucial to their understanding of the concept being described. Students should be able to understand the concept or complete the task without this information. Examples of notational information include keyword shortcuts and minor system adjustments.

---



**Caution** – Indicates that there is a risk of personal injury from a nonelectrical hazard, or risk of irreversible damage to data, software, or the operating system. A caution indicates that the possibility of a hazard (as opposed to certainty) might happen, depending on the action of the user.

---

## Typographical Conventions

Courier is used for the names of commands, files, directories, programming code, and on-screen computer output; for example:

Use `ls -al` to list all files.  
system% You have mail.

Courier is also used to indicate programming constructs, such as class names, methods, and keywords; for example:

The `getServletInfo` method is used to get author information.  
The `java.awt.Dialog` class contains `Dialog` constructor.

**Courier bold** is used for characters and numbers that you type; for example:

To list the files in this directory, type:  
`# ls`

**Courier bold** is also used for each line of programming code that is referenced in a textual description; for example:

```
1 import java.io.*;  
2 import javax.servlet.*;  
3 import javax.servlet.http.*;
```

Notice the `javax.servlet` interface is imported to allow access to its life cycle methods (Line 2).

*Courier italic* is used for variables and command-line placeholders that are replaced with a real name or value; for example:

To delete a file, use the `rm filename` command.

**Courier italic bold** is used to represent variables whose values are to be entered by the student as part of an activity; for example:

Type `chmod a+rwx filename` to grant read, write, and execute rights for `filename` to world, group, and users.

*Palatino italic* is used for book titles, new words or terms, or words that you want to emphasize; for example:

Read Chapter 6 in the *User's Guide*.  
These are called *class* options.

## Additional Conventions

Java programming language examples use the following additional conventions:

- Method names are not followed with parentheses unless a formal or actual parameter list is shown; for example:  
“The `doIt` method...” refers to any method called `doIt`.  
“The `doIt()` method...” refers to a method called `doIt` that takes no arguments.
- Line breaks occur only where there are separations (commas), conjunctions (operators), or white space in the code. Broken code is indented four spaces under the starting code.
- If a command used in the Solaris™ Operating Environment is different from a command used in the Microsoft Windows platform, both commands are shown; for example:

If working in the Solaris Operating Environment

```
$CD SERVER_ROOT/BIN
```

If working in Microsoft Windows

```
C:\>CD SERVER_ROOT\BIN
```

# Introduction to Web Application Technologies

---

## Objectives

This module presents Web application basics: the history of browsers and Web servers and what they do. It also presents the main Web application technologies and provides you with standards by which to evaluate the advantages and disadvantages of each.

Upon completion of this module, you should be able to:

- Describe Internet services
- Describe the World Wide Web
- Distinguish between Web applications and Web sites
- Describe Java servlet technology and list three benefits of this technology compared with traditional Common Gateway Interface (CGI) scripting
- Describe JavaServer Pages technology and list three benefits of JSP pages technology over rival template page technologies
- Describe the Java<sup>TM</sup> 2 Platform, Enterprise Edition (J2EE<sup>TM</sup>)

## Relevance



Discussion – The following questions are relevant to understanding what technologies are available for developing Web applications and the limitations of those technologies:

- What Web applications have you developed?
- Were there goals you could not achieve because of the technology you used?

## Additional Resources

The following references provide additional details on the topics that are presented briefly in this module and described in more detail later in the course:

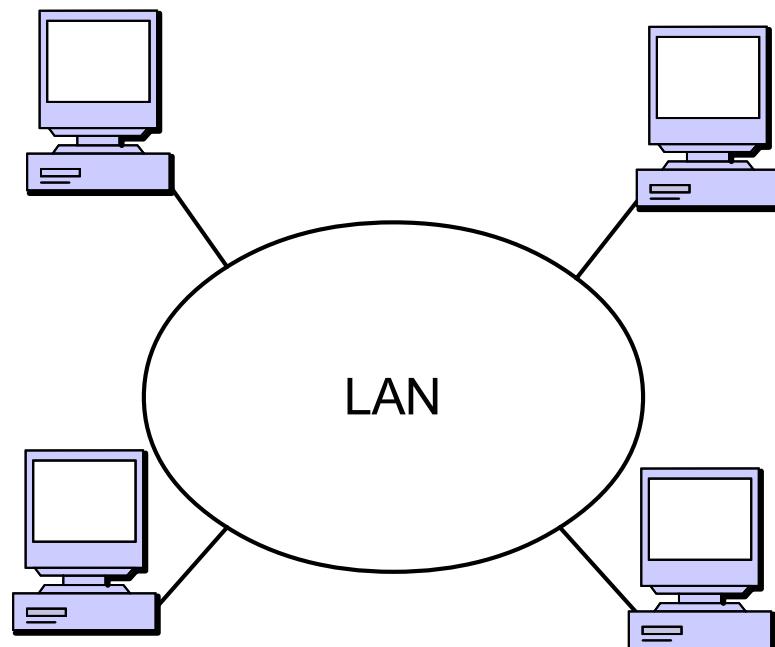
- *Java Servlets Specification*. [Online]. Available:  
<http://java.sun.com/products/servlet/>
- *JavaServer Pages Specification*. [Online]. Available:  
<http://java.sun.com/products/jsp/>
- *Java™ 2 Platform, Enterprise Edition Blueprints*. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Tomcat download page. [Online]. Available:  
<http://jakarta.apache.org/tomcat>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- Shishir Gundavarama, *CGI Programming on the World Wide Web*, Sebastopol: O'Reilly & Associates, Inc., 1996.

# Internet Services

It is important to have a fundamental understanding of the components of the *Internet*: the clients, the servers, and the networks used to connect them.

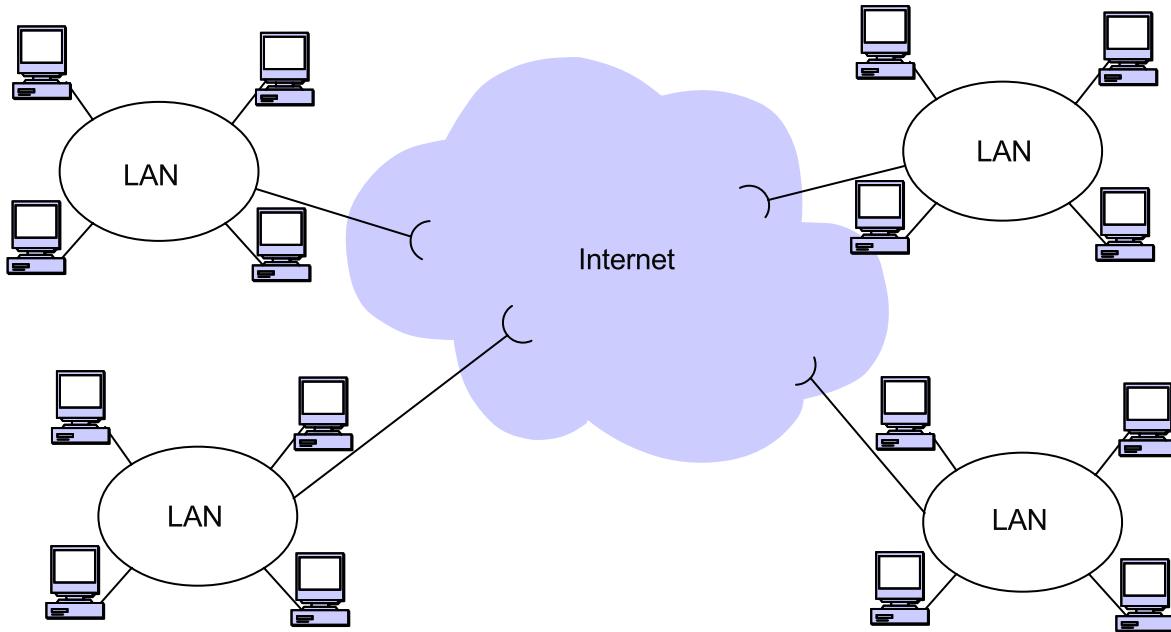
## The Internet Is a Network of Networks

In the early days of computer networking, local area networks (LANs) were created to share resources among members of a particular institution. These networks were constrained to short distances. This is illustrated in Figure 1-1.



**Figure 1-1** Pre-Internet Networking Environment

Soon different institutions started connecting their networks to other networks as well. This enabled increased collaboration among academics and scientists across the world. The resulting “network of networks” is shown in Figure 1-2.

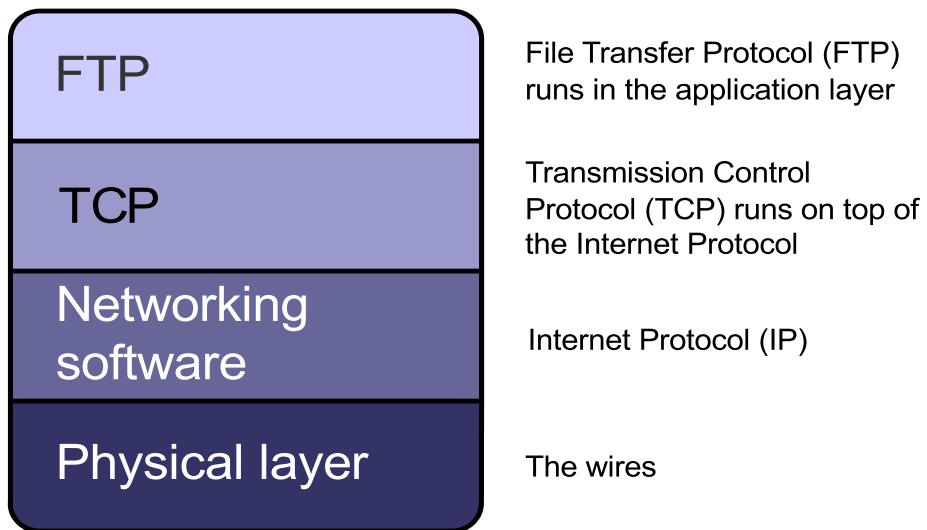


**Figure 1-2** Simplified Illustration of the Internet

Connectivity is just one element of what makes the Internet work. Another element is the language of communication between computers. This is called a protocol.

## Networking Protocol Stack

Servers and clients that interact on a network are connected by several layers of hardware and software working together. These layers are shown in Figure 1-3.



**Figure 1-3** Network Layers of Physical and Software Elements



**Note** – Figure 1-3 is based on the International Organization for Standardization (ISO) seven-layer network stack, but leaves out several features that are unimportant to this discussion. This is often referred to as the “4 layer” model.

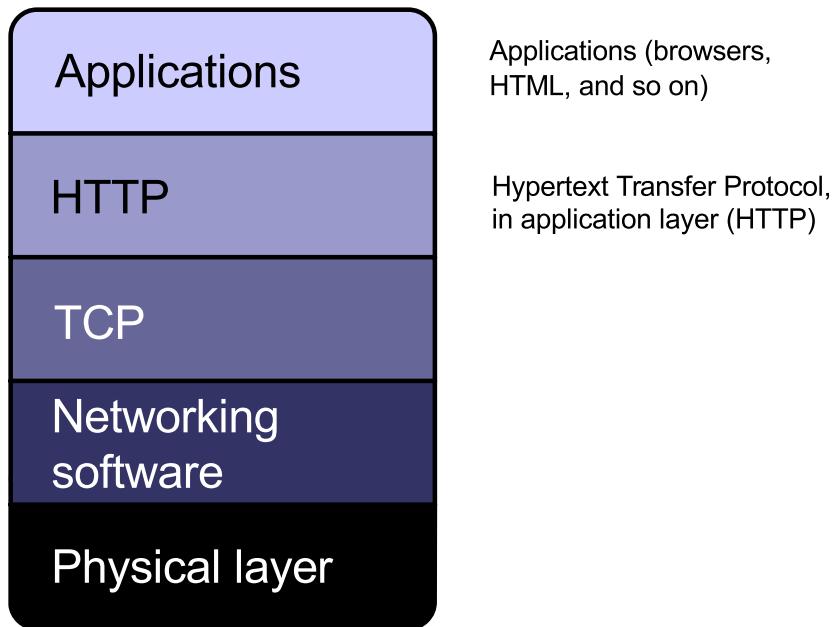
To retrieve a file from a remote server, a user would use a File Transfer Protocol (FTP) program. This required several commands entered into the FTP program to log in to the remote server, navigate to the necessary directory, and then retrieve the file. This is done using a command line interface.

FTP works by sending messages between the client and the remote server. These messages may be brief text commands or files of any size and type. FTP is built on top of the Transmission Control Protocol (TCP), which guarantees that the message is transmitted across the network without any errors.

TCP is built on top of the Internet Protocol (IP), which provides a set of low-level services. TCP can send a message of any size, but IP uses a packet mechanism that sends data in a fixed size. IP does not guarantee delivery of the packet.

IP is built on top of a variety of physical devices. These devices include the cables connecting computers, routers, switches, and so on.

FTP is just one application of an Internet service. There are scores of standard services, including email, telnet, time, system status, and so on. Another example protocol stack is shown in Figure 1-4.

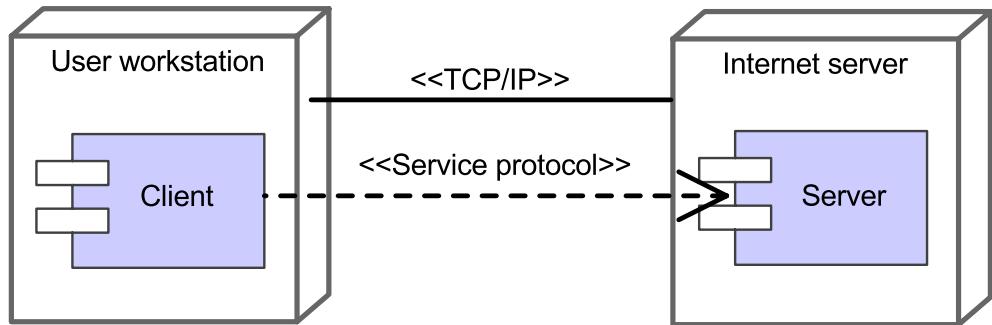


**Figure 1-4** Another Networking Layer Example

You can also create proprietary protocols built on top of TCP.

## Client-Server Architecture

In traditional Internet services, there is a client host and a server host. The client makes requests of the server and the server responds to those requests. TCP/IP is the most common transmission protocol stack. The fundamental client-server architecture is shown in Figure 1-5.



**Figure 1-5**      Fundamental Client-Server Architecture

## Hypertext Transfer Protocol

The *Hypertext Transfer Protocol (HTTP)* is similar to FTP because it is a protocol to transfer files from the server to the client. HTTP was created in conjunction with the related *Hypertext Markup Language (HTML)* standard. There is one fundamental difference between FTP and HTTP: HTTP supports only one request per connection. This means that the client connects to the server to retrieve one file and then disconnects. This mechanism allows more users to connect to a given server over a period of time.

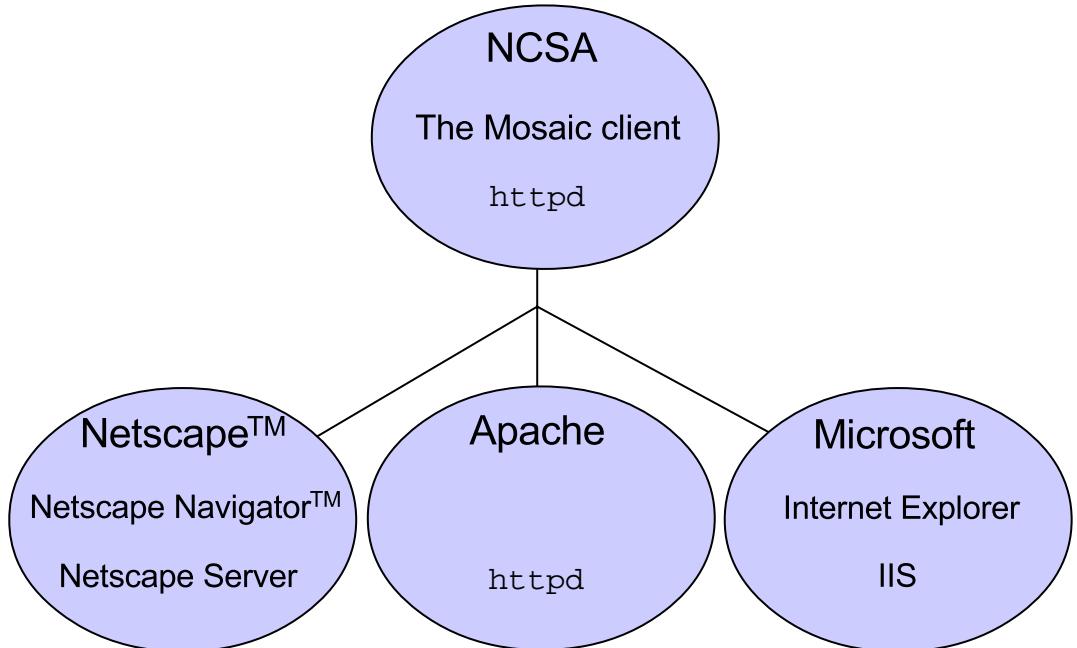
HTML is a document display language that allows users to link from one document to another. For example, a paper on charmed quarks stored at European Organization for Nuclear Research (CERN) in Switzerland could include a “See also” link to a paper on strange quarks stored on a computer in Fargo, North Dakota.

HTML also permits images and other media objects to be embedded in an HTML document. The media objects are stored in files on a server. HTTP also retrieves these files. HTTP can therefore be used to transmit any file that conforms to the Multipurpose Internet Mail Extensions (MIME) specification.

## Web Browsers and Web Servers

To view an HTML document with rich media content, a graphical user interface (GUI) was built on top of the client-side HTTP. This GUI is called a Web browser. The server-side HTTP component is called a Web server. Several companies have developed Web browsers and Web servers; some have developed both. The first Web server was a process called `httpd`; the first widely used browser was Mosaic, created by National Center for Supercomputing Applications (NCSA).

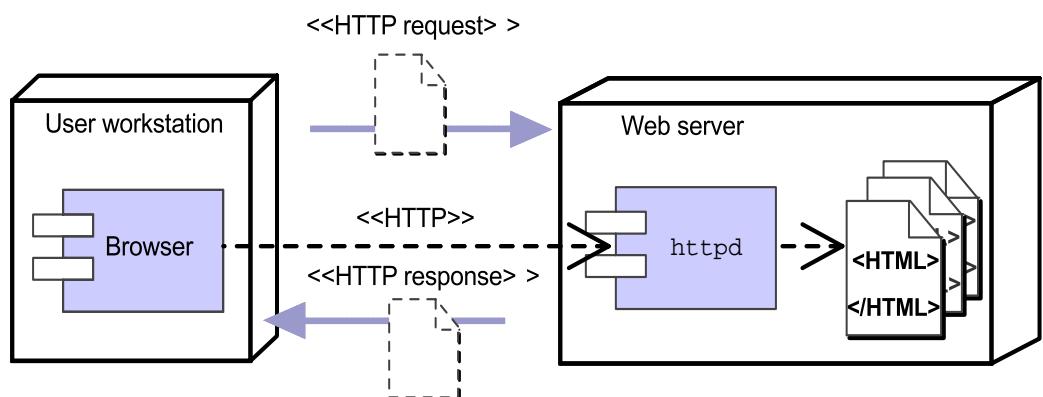
Mosaic became extremely popular, and various companies developed their own Web browsers. The early development of Web browsers and Web servers is illustrated in Figure 1-6.



**Figure 1-6** Early Web Browser and Web Server Vendors

## HTTP Client-Server Architecture

For every exchange over the Web using HTTP, there is a request and a response. This is illustrated in Figure 1-7.



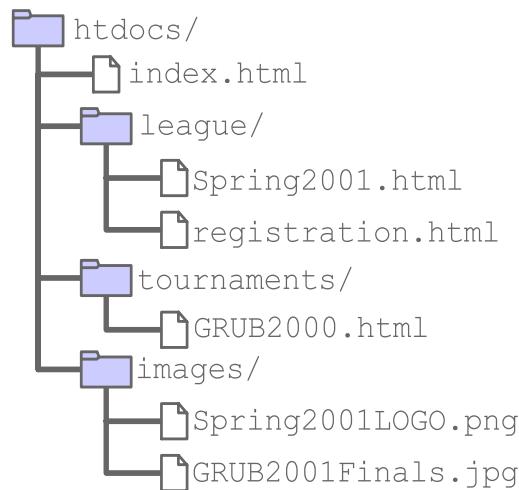
**Figure 1-7** HTTP Client-Server Architecture

The Web browser sends a single request to the server. The Web server determines which file is being requested and sends the data in that file back as the response. The browser interprets the response and represents the content on the screen.

The request information consists of the file or other resource that the user wants and information about the browser. The response information contains the requested file and other information. The request is typically in plain text; the response can be plain text or part plain text, part binary data. (Graphics, for example, must be sent in binary form.)

## The Structure of a Web Site

A *Web site* is a collection of HTML pages and other media files, which contain all the content that is visible to the user on a given Web server. These files are stored on the server and might include a complex directory hierarchy. The Web site is composed of that directory hierarchy. An example of a Web site is shown in Figure 1-8.



**Figure 1-8** An Example Web Site Directory Structure



**Note** – The `index.html` file is a special file used when the user requests a Uniform Resource Locator (URL) that ends in a slash character (/). The Web server presents the user with a directory listing for that URL unless an `index.html` file exists in that directory. If that is the case, then the Web server sends the `index.html` file as the response to the original URL.

## Uniform Resource Locator

A URL is a canonical name that locates a specific resource on the Internet. It consists of:

*protocol://host:port/path/file*

For example:

`http://www.soccer.org:80/league/Spring2001.html`

The *path* element includes the complete directory structure path to find the file. The port number is used to identify the TCP port that is used by the protocol on the server. If the port number is the standard port for the given protocol, then that number can be ignored in the URL. For example, port 80 is the default HTTP port:

`http://www.soccer.org/league/Spring2001.html`

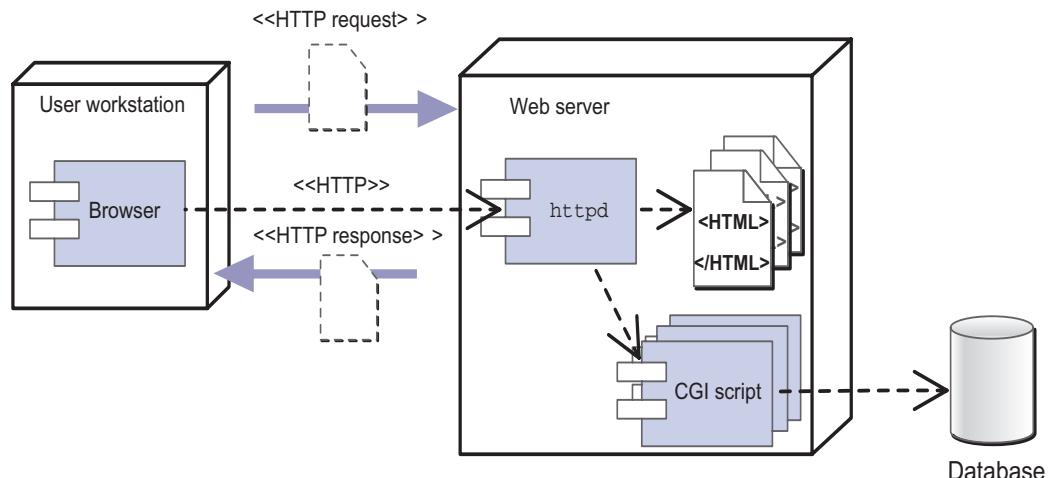
# Web Applications

Very early in the development of HTML, the designers created a mechanism to permit a user to invoke a program on the Web server. This was called the *Common Gateway Interface* (CGI). When a Web site includes CGI processing, this is called a *Web application*.

## CGI Programs on the Web Server

Usually, the browser needs to send data to the CGI program on the server. The CGI specification defines how the data is packaged and sent in the HTTP request to the server. This data is usually typed into the Web browser in an HTML form.

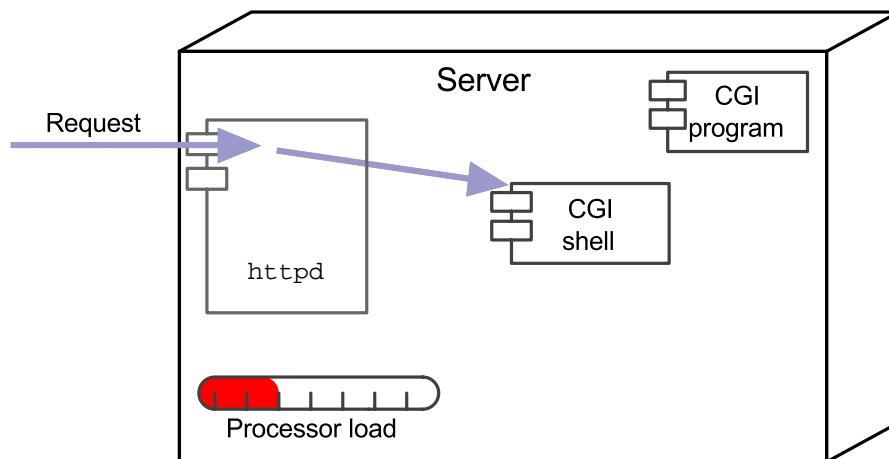
The URL determines which CGI program to execute. This might be a script or an executable file. The CGI program parses the CGI data in the request, processes the data, and generates a response (usually an HTML page). The CGI response is sent back to the Web server, which wraps the response in an HTTP response. The HTTP response is sent back to the Web browser. An example Web application architecture that uses CGI programs is illustrated in Figure 1-9.



**Figure 1-9**      Web Server Architecture With CGI Programs

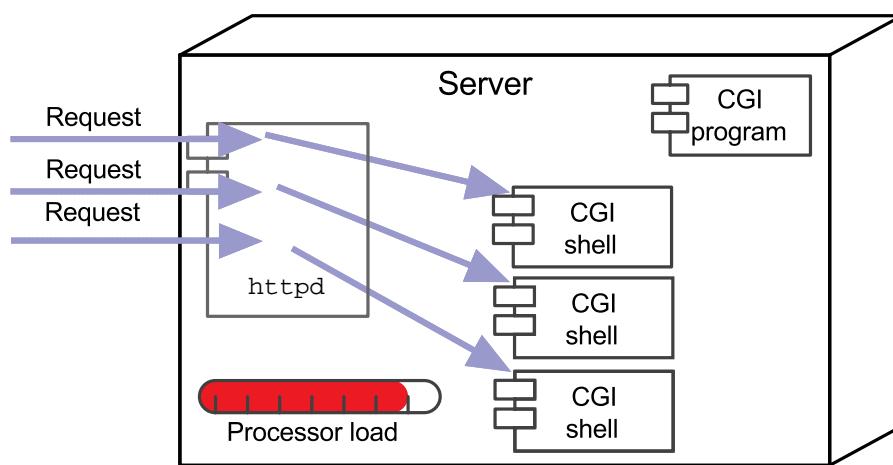
## Execution of CGI Programs

At runtime, a CGI program is launched by the Web server as a separate operating system (OS) shell. The shell includes an OS environment and process to execute the code of the CGI program, which resides within the server's file system. The runtime performance of one CGI request is shown in Figure 1-10.



**Figure 1-10** Running a Single Instance of a CGI Program

However, each new CGI request launches a new operating system shell on the server. The runtime performance of multiple CGI requests is shown in Figure 1-11.



**Figure 1-11** Running Multiple Instances of a CGI Program

## Advantages and Disadvantages of CGI Programs

CGI programs have four *advantages*:

- Programs can be written in a variety of languages, although they are primarily written in Perl.
- A buggy CGI program will not crash the Web server.
- Programs are easy for a Web designer to reference. When the script is written, the designer can reference it in one line in a Web page.
- Because CGI programs execute in their own OS shell, these programs do not have concurrency conflicts with other HTTP requests executing the same CGI program.
- All service providers support CGI programs.

CGI program also have distinct *disadvantages*:

- The response time of CGI programs is very high, because CGI programs execute in their own OS shell. The creation of an OS shell is a heavyweight activity for the OS.
- CGI is not scalable. If the number of people accessing the Web application increases from 50 to 5000, for example, CGI cannot be adapted to handle the load. There is a limit on the number of separate operating system processes a computer can run.
- The languages for CGI are not always secure, or object oriented.
- The CGI script has to generate an HTML response, so the CGI code is mingled with HTML. This is not good separation of presentation and business logic.
- Scripting languages are often platform dependent.

Because of these disadvantages, developers need other CGI solutions. Servlets is the Java technology solution used to process CGI data.

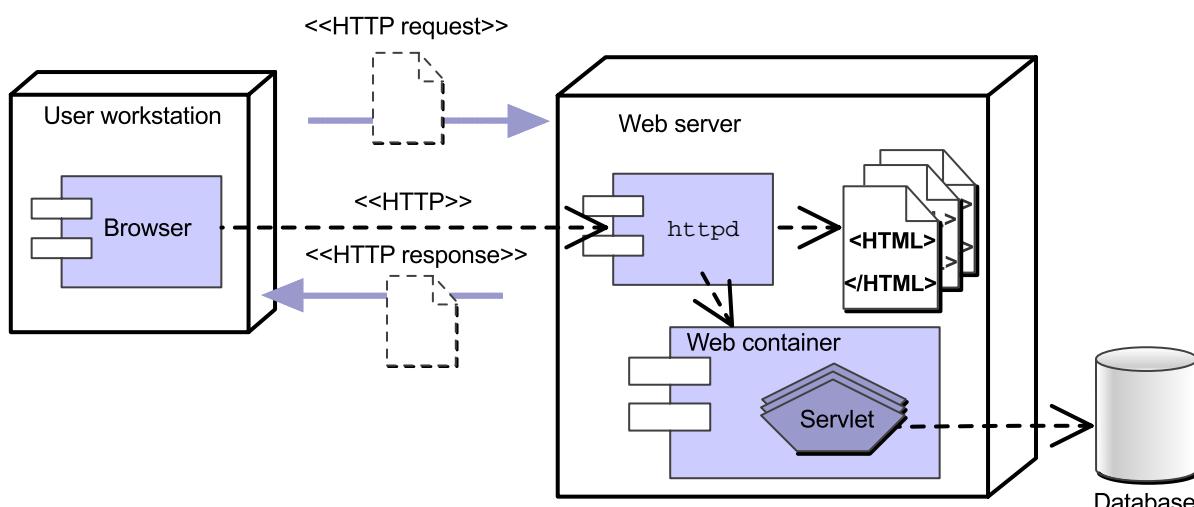
## Java Servlets

Sun Microsystems developed servlets as an advance over traditional CGI technology. A Java servlet is a Java technology program that, like a CGI program, runs on the server. The types of tasks that you can run with servlets are similar to those you can run with CGI; however, the underlying executing architecture is different.

As with CGI scripts, you can write servlets that can understand HTTP requests, generate the response dynamically (possibly querying databases to fulfill the request), and then send a response containing an HTML page or document to the browser.

## Servlets on the Web Server

Unlike CGI programs, servlets run within a component container architecture. This container is called the Web container (the new term for the servlet engine). The Web container is a Java™ virtual machine (JVM™) that supplies an implementation of the servlet application programming interface (API). Servlet instances are components that are managed by the Web container to respond to HTTP requests. This architecture is shown in Figure 1-12.



**Figure 1-12** Web Server Architecture With Java Servlets



**Note** – In some architectures, the Web container acts as a standalone HTTP service; in other architectures, the HTTP service forwards requests to be processed by the Web container.

## Execution of Java Servlets

The basic processing steps for Java servlets are quite similar to the ones for CGI. However, the servlet runs as a thread in the Web container instead of in a separate OS process. The Web container itself is an OS process, but it runs as a service and is available continuously as opposed to a CGI script in which a new OS process (and shell) is created for each request. A servlet executes as a thread within the Web container's process. This is illustrated in Figure 1-13.

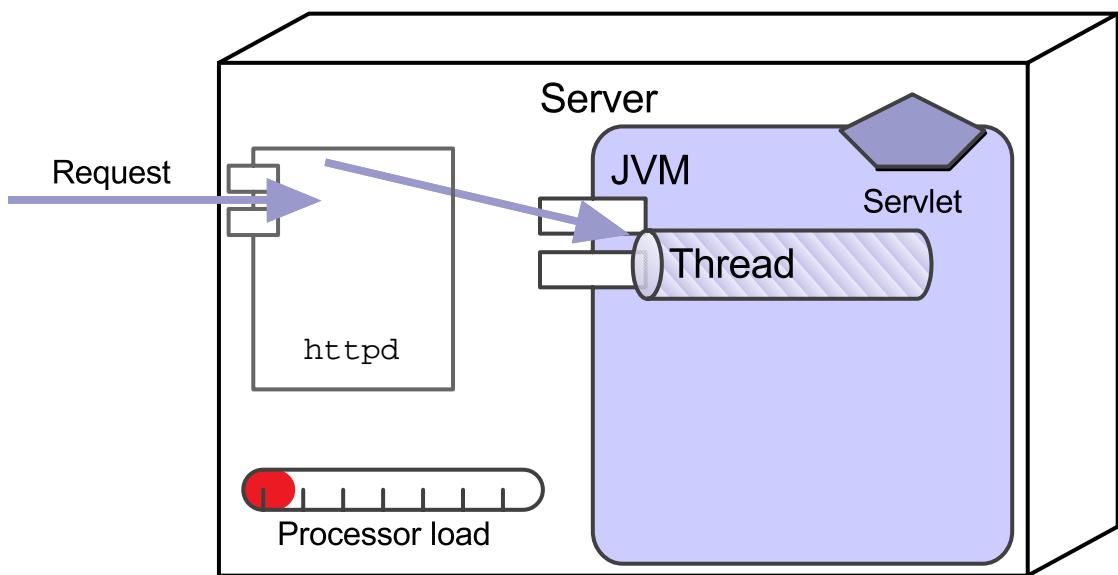


Figure 1-13 Running a Single Instance of a Servlet

When the number of requests for a servlet rises, no additional instances of the servlet or operating system processes are created. Each request is processed concurrently using one Java thread per request. The effect of additional clients requesting the same servlet is shown in Figure 1-14.

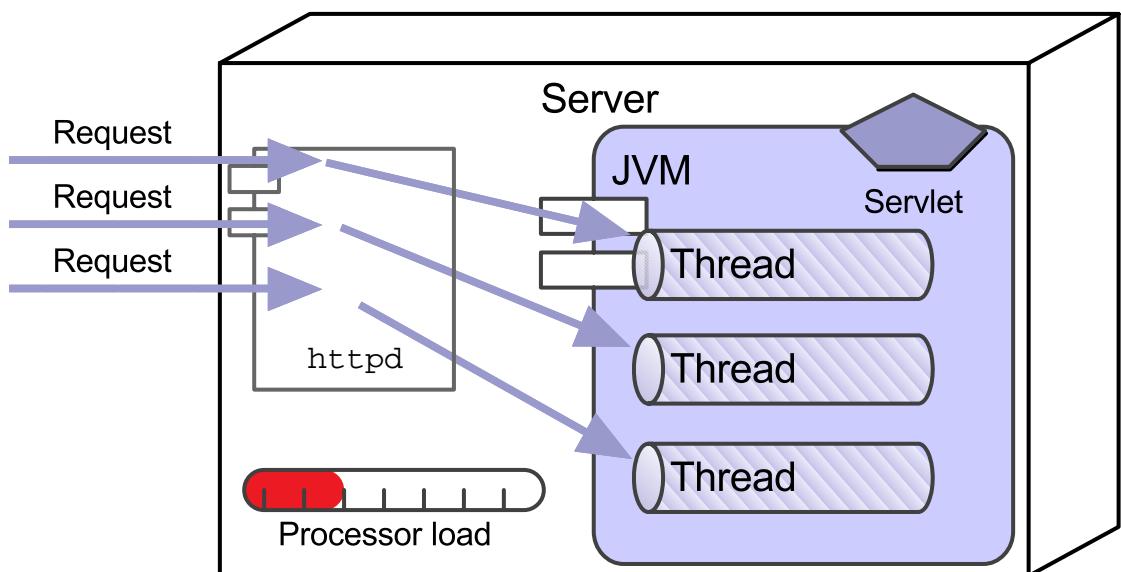


Figure 1-14 Running Multiple Instances of a Servlet

## Advantages and Disadvantages of Java Servlets

Servlets have the following *advantages*:

- Each request is run in a separate thread, so servlet request processing is significantly faster than traditional CGI processing.
- Servlets are scalable. Many more requests can be executed because the Web container uses a thread rather than an operating system process, which is a limited system resource.
- Servlets are robust and object oriented. You have all the capabilities of the Java programming language when you write the servlet, instead of the capabilities of Perl or whatever language you use to write the CGI script.
- Servlets can only be written in the Java programming language, which makes them easy to write if you know the Java programming language. However, using servlets to generate pages with dynamic content requires application development expertise.
- Servlets are platform independent, because they are written in the Java programming language.
- Servlets have access to logging capabilities. Most CGI programs do not.
- The Web container provides additional services to the servlets, such as error handling and security.

Servlets have the following *disadvantages*:

- Servlets often contain both *business logic* and *presentation logic*. Presentation logic is anything that controls how the application presents information to the user. Generating the HTML response within the servlet code is presentation logic. Business logic is anything that manipulates data in order to accomplish something, such as storing data.
- Servlets must handle concurrency issues.

Mixing presentation and business logic means that whenever a Web page changes (which can be monthly or weekly for many applications) the servlets must be rewritten, recompiled, and redeployed.

This disadvantage led to the development of *template pages*, including JavaServer Pages technology.

# Template Pages

JSP pages are just one way of implementing the concept of HTML pages with embedded code, or template pages. There are three mainstream technologies available for creating HTML with embedded code: PHP from Apache, Active Server Pages (ASP) from Microsoft, and JSP from Sun Microsystems. PHP and ASP, however, work only with proprietary Web servers.

With JSP pages, Java technology code fragments are embedded in an HTML-like file. This code is executed at runtime to create dynamic content. An example JSP template page is shown in Code 1-1.

**Code 1-1** An Example JSP Template Page

```
1 <HTML>
2
3 <HEAD>
4 <TITLE>Example JSP Page</TITLE>
5 </HEAD>
6
7 <BODY BGCOLOR='white'>
8
9 <B>Table of numbers squared:</B>
10
11 <TABLE BORDER='1' CELLSPACING='0' CELLPADDING='5'>
12 <TR><TH>number</TH><TH>squared</TH></TR>
13 <% for ( int i=0; i<10; i++ ) { %>
14 <TR><TD><%= i %></TD><TD><%= (i * i) %></TD></TR>
15 <% } %>
16 </TABLE>
17
18 </BODY>
19
20 </HTML>
```

This example page generates an HTML table with the numbers 0 through 9 in the first column and the corresponding squares in the second column. A screen shot from the generated page appears in Figure 1-15.

**Table of numbers squared:**

number	squared
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81

**Figure 1-15** Output of Example Template Page

## Other Template Page Technologies

The following three code examples demonstrate the same “for loop” in three template page technologies: PHP, ASP, and JSP, respectively. This is shown to demonstrate how similar these template page technologies are.

**Code 1-2** PHP (PHP Hypertext Preprocessor)

```
<? for ( $i=0; $i<10; $i++ ) { ?>
<TR><TD><? echo $i ?></TD><TD><? echo ($i * $i) ?></TD></TR>
<? } ?>
```

**Code 1-3** ASP (Active Server Pages)

```
<% FOR I = 0 TO 10 %>
<TR><TD><%= I %></TD><TD><%= (I * I) %></TD></TR>
<% NEXT %>
```

**Code 1-4** JSP (JavaServer Pages)

```
<% for ( int i=0; i<10; i++ ) { %>
<TR><TD><%= i %></TD><TD><%= (i * i) %></TD></TR>
<% } %>
```

## JavaServer Pages Technology

All template page technologies have the same fundamental structure: an HTML page that a Web designer can easily create, with special tags, which indicate to the Web server that code needs to be executed at request-time. This course focuses only on JSP pages.

JSP pages are the opposite of servlets. Instead of Java technology code that contains HTML, template pages are HTML that contains Java technology code. JSP pages are converted by the Web container into a servlet instance. That servlet then processes each request to that JSP page. This feature of JSP pages is an advantage over other template page technologies because JSP pages are compiled into Java technology byte code whereas ASP (or PHP) pages are interpreted on each HTTP request.

The JSP page runs as a servlet; everything that you can do in a servlet you can do in a JSP page. The main difference is that a JSP page should focus on the presentation logic of the Web application

## Advantages and Disadvantages of JavaServer Pages

Because JSP pages are translated into Java servlets, JSP technology has all of the *advantages* of servlets:

- Web applications using JSP pages have high performance and scalability because threads are used rather than operating system's shells or processes.
- JSP technology is built on Java technology, so it is platform independent.
- JSP scripting elements can be written in the Java language so that JSP pages can take advantage of the object-oriented language and all of its APIs.

JSP technology has the following *disadvantages*:

- Often JSP pages contain both presentation logic and business logic.
- JSP pages must also consider concurrency issues.
- JSP pages are difficult to debug.

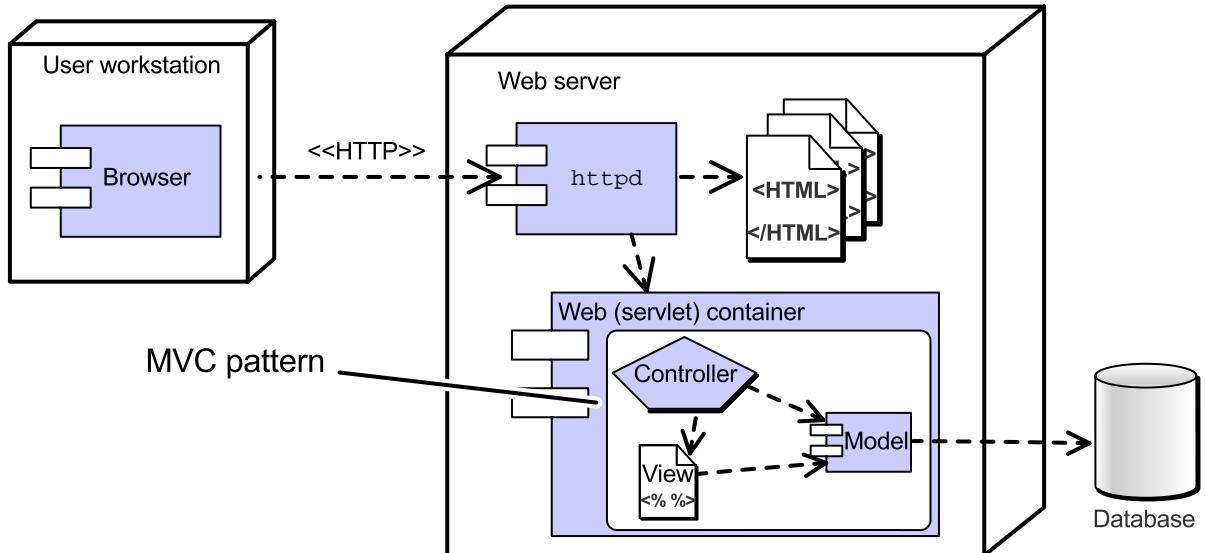
A properly designed Web application should use servlets and JSP pages together to achieve *separation of concerns*.

## The Model 2 Architecture

Using servlets and JSP pages together facilitates proper separation of concerns using a variation on the *Model-View-Controller* (MVC) design pattern. A Web application designed using the *Model 2 architecture* has the following features:

- A servlet acts as the Controller, which verifies form data, updates the Model with the form data, and selects the next View as the response.
- A JSP page acts as the View, which renders the HTML response, retrieving data from the Model necessary to generate the response, and provides HTML forms to permit user interaction.
- Java technology classes act as the Model, which implements the business logic of the Web application.

The Model 1 architecture is shown in Figure 1-16.



**Figure 1-16** Web Server Architecture Using the Model 2 Design

This combination of using servlet Controllers and JSP page Views provides the most advantages. The Model 2 architecture makes Web applications:

- Fast
- Powerful
- Easy to create, for an accomplished Java technology developer
- Cross-platform
- Scalable
- Maintainable (supports separation of presentation and business logic)

## The J2EE Platform

Servlets and JSP pages are an integral part of the Java 2 Platform, Enterprise Edition (J2EE). Also included in the J2EE platform is an Enterprise JavaBeans (EJB) container. An EJB container provides a modular framework for business logic components (the Model elements), called enterprise beans. An EJB container also provides services for transaction management, persistence, security, and life cycle management of enterprise beans.



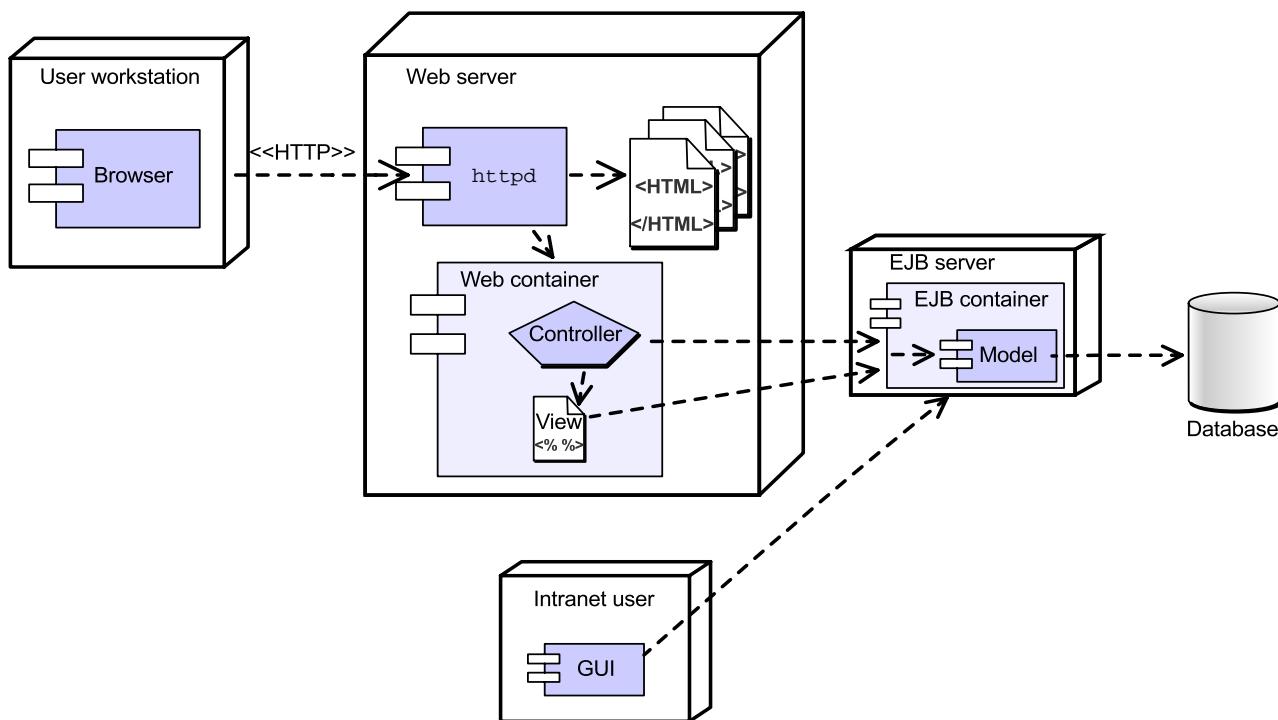
---

**Note** – A detailed discussion of the J2EE platform and EJB components is beyond the scope of this course. If you are interested in EJB components, you should consider taking SL-351, *Enterprise JavaBeans™ Programming*.

---

## An Example of J2EE Architecture

Figure 1-17 illustrates the J2EE platform, which facilitates an architecture in which the business components are placed in a separate tier. This lets GUI applications, as well as Web applications, access the same components.



**Figure 1-17** An Example of J2EE Architecture

The J2EE architecture enhances features such as scalability, extensibility, and maintainability.

Modular design allows for easier modification of the business logic. In addition, enterprise components can leverage their EJB container for services, such as component and resource management, security, persistence, and transactions. It also is an advantage for security, because it partitions business services from the Web tier.

The modular design of the J2EE platform also permits the clean separation of job roles.

## Job Roles

In large Web application projects, development teams are often organized based on the skill sets needed in the project. These are a few of the more common job roles in Web application teams:

- *Content Creator* – Creates View elements

In a Web application, the View elements are either static HTML pages or dynamically generated pages (usually using JSP technology).

- *Web Component Developer* – Creates Controller elements

The Controller elements consist mostly of servlets but can also be classes that support JSP page development (for example, custom tag libraries).

- *Business Component Developer* – Creates Model elements

The Model elements might exist on the Web tier as standard Java technology classes or JavaBeans components or on the EJB tier as EJB components.

- *Data Access Developer* – Creates database access elements

The data access elements perform persistence management for the Model elements that exist in a data source (usually a relational database).

## Web Application Migration

Not every application needs a J2EE architecture, although most with any significant transactional complexity can benefit from it. Most applications start small and build incrementally. It is always beneficial to design an application in such a way that it can be migrated to a scalable, multitier design as a project's scope changes.

Four general types of Web applications can be implemented with the J2EE platform: static HTML, HTML with basic JSP pages and servlets, JSP pages with JavaBeans components, and highly-structured applications that use modular components and enterprise beans. The first three types of applications are considered to be Web-centric, whereas the last type is called J2EE-centric.

A matrix of the relationship between an architecture's complexity and robustness, based on the technologies used, is shown in Figure 1-18. As the richness and robustness of a Web application increases, so does the complexity. The complexity of the application can be managed by proper design that separates the programming concerns. The Web container and J2EE platform provide components that can be used to help manage complex application designs.

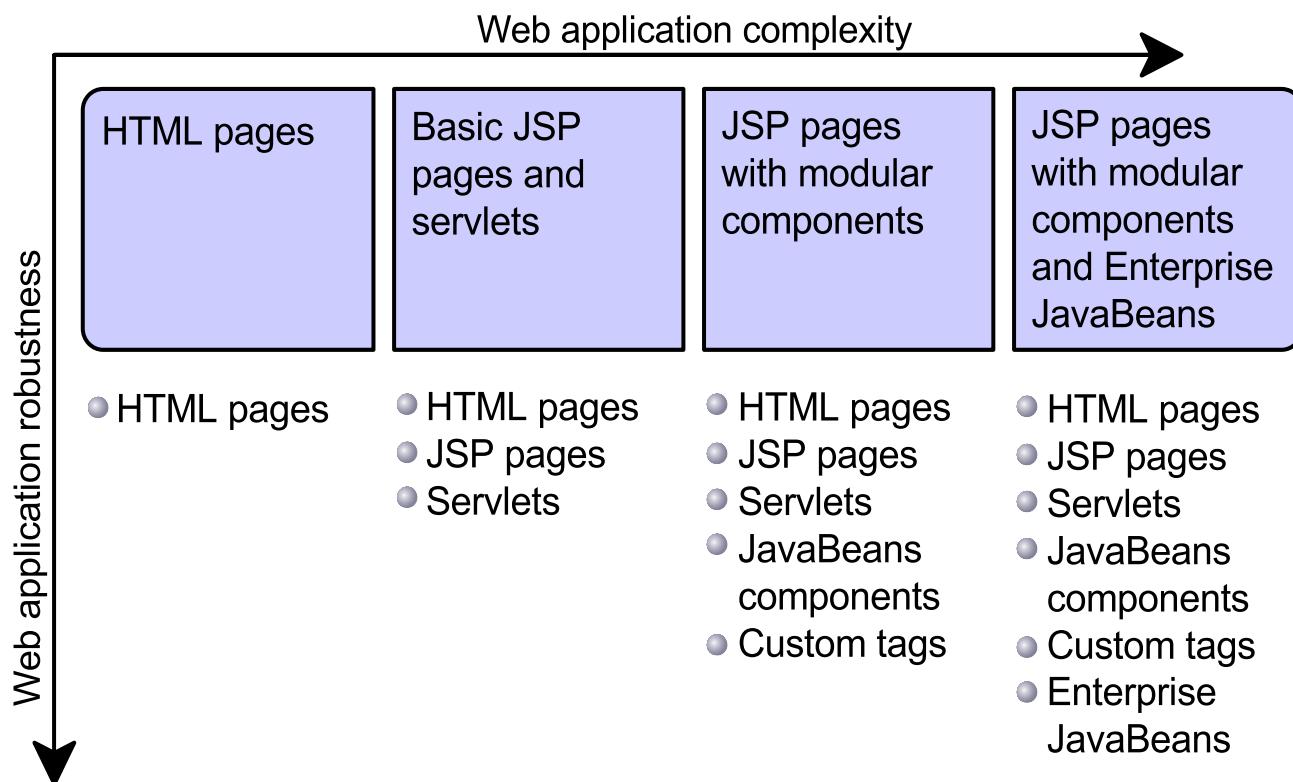


Figure 1-18 Application Design Matrix

## Summary

This module presented Web application basics. The key ideas are:

- Internet services are typically built on top of TCP/IP.
- HTTP is the Internet service for delivering HTML (and other) documents.
- HTTP servers can receive data from HTML forms through CGI.
- Servlets are the Java technology for processing HTTP requests.
- JSP pages are a template-based Java technology for handling presentation logic.
- The Java 2 Platform, Enterprise Edition includes servlets and JSP pages in a broad, enterprise development environment.

## Module 2

---

# Developing a Simple Servlet

---

## Objectives

Upon completion of this module, you should be able to:

- Develop a simple generic servlet
- Describe the Hypertext Transfer Protocol (HTTP)
- Develop a simple HTTP servlet
- Deploy a simple HTTP servlet
- Develop servlets that access request headers
- Develop servlets that manipulate response headers

## Relevance



**Discussion** – The following questions are relevant to understanding what Java servlets are all about:

- What must you do to generate a dynamic HTTP response?
  
- What additional information is provided in the HTTP request and response streams? How can you access this information?

# Generic Internet Services

The Java servlet API supports generic Internet services; that is, it supports services that are not dependent on the HTTP protocol.

## The NetServer Architecture

The World Wide Web runs on the HTTP protocol. The HTTP protocol is the basis for all Web applications. However, it is not the only possible protocol for executing client-server Internet services. In fact, you could create your own protocol. The Java Servlet specification provides an API that allows you to create servlets that use your own protocols.

For example, imagine an Internet service called NetServer with a very simple communication protocol in which the request is of the form:

REQ  
*servlet-class-name*

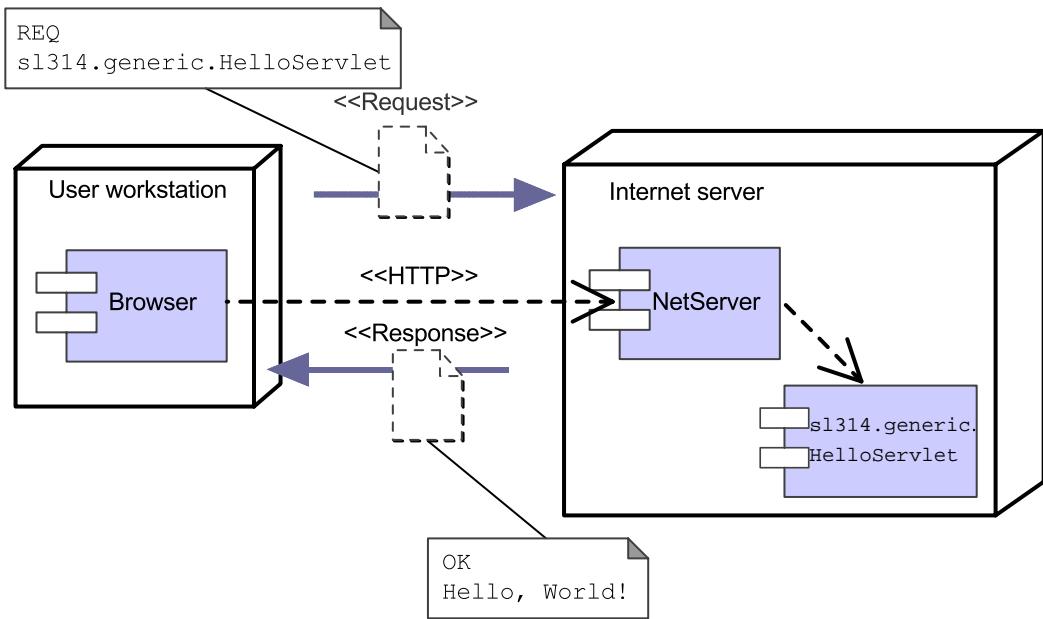
and the response is of the form:

OK  
*response-text*

or a failure response of the form:

FAIL  
*error-message*

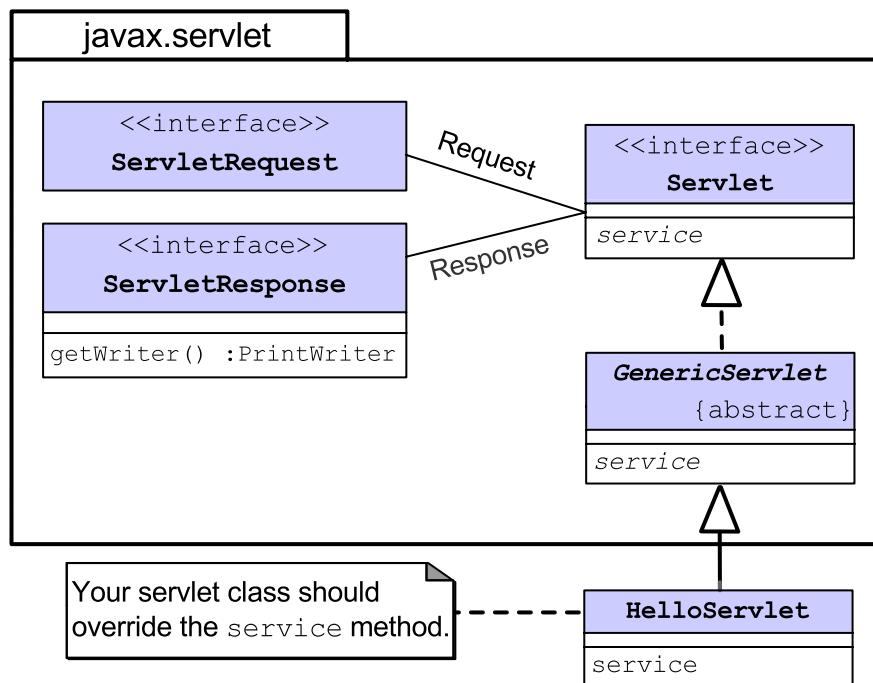
NetServer is a container that activates the servlet class specified in the request and returns the text of the response from the servlet. The architecture for NetServer is shown in Figure 2-1.



**Figure 2-1** The NetServer Architecture

## The Generic Servlets API

As a developer for the NetServer container, what do you need to know? You need to know how to create the servlet component (called HelloServlet, in this example) and how to access the response stream in which you will generate the response output. Additionally, you need to recognize that you have access to the request data, but there is no data of interest in this simplified example. The relevant class structure of the generic servlet API is shown in Figure 2-2.



**Figure 2-2** The Generic Servlet API

Notice that the HelloServlet class extends the GenericServlet class and that this class implements the Servlet interface. The servlet API supplies the GenericServlet class and provides a default implementation of the Servlet interface. The HelloServlet class must override the service method to perform the service for this servlet. In this case, the HelloServlet class simply prints “Hello World” to the response stream.

The service method accepts two arguments: a ServletRequest object and a ServletResponse object. The ServletResponse interface supplies a method to retrieve the response stream; this method is called `getWriter` and returns a `PrintWriter` object.

## The Generic HelloServlet Class

Now that you understand the design of the HelloServlet class. The code is shown in Code 2-1.

**Code 2-1**      The Generic HelloServlet Class

```
1 package sl314.generic;
2
3 import javax.servlet.GenericServlet;
4 import javax.servlet.ServletRequest;
5 import javax.servlet.ServletResponse;
6 // Support classes
7 import java.io.IOException;
8 import java.io.PrintWriter;
9
10 public class HelloServlet extends GenericServlet {
11
12     public void service(ServletRequest request,
13                         ServletResponse response)
14         throws IOException {
15
16     PrintWriter out = response.getWriter();
17
18     // Generate the response
19     out.println("Hello, World!");
20     out.close();
21 }
22 }
```

First, you must import several classes and interfaces from the javax.servlet package (Lines 3–5). Next, you must extend the GenericServlet class (Line 10) and override the service method (Lines 12–13). To generate the output, you must retrieve the writer object using the getWriter method (Line 16). You can generate the response by printing to the writer stream (Line 19). Finally, close the writer stream (Line 20).

## HTTP Servlets

Usually, a development project would not create a new network protocol, but will use HTTP instead. In this situation you would not use the generic servlet API, rather you would use the interfaces and classes in the HTTP servlet API. The HTTP servlet API is located in the `javax.servlet.http` package. In this section, you will see the structure of the HTTP request and response streams, as well as the servlet API for manipulating the data in these streams.

## Hypertext Transfer Protocol

In any communication protocol, the client must transmit a request and the server should transmit some meaningful response. In HTTP, the request is some resource that is specified by a URL. If that URL specifies a static document, then the response includes the text of that document. You can think of the request and response as envelopes around the URL (plus form data) and the response text. This client-server architecture is illustrated in Figure 2-3.

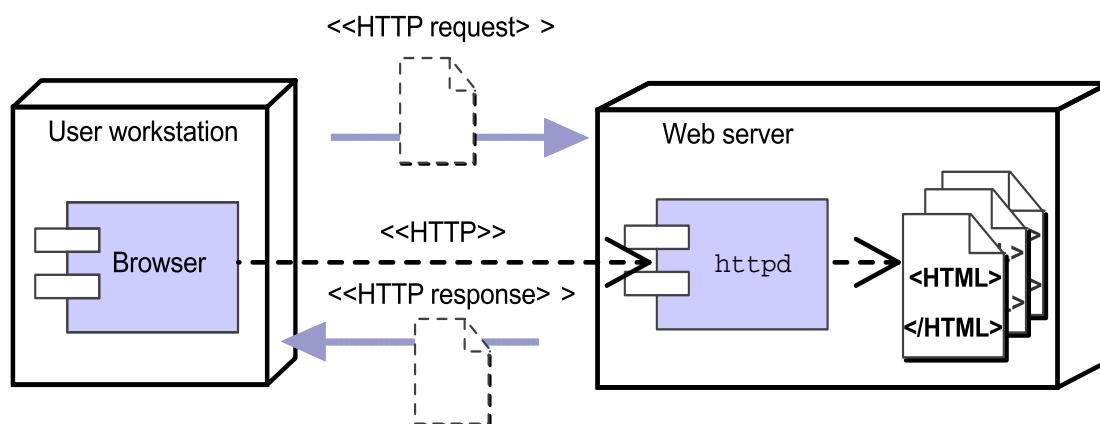


Figure 2-3     HTTP Client-Server Architecture

## HTTP GET Method

The most common HTTP method is the GET request. A GET method is used whenever the user clicks on a hyperlink in the HTML page currently being viewed. A GET method is also used when the user enters a URL into the Location field (for Netscape) or the Address field (for Internet Explorer).

## HTTP Request

The request stream acts as an envelope to the request URL and message body of the HTTP client request. The first line of the request stream is called the request line. It includes the HTTP method (usually either GET or POST), followed by a space character, followed by the requested URL (usually a path to a static file), followed by a space, and finally followed by the HTTP version number.

The request line is followed by any number of request header lines. Each request header includes the header name (for example, User-Agent), followed by a colon character (and space), followed by the value of the header. The list of headers ends with a blank line. After the blank line, there is an optional message body. An example HTTP request stream is shown in Figure 2-4.

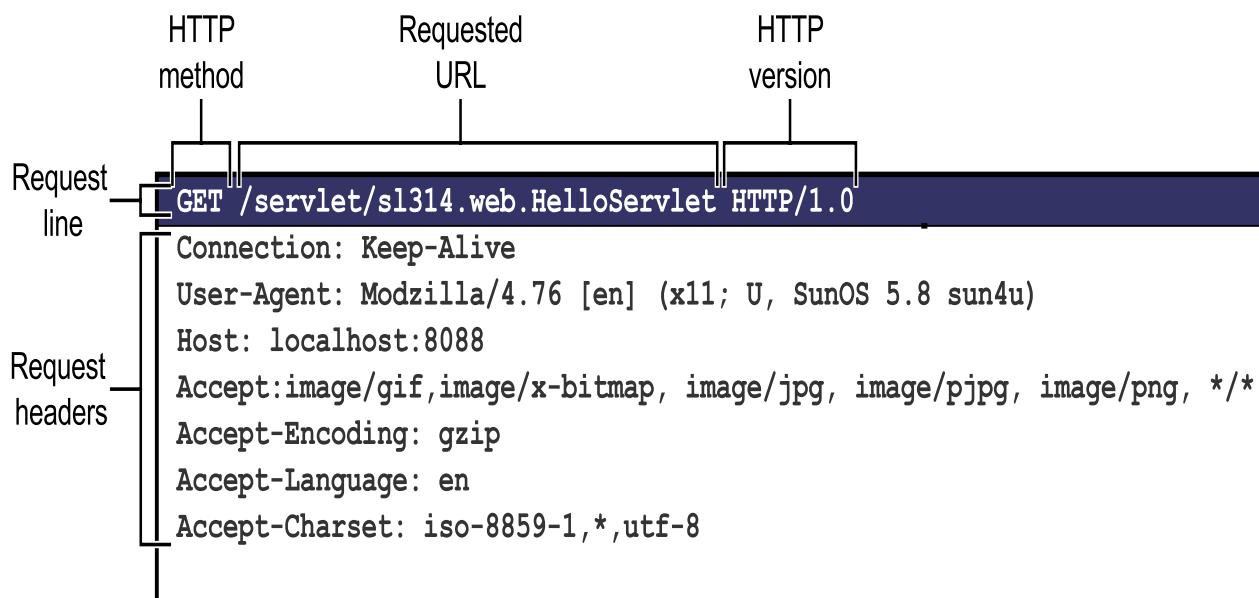


Figure 2-4 Example HTTP Request Stream

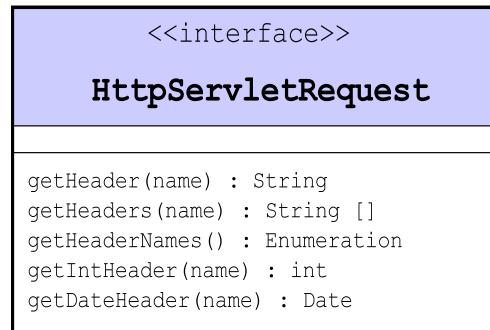
**Note** – For more information on the HTTP request stream see “Request Headers” on page B-6 in Appendix B, “Quick Reference for HTTP.”



## The HttpServletRequest API

The HTTP request information is encapsulated by the `HttpServletRequest` interface. The `getHeaderNames` method returns an enumeration of strings composed of the names of each header in the request stream. To retrieve the value of a specific header, you can use the `getHeader` method; this method returns a `String`. Some header values are strings that represent either an integer or a date. There are two convenient methods, `getIntHeader` and `getDateHeader`, that perform this conversion for you.

The `HttpServletRequest` interface is shown in Figure 2-5.



**Figure 2-5** The `HttpServletRequest` Interface



**Note** – Figure 2-5 shows only a small portion of the API for the `HttpServletRequest` interface. Later in this course, you will be introduced to more of the methods in the API.

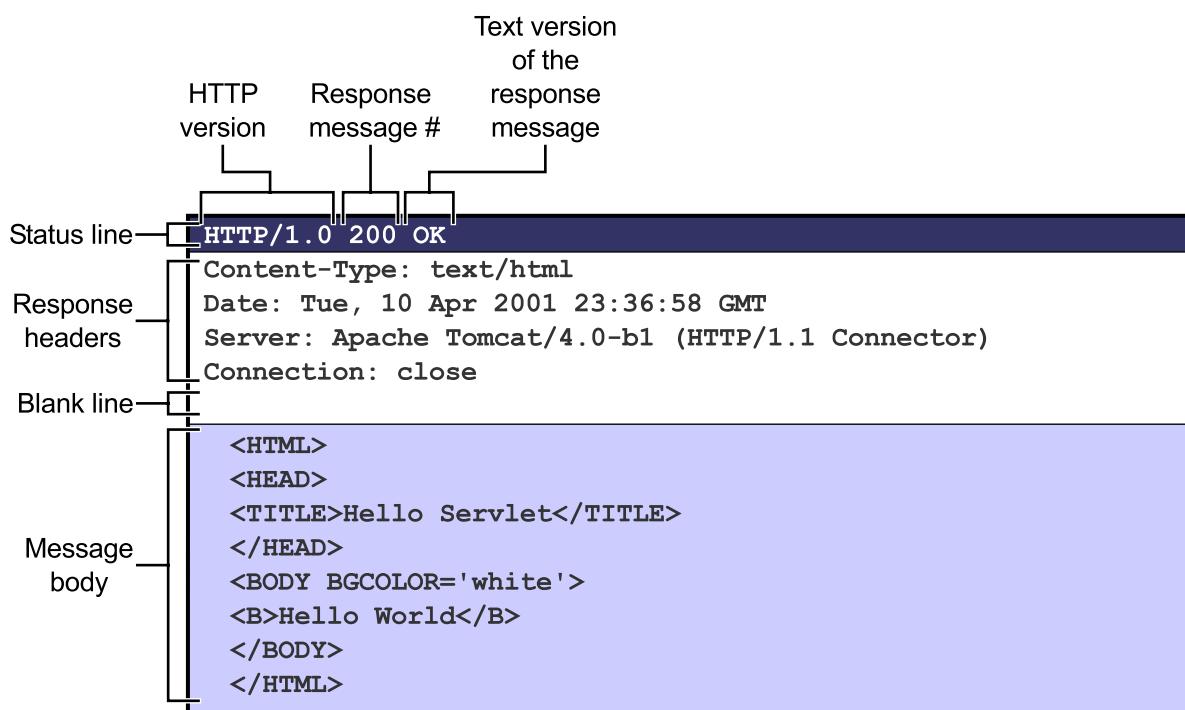
The following code highlights demonstrate a few request methods executed on the request data shown in Figure 2-4 on page 2-9.

```
String userAgent = request.getHeader("User-Agent");
// userAgent == "Mozilla/4.76 [en] (X11; U; SunOS 5.8 sun4u)"
Enumeration mimeTypes = request.getHeaders("Accept");
// first mimeTypes == "image/gif"
// second mimeTypes == "image/x-xbitmap", and so on
Enumeration headerNames = request.getHeaderNames();
// first headerNames == "Connection"
// second headerNames == "User-Agent", and so on
```

## HTTP Response

The response stream acts as an envelope to the message body of the HTTP server response. The first line of the response stream is called the status line. The status line includes the HTTP version number, followed by a space, followed by the numeric status code of the response, followed by a space, and finally followed by a short text message represented by the status code.

The status line is followed by any number of response header lines. The response headers conform to the same structure as request headers. An example HTTP response stream is shown in Figure 2-6.



**Figure 2-6** Example HTTP Response Stream

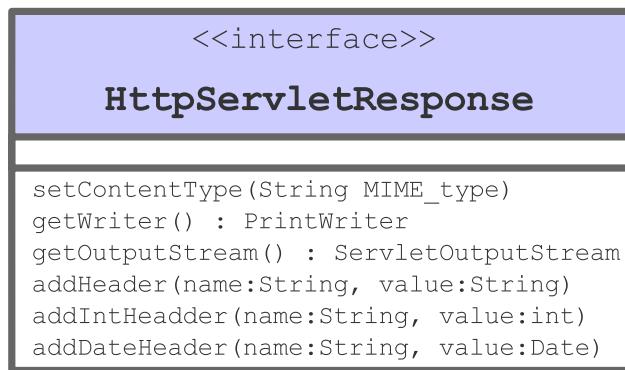
**Note** – For more information on the HTTP response stream read “Response Headers” on page B-9 in Appendix B, “Quick Reference for HTTP.”



## The `HttpServletResponse` API

The HTTP response information is encapsulated by the `HttpServletResponse` interface. You can set a response header using the `addHeader` method. If the header value you want to set is either an integer or a date, then you can use the convenient methods `addIntHeader` or `addDateHeader`.

Also, the `HttpServletResponse` interface gives you access to the body of the response stream. The response body is the data that is sent to the browser to be displayed. The response body is encapsulated in a Java technology stream object. The body stream is intercepted by the Web container and is embedded in the HTTP response stream like a letter in an envelope. The Web container is responsible for packaging the response text with the header information. The `HttpServletResponse` interface is shown in Figure 2-7.



**Figure 2-7**    The `HttpServletResponse` Interface

You need to generate the response text. You must retrieve the response body stream using either the `getWriter` or `getOutputStream` method. If you are generating an HTML response, you should use the `getWriter` method, which returns a character stream, a `PrintWriter` object. If you generate a binary response, such as a graphic image, you should use the `getOutputStream` method, which returns a binary stream, a `ServletOutputStream` object.

You must also set the *content type* of the response text. The content type defines the MIME type of the response text. It is the content type header that tells the Web browser how to render the body of the HTTP response. Examples of a MIME type include `text/plain`, `text/html`, `image/jpeg`, `image/png`, `audio/au`, and so on. The default MIME type for servlets is `text/plain`. To declare a response of any other type, you must use the `setContentType` method.



---

**Note** – Figure 2-7 on page 2-12 shows only a small portion of the API for the `HttpServletResponse` interface. Later in this course, you will be introduced to more of the methods in the API.

---

The following code highlights demonstrate a few request methods that could be used to generate the response header shown in Figure 2-6 on page 2-11.

```
response.setContentType("text/html");
response.addDateHeader("Date", new Date());
response.addHeader("Connection", "close");
```

# Web Container Architecture

Java servlets are components that must exist in a Web container. The Web container is built on top of the Java™ 2 Platform, Standard Edition (J2SE™) and implements the servlet API and all of the services required to process HTTP (and other TCP/IP) requests.

Just as NetServer was a container for generic servlets, vendor implementations of the servlet API provide an HTTP-based Web container. The architecture for a Web server that uses a Web container is illustrated in Figure 2-8.

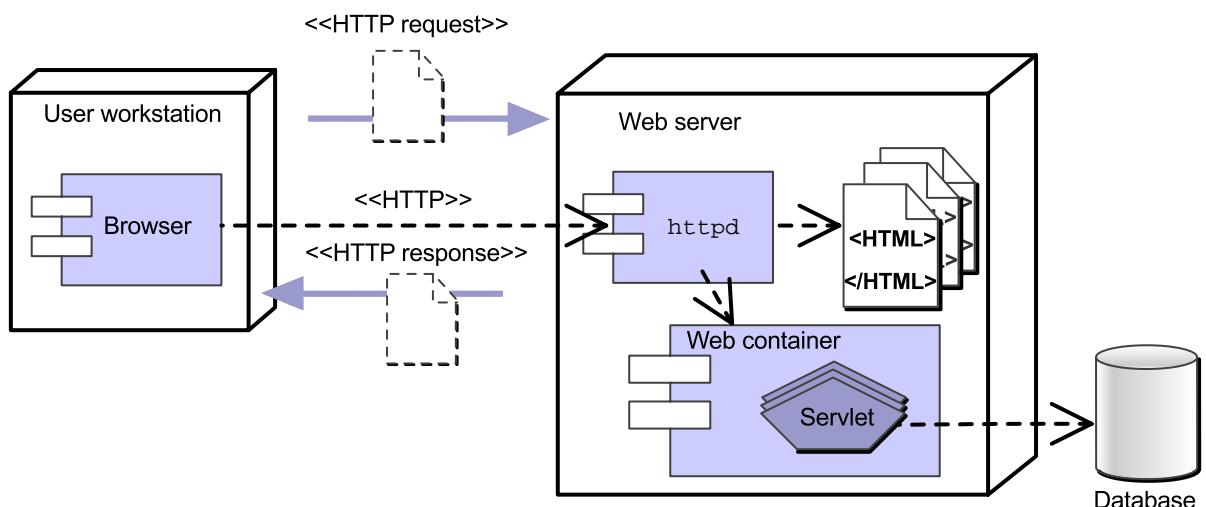


Figure 2-8     Web Container Architecture

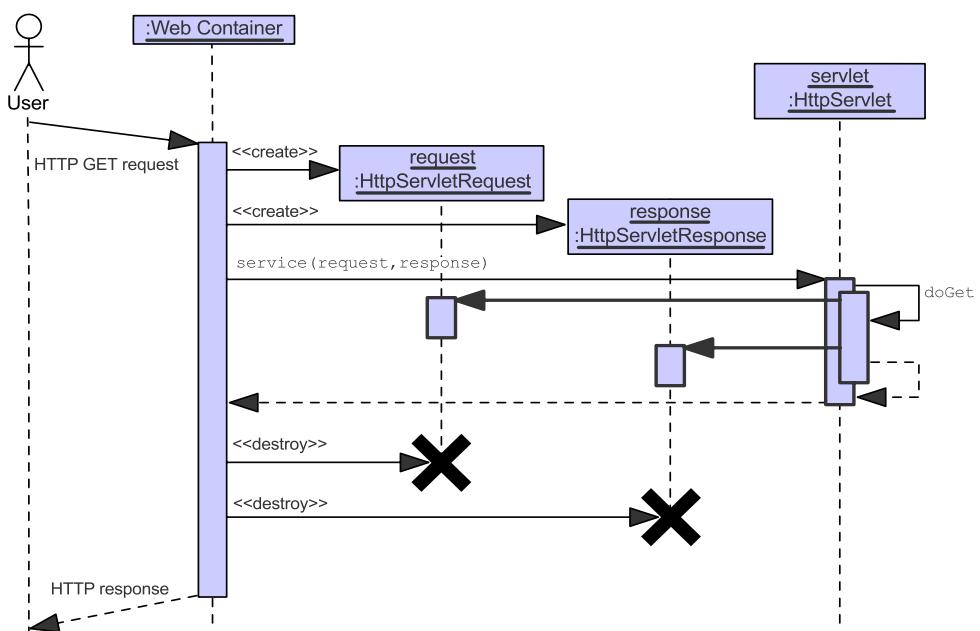
## The Web Container

A Web container may be used in conjunction with an HTTP service (as seen in Figure 2-8) or as a standalone Web server. For example, you can use Tomcat in either mode. If the Web container is used in conjunction with an HTTP service, then the Web container uses an internal communication protocol to pass the request and response stream data. The HTTP service must recognize a request for a servlet and pass it along to the Web container. If the Web container is used in the standalone mode, then it must respond to all requests including requests for static pages.

The Web container activates the servlet that matches the request URL by calling the service method on an instance of the servlet class. Specifically, the activation of the service method for a given HTTP request is handled in a separate thread within the Web container process.

## Sequence Diagram of a HTTP GET Request

The Web container converts the HTTP request and response streams into runtime objects that implement the HttpServletRequest and HttpServletResponse interfaces. These objects are then passed to the requested servlet as parameters to the service method. This process is illustrated in Figure 2-9.

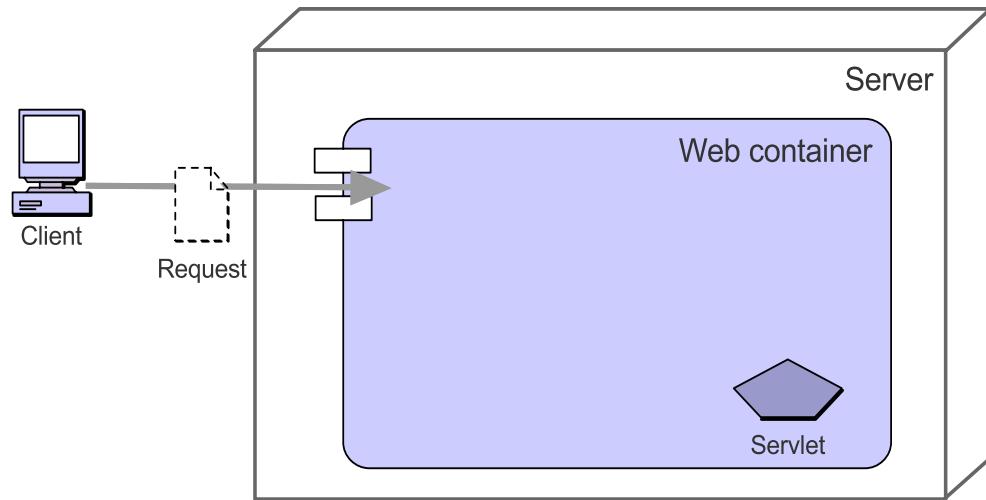


**Figure 2-9** Sequence Diagram of an HTTP GET Request

## Request and Response Process

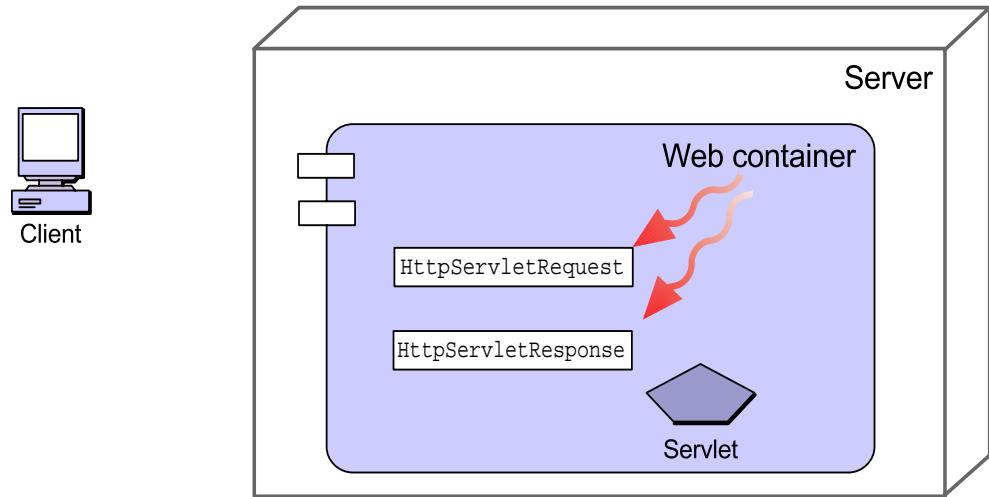
This section illustrates the Web container's request and response process using component diagrams.

The first step in this process is that the Client sends an HTTP request to the Web container. This step is illustrated in Figure 2-10.



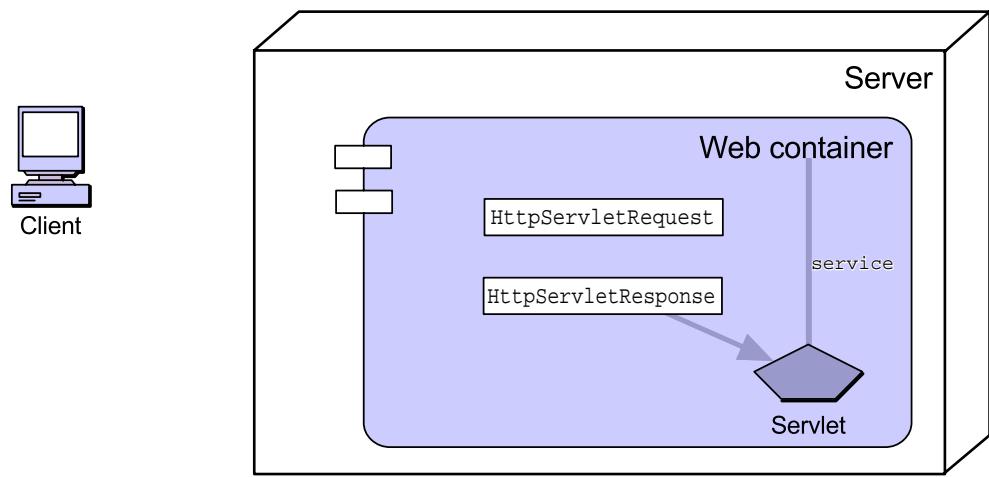
**Figure 2-10** HTTP Request Stream Is Received by the Web Container

Next, the Web container creates an object that encapsulates the data in the request stream. The Web container also creates an object that encapsulates the response stream. This step is illustrated in Figure 2-11.



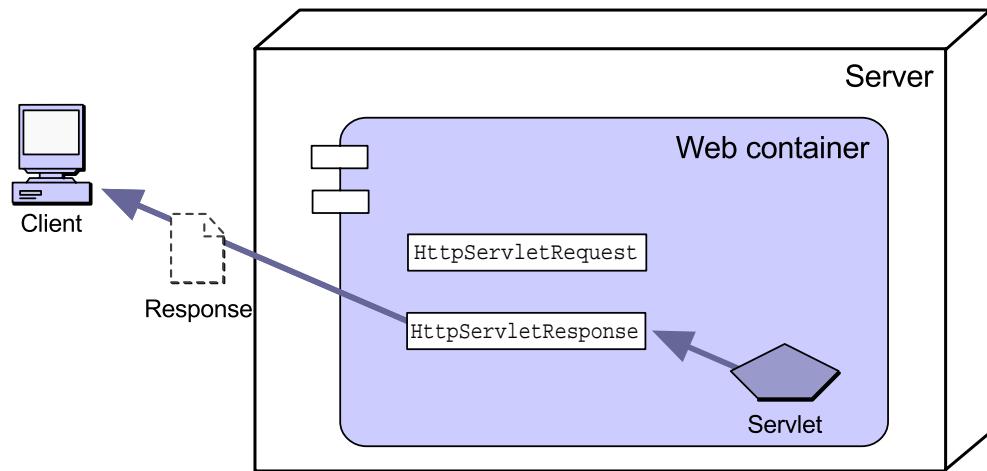
**Figure 2-11** The Web Container Constructs an `HttpServletRequest` Object From the HTTP Request Stream

Next, the Web container executes the `service` method on the requested servlet. The request and response objects are passed as arguments to this method. The execution of the `service` method occurs in a separate thread. This step is illustrated in Figure 2-12.



**Figure 2-12** The Web Container Executes the Servlet

Finally, the text of the response generated by the servlet is packaged into an HTTP response stream, which is sent to the HTTP service and forwarded to the Client. This step is illustrated in Figure 2-13.

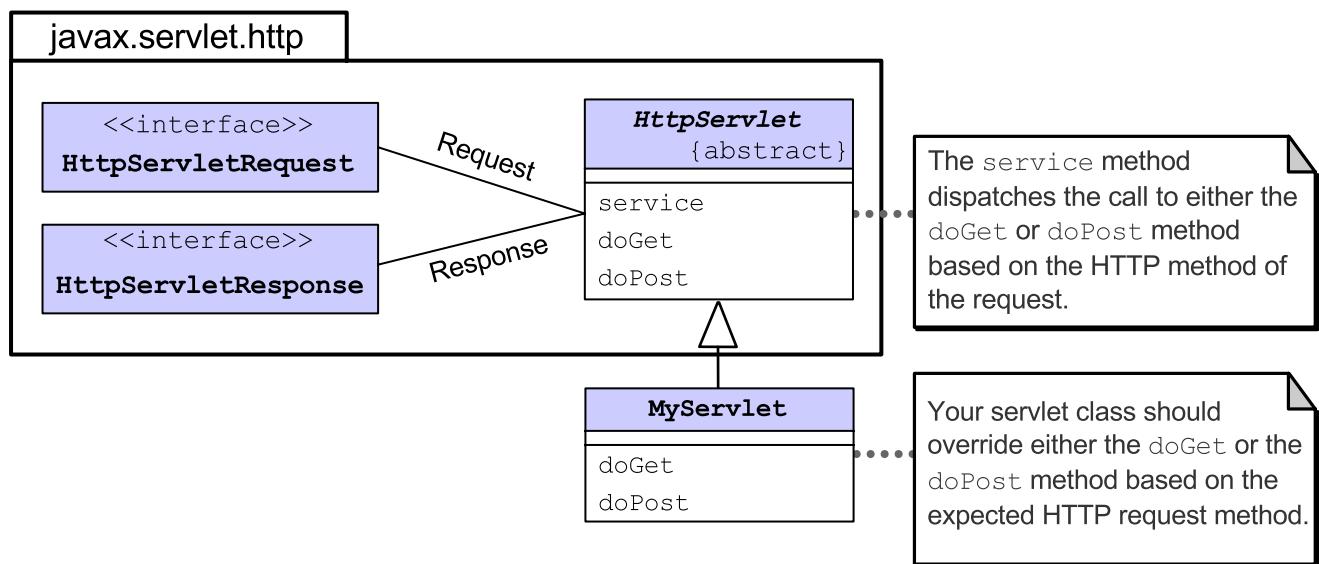


**Figure 2-13** The Servlet Generates a Dynamic Response

## The HTTP Servlet API

As an HTTP servlet developer, what do you need to do? When developing generic servlets, you must create a subclass of GenericServlet and override the service method. However, for HTTP services, the servlet API provides a special class called HttpServlet that you should extend. This class supplies a default implementation of the service method, so you should not override this method.

The HttpServlet service method looks at the HTTP method in the request stream and dispatches to a doXyz instance method based on the HTTP method. So, for example, if the servlet needs to respond to an HTTP GET request, you should override the doGet method. This API is illustrated in Figure 2-14.



**Figure 2-14** The HTTP Servlet API



**Note –** When you extend the GenericServlet class, you must override the service method. However, when you extend the HttpServlet class, you almost never override the service method. The reason is that the HttpServlet class provides a reasonable default implementation of the service method which is specifically designed to handle HTTP request methods. The service method dispatches to more specific doXyz methods.

## The HTTP HelloServlet Class

This section describes an HTTP version of the HelloServlet class. This servlet responds to an HTTP GET request method. The most important difference between the generic Hello servlet (see “The Generic HelloServlet Class” on page 2-6) and the HTTP Hello servlet is that the HTTP version overrides the `doGet` method instead of the `service` method. Another important difference is that you must set the content type to match the MIME type of the content being generated. In this example, HTML is being generated by the servlet. The HTTP HelloServlet class is shown in Code 2-2.

**Code 2-2** The HTTP HelloServlet Class

```
1 package sl314.web;
2
3 import javax.servlet.http.HttpServlet;
4 import javax.servlet.http.HttpServletRequest;
5 import javax.servlet.http.HttpServletResponse;
6 // Support classes
7 import java.io.PrintWriter;
8 import java.io.IOException;
9
10 public class HelloServlet extends HttpServlet {
11
12     public void doGet(HttpServletRequest request,
13                         HttpServletResponse response)
14         throws IOException {
15
16         // Specify the content type is HTML
17         response.setContentType("text/html");
18         PrintWriter out = response.getWriter();
19
20         // Generate the HTML response
21         out.println("<HTML>");
22         out.println("<HEAD>");
23         out.println("<TITLE>Hello Servlet</TITLE> ");
24         out.println("</HEAD>");
25         out.println("<BODY BGCOLOR='white'>"); 
26         out.println("<B>Hello, World</B>"); 
27         out.println("</BODY>"); 
28         out.println("</HTML> ");
29
30         out.close();
31     }
32 }
```

## Deploying a Servlet

To deploy a simple servlet, you must first have a running Web container on your server. You then deploy the servlet into the Web container's environment. Finally, when the Web container is running, you activate the servlet from a Web browser. The following section describes these steps in greater detail.

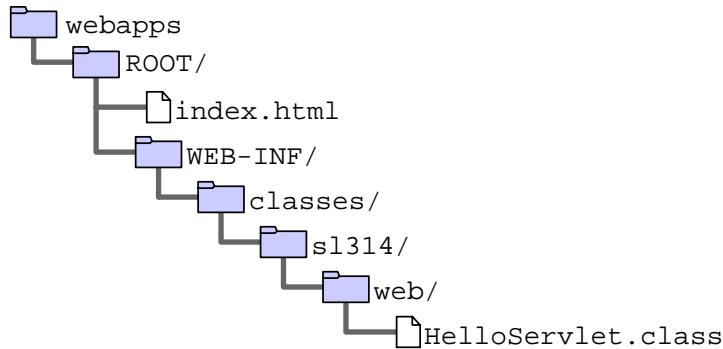
### Installing, Configuring, and Running the Web Container

The installation, configuration, and runtime procedures for any given Web container are vendor specific. Some use XML files for configuration; others use GUI front-end tools. Refer to the user's manual for your Web container for details.

### Deploying the Servlet to the Web Container

To deploy a simple servlet to the Web container, compile the source file for the servlet and store the .class file in a directory associated with the Web container. Prior to v2.2 of the Servlet specification, the directory containing the class files was vendor-specific. In the Servlet specification v2.2 and above, the class files should be stored in the ROOT/WEB-INF/classes/ directory.

To be more specific, in Tomcat, you should place simple servlets in the webapps/ROOT/WEB-INF/classes/ directory. This directory structure is illustrated in Figure 2-15. The HelloServlet.class file is stored in the sl314/web/ directory. This maps directly to the package structure for the servlet class.



**Figure 2-15** Deploying a Simple Servlet .class File

## Activating the Servlet in a Web Browser

When the servlet class file has been deployed and the Web container has been started, you can activate the servlet using a Web browser. The URL for a simple servlet takes the form:

`http://hostname:port/servlet/fully_qualified_class`

For example:

`http://localhost:8080/servlet/sl314.web.HelloServlet`

## Summary

The following are guidelines for writing a simple servlet:

- The servlet should be in a package. As Web applications grow, the package structure evolves.
- The servlet should generate a response, sending the message back to the browser in HTML. The servlet retrieves the `PrintWriter` object using the `getWriter` method on the `HttpServletResponse` interface.
- You must specify the content type of the response message. Use the `setContentType` method on the `HttpServletResponse` interface. This method takes a `String` that names a MIME type; such as "text/html."

## Certification Exam Notes

This module presented most of the objectives for Section 1, "The Servlet Model," of the Sun Certification Web Component Developer certification exam:

- 1.1 For each of the HTTP methods, GET, POST, and PUT, identify the corresponding method in the `HttpServlet` class.
- 1.2 For each of the HTTP methods, GET, POST, and HEAD, identify triggers that might cause a browser to use the method, and identify benefits or functionality of the method.
- 1.3 For each of the following operations, identify the interface and method name that should be used:
  - Retrieve HTTP request header information
  - Set an HTTP response header
  - Set the content type of the response
  - Acquire a text stream for the response
  - Acquire a binary stream for the response

For objectives 1.1 and 1.2, the POST method is described in Module 3, "Developing a Simple Servlet That Uses HTML Forms." The PUT and HEAD methods are not presented in this course. Review Section 2.1.1 "HTTP Specific Request Handling Methods," of the Servlet specification (v2.3).

## Module 3

---

# Developing a Simple Servlet That Uses HTML Forms

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the structure of HTML FORM tags
- Describe how HTML forms send data using the CGI
- Develop an HTTP servlet that accesses form data

## Relevance



**Discussion** – The following questions are relevant to developing servlets that respond to HTML form data:

- How do you create an HTML form?
  
- How do you extract the form data from the HTTP request stream?

## Additional Resources

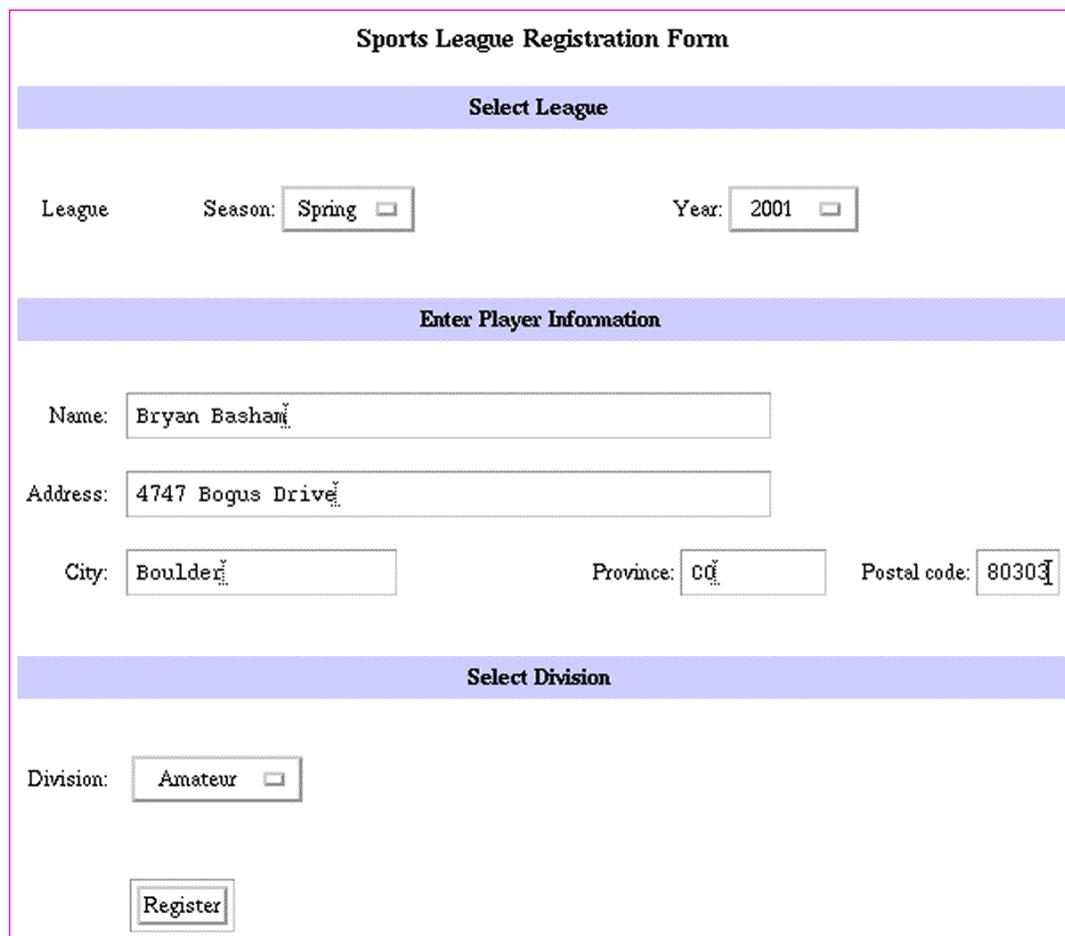


**Additional resources** – The following references provide additional information on the topics described in this module:

- Spainhour, Stephen, and Robert Eckstein. *Webmaster in a Nutshell, Second Edition*. Sebastopol: O'Reilly & Associates, Inc., 1999.
- HTML 4.01 Specification. [Online]. Available at:  
<http://www.w3.org/TR/html4/>
- Hypertext Transfer Protocol – HTTP/1.1. [Online]. Available at:  
<http://www.ietf.org/rfc/rfc2068.txt>
- Common Gateway Interface. [Online]. Available at:  
<http://www.w3.org/CGI/>

# HTML Forms

HTML forms are a fundamental part of any Web application. They provide the user interface to the application. For example, a soccer league Web application could have a league registration form as shown in Figure 3-1.



The figure shows a sample HTML form for a sports league registration. The form is titled "Sports League Registration Form" and is divided into three main sections: "Select League", "Enter Player Information", and "Select Division".

**Select League:**  
League:   
Season:    
Year:

**Enter Player Information:**  
Name:   
Address:   
City:  Province:  Postal code:

**Select Division:**  
Division:

**Figure 3-1** An Example HTML Form

The data entered in the fields of the form are transmitted with the HTTP request when a submit button is selected.

## The FORM Tag

In an HTML page, the FORM tag acts as a container for a specific set of GUI components. The GUI components are specified with input tags. There are several varieties of input tags, as you will see in the next few sections. An example HTML form is shown in Code 3-1.

**Code 3-1** Simple FORM Tag Example

```
7  <BODY BGCOLOR='white'>
8
9  <B>Say Hello Form</B>
10
11 <FORM ACTION='/servlet/sl314.web.FormBasedHello' METHOD='POST'>
12 Name: <INPUT TYPE='text' NAME='name'> <BR>
13 <BR>
14 <INPUT TYPE='submit'>
15 </FORM>
16
17 </BODY>
```

This HTML page creates a GUI for entering the user's name. This form is shown in Figure 3-2.

The figure shows a screenshot of a web browser window. The title bar says "Say Hello Form". Inside the window, there is a text input field with the placeholder "Name:" followed by the text "Bryar". Below the input field is a "Submit Query" button.

**Figure 3-2** The "Say Hello" HTML Form



**Note** – An HTML page may contain any number of forms. Each FORM tag contains the input tags for that specific form. In general, GUI components cannot be shared between forms even within the same HTML page.

---

## HTML Form Components

Web browsers support several major GUI components. These are listed in Table 3-1.

**Table 3-1** HTML Form Components

Component	Tag	Description
Textfield	<INPUT TYPE='text' ...>	Enter a single line of text.
Submit button	<INPUT TYPE='submit'>	Click the button to submit the form.
Reset button	<INPUT TYPE='reset'>	Click the button to reset the fields in the form.
Checkbox	<INPUT TYPE='checkbox' ...>	Choose one or more options.
Radio button	<INPUT TYPE='radio' ...>	Choose only one option.
Password	<INPUT TYPE='password' ...>	Enter a single line of text, but the text entered cannot be seen.
Hidden field	<INPUT TYPE='hidden' ...>	This is a static data field. This does not show up in the HTML form in the browser window, but the data is sent to the server in the CGI.
Select drop-down list	<SELECT ...> <OPTION ...> ... </SELECT>	Select one or more options from a list box.
Textarea	<TEXTAREA ...> ... </TEXTAREA>	Enter a paragraph of text.

## Input Tags

There are several types of INPUT tags. They all have three tag attributes in common:

- **TYPE** – The type of the GUI component.

This mandatory attribute specifies the type of the GUI component. The valid values of this attribute are: `text`, `submit`, `reset`, `checkbox`, `radio`, `password`, and `hidden`.

- **NAME** – The name of the form parameter.

This mandatory attribute specifies the name of the parameter that is used in the form data in the HTTP request.

- **VALUE** – The default value of the GUI component.

This is the value that is set in the GUI component when the HTML page is initially rendered in the Web browser. This is an optional attribute.

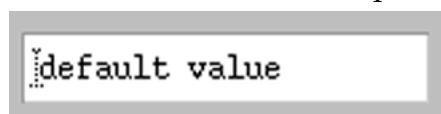
## Textfield Component

An INPUT tag of type `text` creates a Textfield GUI component in the HTML form. An example Textfield component is shown in Code 3-2.

**Code 3-2** A Textfield HTML Tag

```
<INPUT TYPE='text' NAME='name' VALUE='default value' SIZE='20'>
```

A rendered Textfield component is shown in Figure 3-3.



**Figure 3-3** The Textfield Component

A Textfield component allows the user to enter a single line of text. The data entered into this field by the user is included in the form data of the HTTP request.



**Note** – A Textfield component accepts an optional `SIZE` attribute, which allows you to change the width of the field as it appears in the Web browser screen. There is also a `MAXSIZE` attribute, which determines the maximum number of characters typed into the field.

## Submit Button Components

An INPUT tag of type submit creates a Submit button GUI component in the HTML form. An example Submit button component is shown in Code 3-3.

**Code 3-3** Submit Button HTML Tags

```
<INPUT TYPE='submit'> <BR>
<INPUT TYPE='submit' VALUE='Register'> <BR>
<INPUT TYPE='submit' NAME='operation' VALUE='Send Mail'> <BR>
```

The example Submit button components are rendered as shown in Figure 3-4.



**Figure 3-4** Submit Button Component

A Submit button components triggers an HTTP request for this HTML form. If the Submit button tag *does not* include a VALUE attribute, then the phrase "Submit Query" is used as the text of the button. If the Submit button tag *does* include a VALUE attribute, then the value of that attribute is used as the text of the button.

If the Submit button tag *does not* include a NAME attribute, then form data is not sent for this GUI component. If the Submit button tag *does* include a NAME attribute, then that name (and the VALUE attribute) is used in the form data. This feature allows you to represent multiple actions that the form can process. For example, if the third Submit button tag in Code 3-3 is clicked, then the name-value pair of "operation=Send+Mail" is included in the form data sent in the HTTP request.

## Reset Button Component

An INPUT tag of type `reset` creates a Reset button GUI component in the HTML form. An example Reset button component is shown in Code 3-4.

**Code 3-4** A Reset Button HTML Tag

```
<INPUT TYPE='reset'>
```

A rendered Reset button component is shown in Figure 3-5.



**Figure 3-5** The Reset Button Component

The Reset button component is special in the HTML form. It does not send any form data to the server. When selected, this button resets all of the GUI components in the HTML form to their default values.

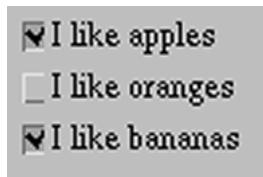
## Checkbox Component

An INPUT tag of type checkbox creates a Checkbox GUI component in the HTML form. An example Checkbox component is shown in Code 3-5.

**Code 3-5** A Checkbox HTML Tag

```
<INPUT TYPE='checkbox' NAME='fruit' VALUE='apple'> I like apples <BR>
<INPUT TYPE='checkbox' NAME='fruit' VALUE='orange'> I like oranges <BR>
<INPUT TYPE='checkbox' NAME='fruit' VALUE='banana'> I like bananas <BR>
```

A rendered Checkbox component is shown in Figure 3-6.



**Figure 3-6** The Checkbox Component

The Checkbox component allows the user to select multiple items from a set of values. For example, if “I like apples” and “I like bananas” are both checked, then two name-value pairs of “fruit=apple” and “fruit=banana” are included in the form data sent in the HTTP request. However, if no checkboxes are selected in the HTML form, then no name-value pairs are included in the form data sent in the request. You must keep this feature in mind when extracting data from the HTTP request for checkboxes.

More specifically, the Checkbox component allows the user to toggle an item from checked to unchecked or from unchecked to checked. The checked position indicates that the VALUE of that Checkbox component will be added to the form data. If there are multiple Checkbox components, a user can toggle one independently of the others.

Checkbox components are grouped together by the NAME attribute. You can develop an HTML form with multiple, independent groups of checkboxes with different names.

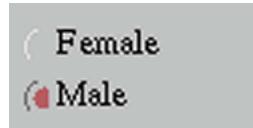
## Radio Button Component

An INPUT tag of type radio creates a Radio Button GUI component in the HTML form. An example Radio Button component is shown in Code 3-6.

**Code 3-6** A Radio Button HTML Tag

```
<INPUT TYPE='radio' NAME='gender' VALUE='F'> Female <BR>
<INPUT TYPE='radio' NAME='gender' VALUE='M'> Male <BR>
```

A rendered Radio Button component as shown in Figure 3-7.



**Figure 3-7** The Radio Button Component

The Radio Button component allows the user to select one item from a mutually exclusive set of values.

Radio Button components are grouped together by the NAME attribute. You can develop an HTML form with multiple, independent groups of radio buttons all with different names. It is the name of the radio button groups that determine the “mutually exclusive set of values” mentioned above.

## Password Component

An INPUT tag of type password creates a Password GUI component in the HTML form. An example Password component is shown in Code 3-7.

**Code 3-7** A Password HTML Tag

```
<INPUT TYPE='password' NAME='psword' VALUE='secret' MAXSIZE='16'>
```

A rendered Password component as shown in Figure 3-8.



**Figure 3-8** The Password Component

The Password component is similar to a Textfield component, but the characters typed into the field are obscured. However, the value of the field is sent in the form data “in the open,” which means that the text entered in the field is not encrypted when it is sent in the HTTP request stream.

## Hidden Field Component

An INPUT tag of type hidden creates a non-visual component in the HTML form. An example Hidden component is shown in Code 3-8.

**Code 3-8** A Hidden Field Component

```
<INPUT TYPE='hidden' NAME='action' VALUE='SelectLeague'>
```

A Hidden component is not rendered in the GUI. The value of this type of field is sent directly in the form data of the HTTP request. Think of hidden fields as constant name-value pairs sent in the form data. For example, the Hidden field in Code 3-8 will include “action=SelectLeague” in the form data of the HTTP request.

## The SELECT Tag

A SELECT tag creates a GUI component in the HTML form to select items from a list. There are two variations: single selection and multiple selection.

An example single selection component is shown in Code 3-9.

**Code 3-9** A Single Selection HTML Tag

```
<SELECT NAME='favoriteArtist'>
    <OPTION VALUE='Genesis'> Genesis
    <OPTION VALUE='PinkFloyd' SELECTED> Pink Floyd
    <OPTION VALUE='KingCrimson'> King Crimson
</SELECT>
```

A rendered single selection component is shown in Figure 3-9.



**Figure 3-9** The Single Selection Component

The OPTION tag specifies the set of items (or options) that are selectable. The SELECTED attribute determines which option is the default selection when the HTML form is initially rendered or reset.

An example multiple selection component is shown in Code 3-10.

**Code 3-10** A Multiple Selection HTML Tag

```
<SELECT NAME='sports' MULTIPLE>
    <OPTION VALUE='soccer' SELECTED> Soccer
    <OPTION VALUE='tennis'> Tennis
    <OPTION VALUE='ultimate' SELECTED> Ultimate Frisbee
</SELECT>
```

A rendered multiple selection component is shown in Figure 3-10.



**Figure 3-10** The Multiple Selection Component

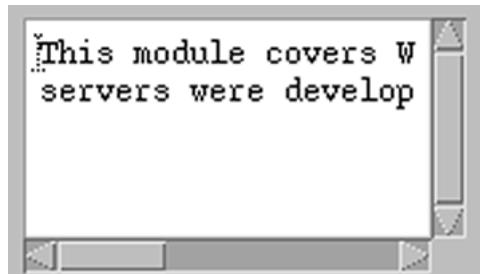
## The TEXTAREA Tag

A TEXTAREA tag creates a Textarea GUI component in the HTML form. An example Textarea component is shown in Code 3-11.

**Code 3-11** A Textarea HTML Tag

```
<TEXTAREA NAME='comment' ROWS='5' COLUMNS='70'>  
This module covers Web application basics: how browsers and Web  
servers were developed and what they do. It also...  
</TEXTAREA>
```

A rendered Textarea component is shown in Figure 3-11.



**Figure 3-11** The Textarea Component

The Textarea component allows the user to enter an arbitrary amount of text. Multiline input is permitted in the Textarea component.

## Form Data in the HTTP Request

HTTP includes a specification for data transmission used to send HTML form data from the Web browser to the Web server. The data from each field in the HTML form is packaged into name-value pairs in the form data sent in the HTTP request. The names in a name-value pair are taken from the NAME attribute of each input tag in the HTML form. The value in a name-value pair is either the character string typed into the field (for Textfield, Textarea, and Password components) or the value of the VALUE attribute of the selected item (for Checkbox, Radio Button, Single and Multiple Selection, and Hidden components).

Syntax for the data transmission of the HTML form data:

*fieldName1=fieldValue1&fieldName2=fieldValue2&...*

The name-value pairs are separated by the ampersand (&) character. The name and value within a single pair are separated by the equals (=) character. Space characters are turned into plus-sign (+) characters.

Examples:

name=Bryan

season=Spring&year=2001&name=Bryan+Basham&address=4747+  
Bogus+Drive&city=Boulder&province=Colorado&postalCode=8  
0303&division=Amateur



**Note** – A given field name can occur more than once in the collection of name-value pairs. This means that an HTML form may have multiple values for a given field. This is possible with Checkbox and Multiple Selection components. For example, in Figure 3-6 the name-value pairs “fruit=apple&fruit=banana” would be included in the form data sent in the HTTP request.

---

## HTTP GET Method Request

If the METHOD attribute of the FORM tag is set to GET, or if it is left out, then the HTTP GET method is used to send the form data to the Web server. This is shown in Line 11 of Code 3-12. The form data is appended to the URL specified in the ACTION attribute of the FORM tag. You can also enter the form data directly onto a URL either in the Location field (for Netscape Navigator) or the Address field (for Internet Explorer). You can also include form data directly in a link tag (<A HREF> tag).

**Code 3-12** FORM Tag Example Using the GET HTTP Method

```

1  <HTML>
2
3  <HEAD>
4  <TITLE>Say Hello GET Form</TITLE>
5  </HEAD>
6
7  <BODY>
8
9  <B>Say Hello GET Form</B>
10
11 <FORM ACTION='/servlet/sl314.web.FormBasedHello' METHOD='GET'>
12 Name: <INPUT TYPE='text' NAME='name'> <BR>
13 <BR>
14 <INPUT TYPE='submit'>
15 </FORM>
16
17 </BODY>
18
19 </HTML>
```

In the HTTP request stream, the form data is seen as part of the URL in the request line. This is shown in Figure 3-12.

```

GET /servlet/sl314.web.FormBasedHello?name=Bryan HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.76C-CCK-MCD Netscape [en] (X11; U; SunOS 5.8 sun4u)
Host: localhost:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: ISO-8859-1,* ,utf-8
```

**Figure 3-12** Example GET Request With Form Data

## HTTP POST Method Request

If the METHOD attribute of the FORM tag is set to POST, then the HTTP POST method sends the form data to the Web server. This is shown in Line 11 of Code 3-13. For POST requests, the form data is placed as the body of the HTTP request stream.

**Code 3-13** FORM Tag Example Using the POST HTTP Method

```
1  <HTML>
2
3  <HEAD>
4  <TITLE>Say Hello Form</TITLE>
5  </HEAD>
6
7  <BODY BGCOLOR='white'>
8
9  <B>Say Hello Form</B>
10
11 <FORM ACTION='/servlet/sl314.web.FormBasedHello' METHOD='POST'>
12 Name: <INPUT TYPE='text' NAME='name'> <BR>
13 <BR>
14 <INPUT TYPE='submit'>
15 </FORM>
16
17 </BODY>
18
19 </HTML>
```

In the HTTP request stream, the form data is included as the message body. This is shown in Figure 3-13.

```
POST /register HTTP/1.0
Referer: http://localhost:8080/model1/formEchoServer.html
Connection: Keep-Alive
User-Agent: Mozilla/4.76C-CCK-MCD Netscape [en] (X11; U; SunOS 5.8 sun4u)
Host: localhost:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: ISO-8859-1,* ,utf-8
Content-type: application/x-www-form-urlencoded
Content-length: 129

season=Spring&year=2001&name=Bryan+Basham&address=4747+Bogus+Drive&city=Bould
&province=Colorado&postalCode=80303&division=Amateur
```

**Figure 3-13** Example POST Request With Form Data

You can only activate the HTTP POST method from an HTML form.

## To GET or to POST?

The fundamental difference between the HTTP GET and POST methods is the way in which HTML form data is sent to the Web server. For a GET request, the form data is sent as part of the request URL. For a POST request, the form data is sent in the message body of the request stream. How else does the GET HTTP method differ from the POST HTTP method?

The HTTP specification v1.1 states that the GET method “should never have the significance of taking an action other than retrieval.” (RFC 2068, Section 9.1.1) The intent of this statement is that GET requests *should not* change the state of the server. This condition is called *idempotent*. For example, a GET request should not be used to update a database. You might want to use a GET request, if:

- The amount of form data is small.
- You want to allow the request to be bookmarked.

The HTTP POST method is intended to be used to modify the server. You should use a POST request, if:

- The amount of form data is large.
- The contents of the form data should not be visible in the request URL. For example, you would not want a password field shown in the URL.

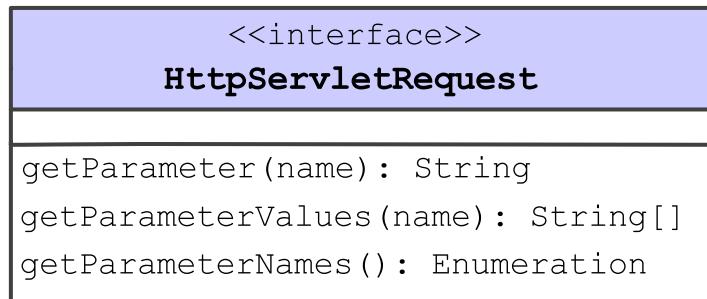
# How Servlets Access Form Data

This section describes how a servlet can access the HTML form data in the HTTP request.

## The Servlet API

The Web container parses the form data in the HTTP request stream. The `HttpServletRequest` interface provides a clean interface for retrieving the field values of the form.

You can write a servlet to access form data by using the `getParameter` method in the `HttpServletRequest` interface. This method takes the field name as an argument and returns the string for that field. Any data, numeric values for example, must be converted by the servlet. If the field does not exist, then `null` is returned. If the field contains multiple values, then use the `getParameterValues` method, which returns an array of strings. To get an enumeration of the field names in the form data, use the `getParameterNames` method. These methods are illustrated in Figure 3-14.



**Figure 3-14** CGI Methods in the `HttpServletRequest` Interface

Examples:

```
String name = request.getParameter("name");
String[] sports = request.getParameterValues("sports");
```

## The FormBasedHello Servlet

In the new Hello servlet example, there is an HTML form that requests the user's name (see Code 3-1 and Figure 3-2 on page 3-4). The action of the form executes the FormBasedHello servlet which generates a customized "Hello" response. This example is shown in Code 3-14. This servlet must then respond to both the GET and POST HTTP methods. To do that, the servlet code overrides both the doGet and doPost methods (Lines 15 and 21). Because the servlet must generate the same message for either type of HTTP request, the servlet uses a common method, generateResponse, to generate the HTTP response.

**Code 3-14**      The FormBasedHello Servlet

```
1 package s1314.web;
2
3 import javax.servlet.http.HttpServlet;
4 import javax.servlet.http.HttpServletRequest;
5 import javax.servlet.http.HttpServletResponse;
6 // Support classes
7 import java.io.IOException;
8 import java.io.PrintWriter;
9
10
11 public class FormBasedHello extends HttpServlet {
12
13     private static final String DEFAULT_NAME = "World";
14
15     public void doGet(HttpServletRequest request,
16                         HttpServletResponse response)
17             throws IOException {
18         generateResponse(request, response);
19     }
20
21     public void doPost(HttpServletRequest request,
22                         HttpServletResponse response)
23             throws IOException {
24         generateResponse(request, response);
25     }
26 }
```

The generateResponse method uses the getParameter method to extract the name field from the form data sent by the HTML form (Line 32) and uses that value to generate the dynamic “Hello” response (Line 47). This code is listed in Code 3-15.

**Code 3-15**      The FormBasedHello Servlet (continued)

```
27 public void generateResponse(HttpServletRequest request,
28                               HttpServletResponse response)
29         throws IOException {
30
31     // Determine the specified name (or use default)
32     String name = request.getParameter("name");
33     if ( (name == null) || (name.length() == 0) ) {
34         name = DEFAULT_NAME;
35     }
36
37     // Specify the content type is HTML
38     response.setContentType("text/html");
39     PrintWriter out = response.getWriter();
40
41     // Generate the HTML response
42     out.println("<HTML>");
43     out.println("<HEAD>");
44     out.println("<TITLE>Hello Servlet</TITLE>");
45     out.println("</HEAD>");
46     out.println("<BODY BGCOLOR='white'>");
47     out.println("<B>Hello, " + name + "</B>");
48     out.println("</BODY>");
49     out.println("</HTML>");
50
51     out.close();
52 }
53 }
```



---

**Note** – The conditional code in Line 33 is used to determine if the name parameter was either not sent in the HTTP request (the null value) or if the user did not enter a value in the name form field (the string is empty).

## Summary

Developing a servlet that processes an HTML form requires the following knowledge:

- HTML FORM and INPUT tags are used by the browser to send form data to a servlet.
- The ACTION attribute of the FORM tag tells the browser which servlet to activate. The METHOD attribute tells the browser which HTTP method to use.
- The NAME attribute in the INPUT tag provides the name in the form data.
- The HttpServletRequest object encapsulates the form data.
- The getParameter method is used to extract the value (as a String) of the parameter:

```
String city = request.getParameter("city");
```

Any additional data conversion, for example from a string to an integer, must be performed by the servlet.

## Certification Exam Notes

This module presented some of the objectives for Section 1, “The Servlet Model,” of the Sun Certification Web Component Developer certification exam:

- 1.1 For each of the HTTP methods, GET, POST, and PUT, identify the corresponding method in the `HttpServlet` class.
- 1.2 For each of the HTTP methods, GET, POST, and HEAD, identify triggers that might cause a browser to use the method, and identify benefits or functionality of the method.
- 1.3 For each of the following operations, identify the interface and method name that should be used:
  - Retrieve HTML form parameters from the request

The PUT and HEAD methods are not presented in this course. Review Section 2.1.1, “HTTP Specific Request Handling Methods,” of the Servlet specification (v2.3).



# Developing a Web Application Using a Deployment Descriptor

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the requirements of a robust Web application model
- Develop a Web application using a deployment descriptor

## Problems With Simple Servlets

The term *simple servlet* is artificial; it refers to the deployment strategy used in older version of the Servlet specification. This deployment strategy placed all servlet classes under the `servlets` directory. There are a number of problems with this strategy. This module describes several of these problems.

This module also introduces you to the concept of a Web application *deployment descriptor*. Using a deployment descriptor standardizes the configuration of a Web application and solves most of the problems with the simple servlet deployment strategy.

### Problems With Deploying in One Place

The simple servlet deployment strategy places all servlet class files into a single directory. This is common practice with CGI programs and scripts where the `cgi-bin` is the default deployment directory for CGI scripts.

In large Web applications, the simple servlet deployment strategy makes it hard to maintain and organize dynamic Web components. Keeping all servlets, especially unrelated servlets, in one directory is not a good organizational strategy. Frequently, a Web site is actually composed of a group of multiple, smaller Web applications, which permits the maintenance of the individual Web applications by independent teams.

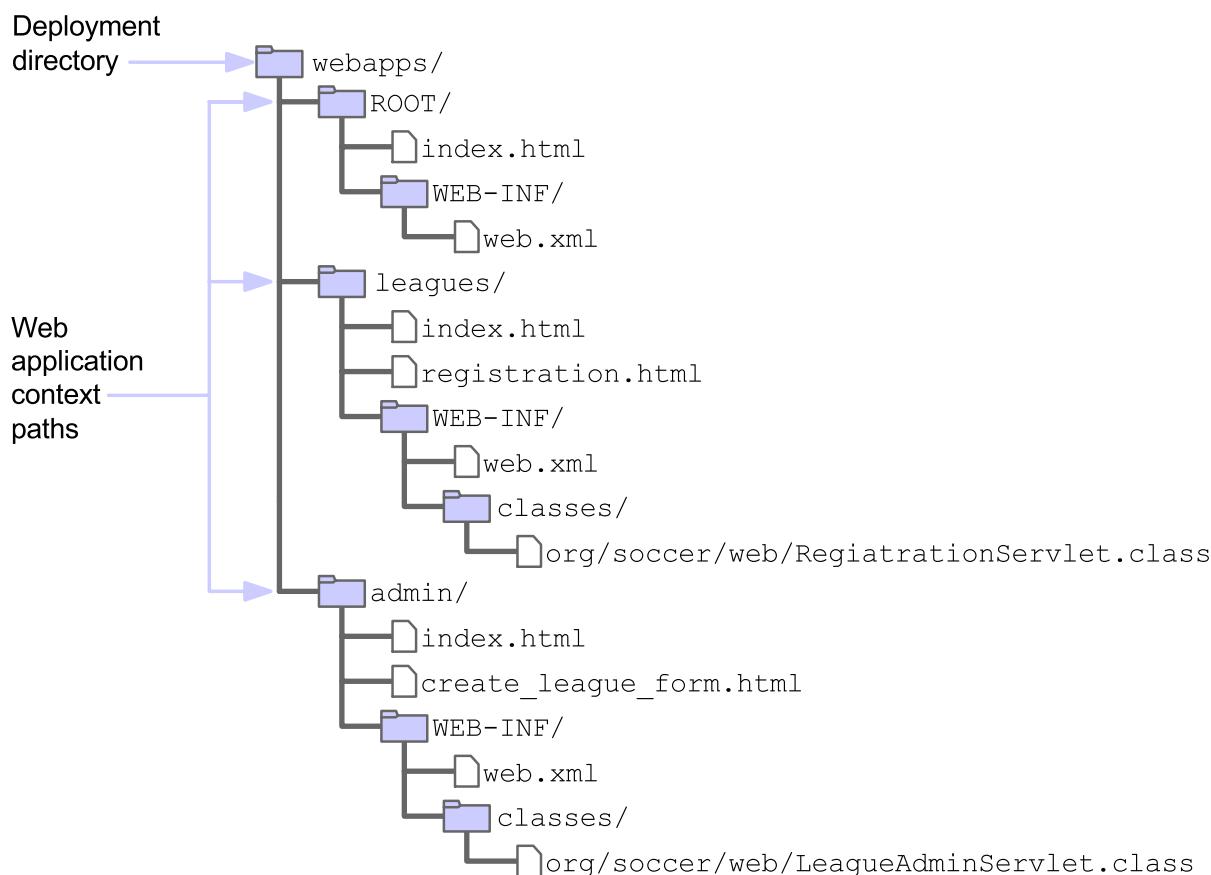
Another issue that the simple servlet strategy raises is with security. Having all of your servlets in one directory presents a risk that hackers might try to activate confidential services. The reason this is a risk is that if your secure servlets are located in a known directory on the Web server, it might be possible to hack into that directory and download the servlet code, which could then be disassembled to inspect for weaknesses in the servlet code.

## Multiple Web Applications

As of version 2.2 of the Servlet specification, a Web container must support the deployment of multiple Web applications. For example, a soccer league might want to organize its Web site into three separate Web applications:

- The main or ROOT Web application that acts as a portal to the rest of the Web site
- A leagues Web application that manages all of the player registration functionality
- An admin Web application that allows league coordinators to manage the creation of soccer leagues.

This hypothetical structure is shown in Figure 4-1.

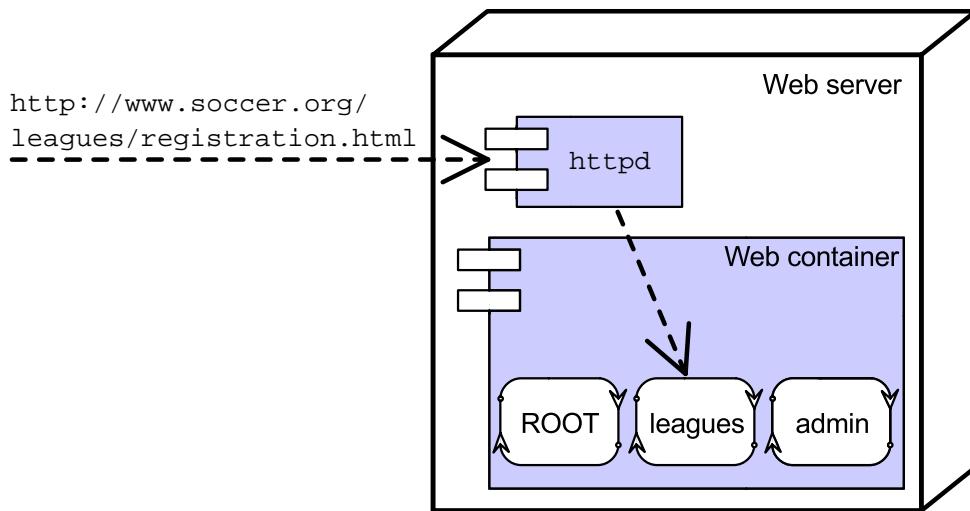


**Figure 4-1** Deployment Structure of Multiple Web Applications



**Note** – The Tomcat server uses the directory named webapps as the top-level directory that contains each Web application. This is not part of the Servlet specification and might be different in other Web container products. The Tomcat server also allows Web applications to be located in other directories and can be configured using the conf/server.xml file.

The Web container maintains an isolated runtime environment for each Web application. This is illustrated in Figure 4-2.



**Figure 4-2** Runtime Structure of Multiple Web Applications

## Web Application Context Name

As shown in Figure 4-2, the URL you use to access a Web application uses a “context name” as the first level of hierarchy in the path structure of the URL. The general syntax of a servlet Web application URL is:

`http://host:port/context/path/file`

For example:

`http://www.soccer.org/leagues/registration.html`  
`http://www.soccer.org/admin/create_league_form.html`

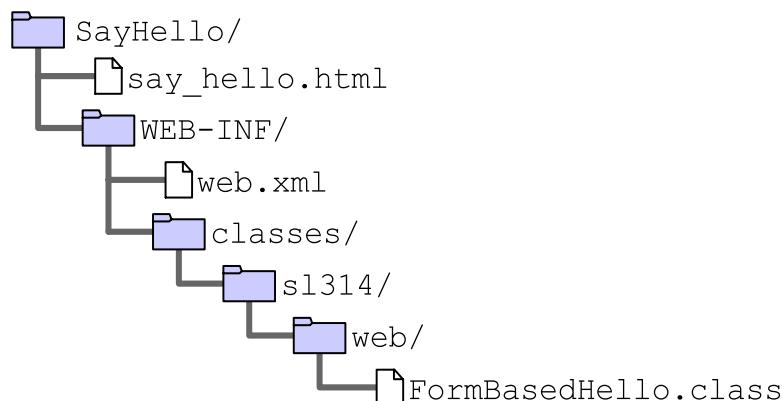
Finally, the “ROOT” Web application uses no context name. An example URL in the ROOT Web application might be:

`http://www.soccer.org/main.html`

## Problems With Servlet Naming

In the simple servlet deployment strategy, you access a servlet by using the fully-qualified servlet class name in the URL.

Suppose that you have a Web application with the context name of “SayHello” and you had a servlet with the fully qualified class name of `sl314.web.FormBasedServlet`. This is shown in Figure 4-3.



**Figure 4-3** The SayHello Web Application

In the simple servlet deployment model, the URL to access this servlet would be:

`http://localhost:8080/SayHello/servlet/sl314.web.FormBasedServlet`

There are two problems with this servlet deployment model. First, package and class names can be very long, which requires excessive typing for the page designer.

Second, including the fully qualified class name of the servlet in the static HTML pages might reveal implementation details about the Web application which could present a security risk.

## Solutions to Servlet Naming Problems

The solution provided by the Servlet specification is to provide a URL mapping for a given servlet. This mapping can be significantly shorter than the fully qualified class name and it should hide the implementation details of the Web application structure. While other pre-v2.2 servlet engines provided servlet aliases, the v2.2 Servlet specification standardizes the configuration of these aliases or mappings.

In the deployment descriptor, servlet mapping is performed in two steps. First, a *servlet definition* is configured which names a particular servlet and specifies the fully qualified Java technology class that implements that servlet. For example, you could create a servlet called "Hello" that is implemented by the `s1314.web.FormBasedHello` class. Second, a *servlet mapping* is created which identifies a URL structure that maps to the named servlet definition. For example, the "Hello" servlet could be mapped to the `/greeting` URL. This example is shown in Code 4-1.

**Code 4-1** An Example Servlet Mapping

```
12 <servlet>
13   <servlet-name>Hello</servlet-name>
14   <servlet-class>s1314.web.FormBasedHello</servlet-class>
15 </servlet>
16
17 <servlet-mapping>
18   <servlet-name>Hello</servlet-name>
19   <url-pattern>/greeting</url-pattern>
20 </servlet-mapping>
```



---

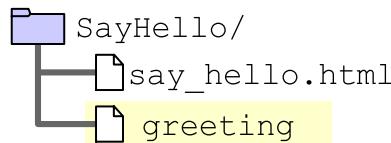
**Note** – The URL pattern must begin with a slash (/) character. This indicates that the entry point (or root) of the Web application's hierarchy. The URL pattern can place the servlet's mapped name anywhere within the logical directory hierarchy of the Web application.

---

Given the servlet definition and the corresponding servlet mapping, the user can now use the shortened URL to access this servlet. In this example, the URL would look like:

`http://localhost:8080/SayHello/greeting`

You can think of the servlet mappings as virtual files in your Web application. For example, the SayHello Web application consists of two Web components: a static HTML form called `say_hello.html` and a dynamic servlet that is mapped to the logical name `greeting`. This quasi-directory structure is illustrated in Figure 4-4.



**Figure 4-4** Quasi-Directory Structure of the SayHello Web Application

## Problems Using Common Services

In past versions of the Servlet specification, services like security, error page handling, declaration of initialization parameters, and so on, were not clearly defined. Servlet container vendors implemented these features in a variety of different ways. This lead to vendor-specific deployment strategies that made it difficult to port Web applications to different Web containers.

Since version 2.2 of the Servlet specification, the Deployer uses the deployment descriptor to configure the services used by the Web application. Throughout this course you will see examples of the services mentioned above and how they are configured using the deployment descriptor.

# Developing a Web Application Using a Deployment Descriptor

This section describes:

- The structure of the deployment descriptor file
- One possible structure for your development environment
- The structure of the deployment environment
- The structure of the Web Archive (WAR) file

## The Deployment Descriptor

The deployment descriptor is an eXtensible Markup Language (XML) file that specifies the configuration and deployment information about a specific Web application. The structure of the deployment descriptor is specified by the following Document Type Definition (DTD) file:  
[http://java.sun.com/dtd/web-app\\_2\\_3.dtd](http://java.sun.com/dtd/web-app_2_3.dtd). An example deployment descriptor is shown in Code 4-2 on page 4-9.

**Code 4-2** An Example Deployment Descriptor

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
3
4  <web-app>
5
6      <display-name>SL-314 WebApp Example</display-name>
7      <description>
8          This Web Application demonstrates a simple deployment descriptor.
9          It also demonstrates a servlet definition and servlet mapping.
10     </description>
11
12     <servlet>
13         <servlet-name>Hello</servlet-name>
14         <servlet-class>sl314.web.FormBasedHello</servlet-class>
15     </servlet>
16
17     <servlet-mapping>
18         <servlet-name>Hello</servlet-name>
19         <url-pattern>/greeting</url-pattern>
20     </servlet-mapping>
21
22 </web-app>
23
```

Line 1 declares that this is an XML file. Line 2 declares the DTD type for this XML file. The root element for the Web application DTD structure is the web-app tag (Lines 4–23). The web-app element might contain display-name and description elements as well as many others. Lines 12–15 show a servlet definition. You may have any number of servlet definitions in a Web application. They must all be grouped together in the deployment descriptor. Lines 17–20 show a servlet mapping. Again, the deployment descriptor may have any number of servlet mapping elements, and they must be grouped together.

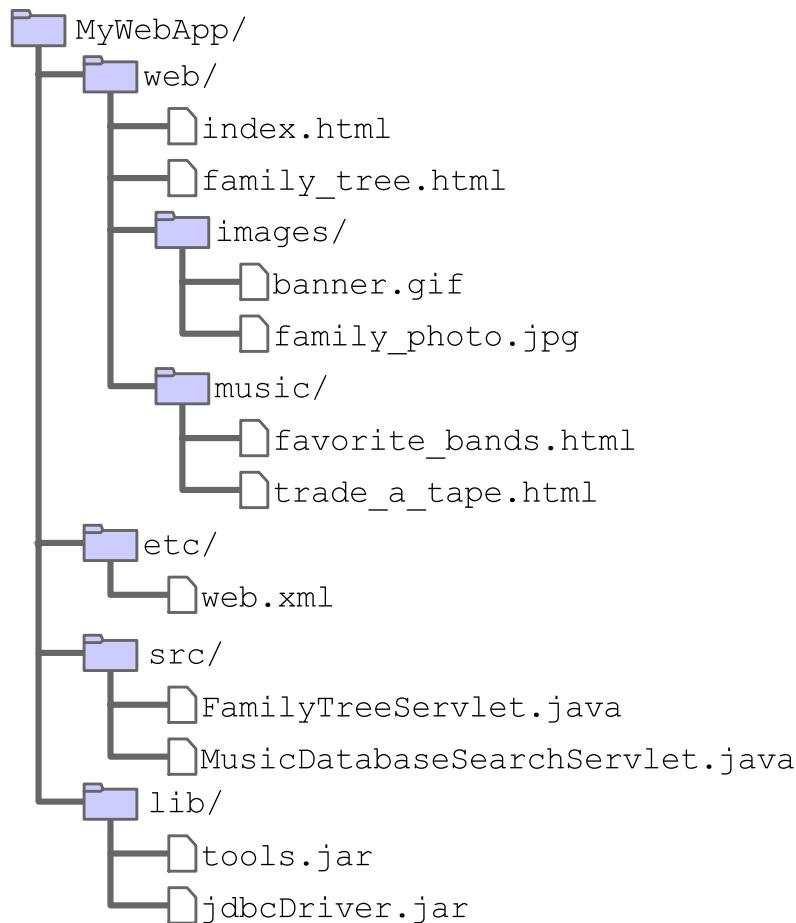
---

**Note** – For more information about XML and DTDs, see Appendix E, “Quick Reference for XML.”



## A Development Environment

There are four main elements within a Web application: the static HTML files, the servlet and related Java source files, the auxiliary library Java Archive (JAR) file, and the deployment descriptor. These elements can be grouped into the following directories: web, src, lib, etc (respectively). This development environment is illustrated in Figure 4-5.



**Figure 4-5** An Example Development Environment Structure



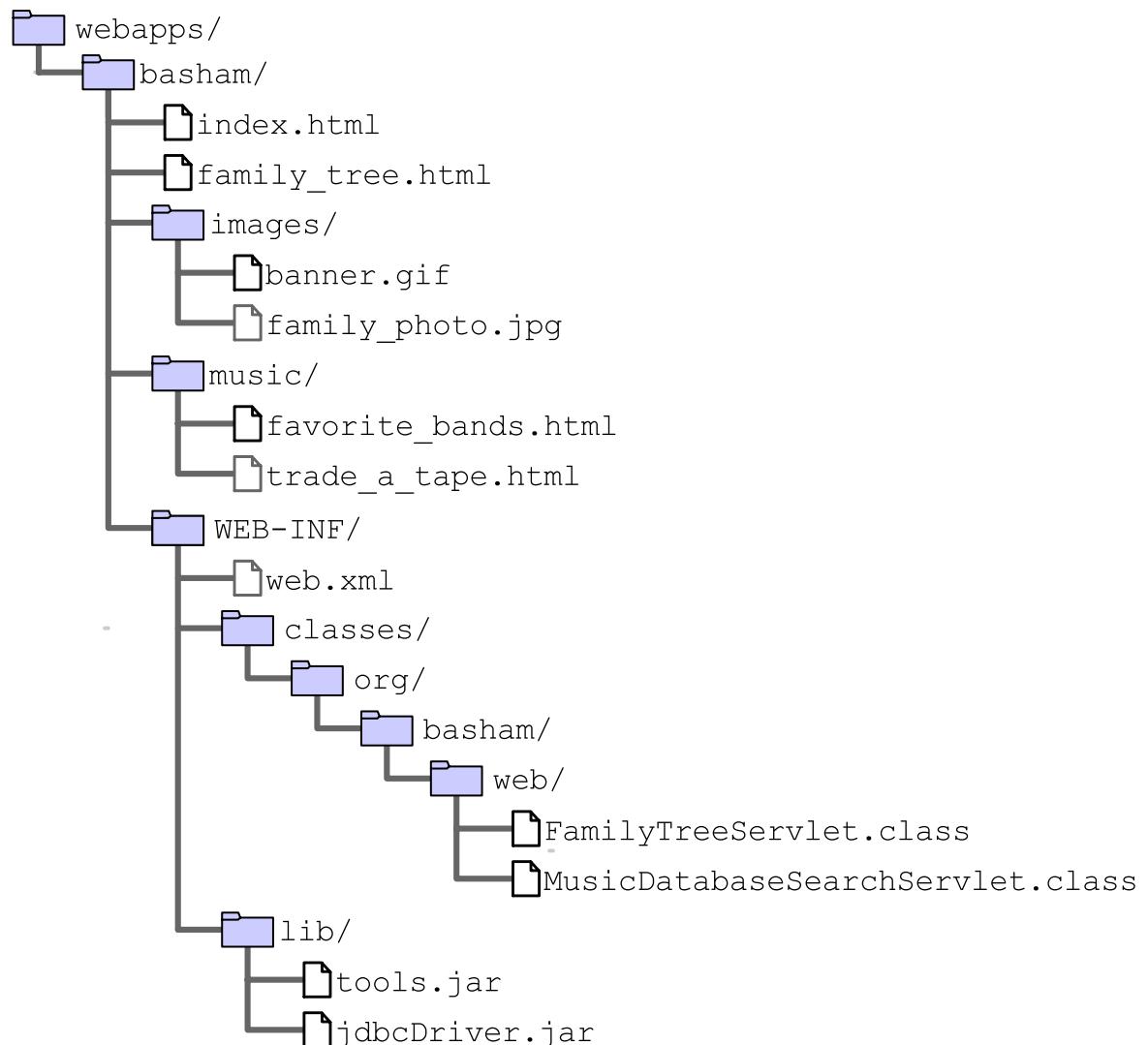
**Note** – The development environment shown in Figure 4-5 is not a development structure that is mandated in the Servlet specification. This is a structure that is promoted by the development community for the Tomcat server and is used for the examples of this course.

## The Deployment Environment

When a Web application is deployed to the Web container, the directory structure must follow a particular format:

- The static HTML files are stored in the top level directory of the Web application.
- The servlet and related Java technology class files must be stored in the `WEB-INF/classes` directory.
- The auxiliary library Java Archive (JAR) files must be stored in the `WEB-INF/lib` directory.
- The deployment descriptor must be stored in the file called `web.xml` in the `WEB-INF` directory.

The Web application deployment environment is illustrated in Figure 4-6.



**Figure 4-6** An Example Deployment Environment Structure

The Web application deployment environment is mandated in the Servlet specification. All of the directories under `basham`, but not including `WEB-INF`, are part of the Web application's URL hierarchy. You can have any number of subdirectories and any depth to the hierarchy.



**Note** – The `WEB-INF` directory *is not* accessible directly through HTTP. Therefore, no files under the `WEB-INF` directory can be specified in a URL. For the example, `http://localhost/basham/WEB-INF/web.xml` is not a valid URL for the example Web application. This feature also helps secure the servlet classes from hackers.

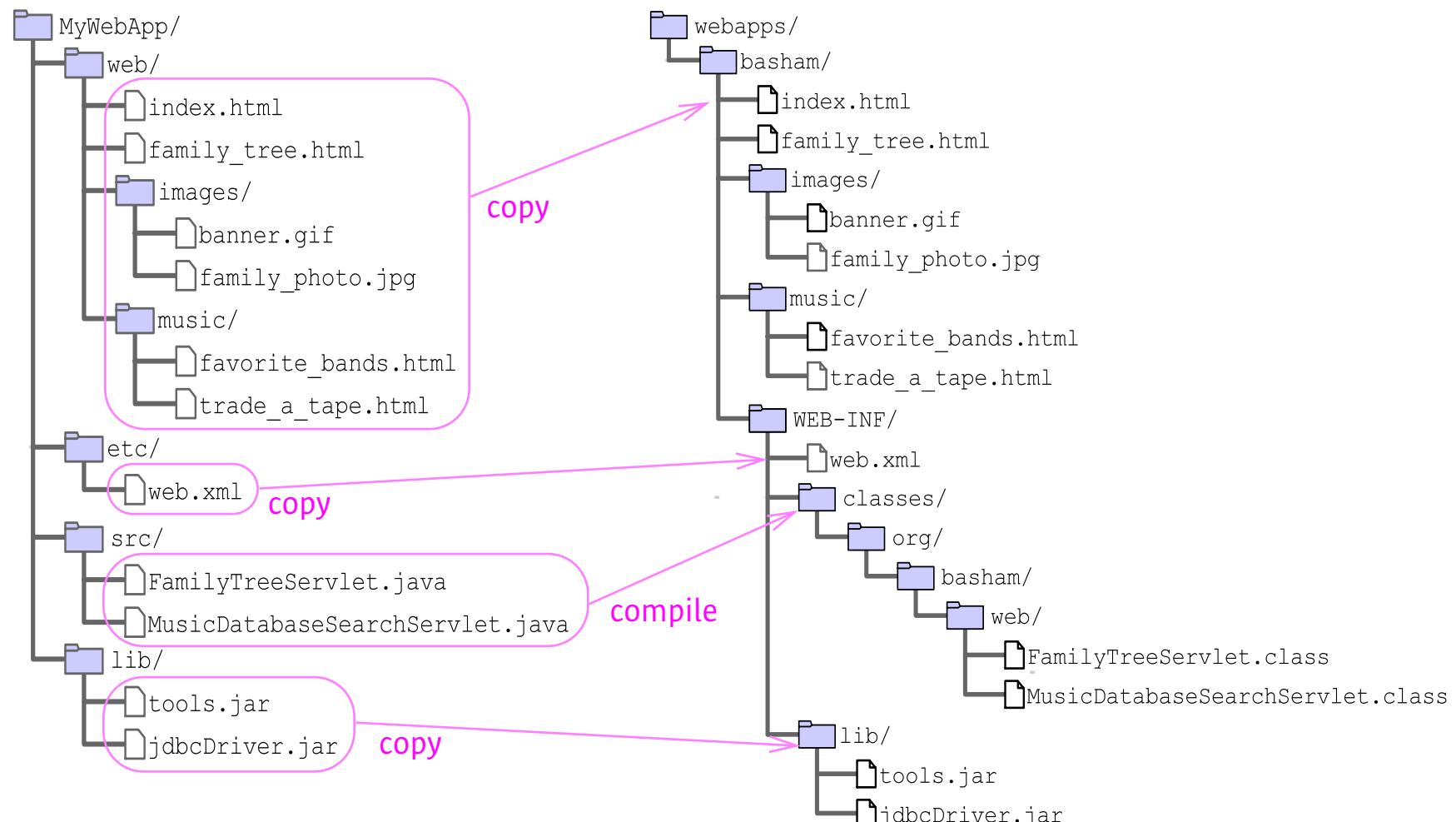
---

The mapping between the development environment and the deployment environment is a straightforward process consisting of the following steps:

1. The static HTML files are copied from the web directory to the top-level Web application directory.
2. The servlet and Java source files are compiled into the WEB-INF/classes directory.
3. The auxiliary library JAR files are copied into the WEB-INF/lib directory.
4. The deployment descriptor is copied into the WEB-INF directory.

This mapping is illustrated in Figure 4-7 on page 4-14.

## Developing a Web Application Using a Deployment Descriptor



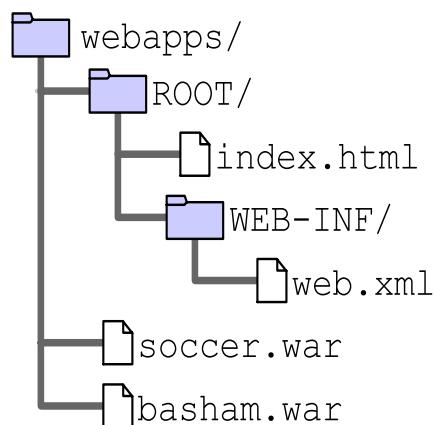
Development to Deployment Mapping

These four steps for deploying a Web application from the development to the deployment environments seem simple enough, but with a large application this process can be very tedious and time-consuming to perform manually. Typically, you should use a build tool to perform these steps.

## The Web Archive (WAR) File Format

There is another deployment strategy that is specified in the Servlet specification. A *Web Archive* is a JAR file that includes the complete Web application deployment structure in a single archive file. The file structure in the WAR file must include all static Web pages, the WEB-INF directory, the web.xml file, all libraries JAR files, and all Web component class files. You can create a WAR file by using the jar tool to create the file archive of the Web application's deployment structure.

On the Tomcat server, you can deploy a WAR file in the webapps directory. This is shown in Figure 4-8.



**Figure 4-8** Deployment of a WAR File on a Tomcat Server

## Summary

This module presented the forces that lead to the creation of the modern Web application deployment strategy which uses an XML configuration file called the deployment descriptor. Other topics covered in this module are:

- The Web application deployment descriptor solves several problems:
  - Modularizing Web applications, by using multiple, small Web applications within a single Web site
  - Hiding servlet class names, by using the deployment descriptor to create servlet definitions and URL mappings
  - Providing declarative services (such as security)
- The `web.xml` deployment descriptor is written in XML.
- The development file structure usually includes these directories: `etc`, `lib`, `src`, and `web`.
- The deployment file structure includes: Web files at the top level, `WEB-INF`, `WEB-INF/lib`, and `WEB-INF/classes` directories.
- Your build tool should copy and compile files from development to deployment.

## Certification Exam Notes

This module presented all of the objectives for Section 2, “The Structure and Deployment of Modern Servlet Web Applications,” of the Sun Certification Web Component Developer certification exam:

- 2.1 Identify the structure of a Web application and Web Archive file, the name of the Web application deployment descriptor, and the name of the directories where you place the following:
  - The Web application deployment descriptor
  - The Web application class files
  - Any auxiliary JAR files
- 2.2 Match the name with a description of purpose or functionality, for each of the following deployment descriptor elements:
  - Servlet instance
  - Servlet name
  - Servlet class
  - Initialization parameters
  - URL to named servlet mapping

For objective 2.2, the initialization parameters part is described in “Initialization Parameters,” on page 5-6 in Module 5, “Configuring Servlets.” The phrase “servlet instance” in objective 2.2 has the same meaning as “servlet definition.”

# Configuring Servlets

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the events in a servlet life cycle and the corresponding servlet API methods
- Describe servlet initialization parameters and their use with individual servlet instances
- Write servlet code to access the configured initialization parameters

## Relevance

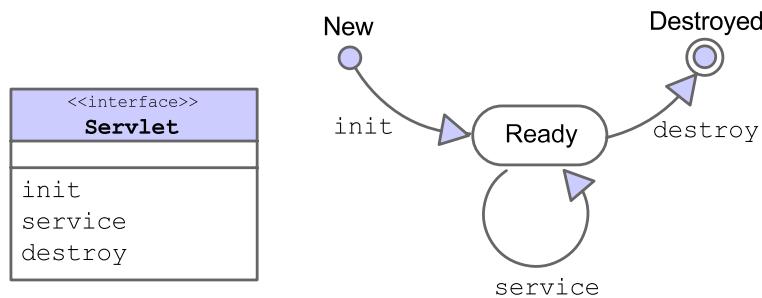


**Discussion** – The following questions are relevant to understanding how to configure a servlet within a Web application:

- There are often static pieces of information that a servlet needs but they should not be hard-coded into the servlet class. How can you configure this information in a Web application?
  
- A servlet definition is a single instance of a servlet class. How does the Web container create and prepare the servlet instance to accept HTTP requests?

# Servlet Life Cycle Overview

The Web container manages the life cycle of a servlet instance by calling three methods defined in the Servlet interface: `init`, `service`, and `destroy`. You can override these methods in your servlets when you want to manage the servlet's behavior or its resources. The servlet life cycle is illustrated in Figure 5-1.



**Figure 5-1**      Servlet Life Cycle API and State Diagram

## The `init` Life Cycle Method

The `init` method is called by the Web container when the servlet instance is first created. The Servlet specification guarantees that no requests will be processed by this servlet until the `init` method has completed.

You may override this method to perform any initialization necessary for the servlet. This may include storing initialization parameters or creating resources that are only used by that servlet instance.

**Note** – Resources that are shared by multiple instances should be configured at the Web application level. This topic is presented in Module 6, “Sharing Resources Using the Servlet Context.”



## The service Life Cycle Method

The service method is called by the Web container to process a user request. The `HttpServlet` class implements the `service` method by dispatching to `doGet` or `doPost`, depending on the HTTP request method (GET or POST). Normally, you override the `doGet` or `doPost` methods rather than the `service` method.



**Note** – The Web container executes the `service` method for each HTTP request within a unique thread within the Web container's JVM. Threading issues are discussed in Module 11, "Understanding Web Application Concurrency Issues."

---

## The destroy Life Cycle Method

The `destroy` method is called by the Web container when the servlet instance is being destroyed. The Servlet specification guarantees that all requests will be completely processed before the `destroy` method is called.

You would override the `destroy` method when you need to release any servlet-specific resources that you had created or opened in the `init` method or when you need to store the state of the servlet to some persistent file or database.

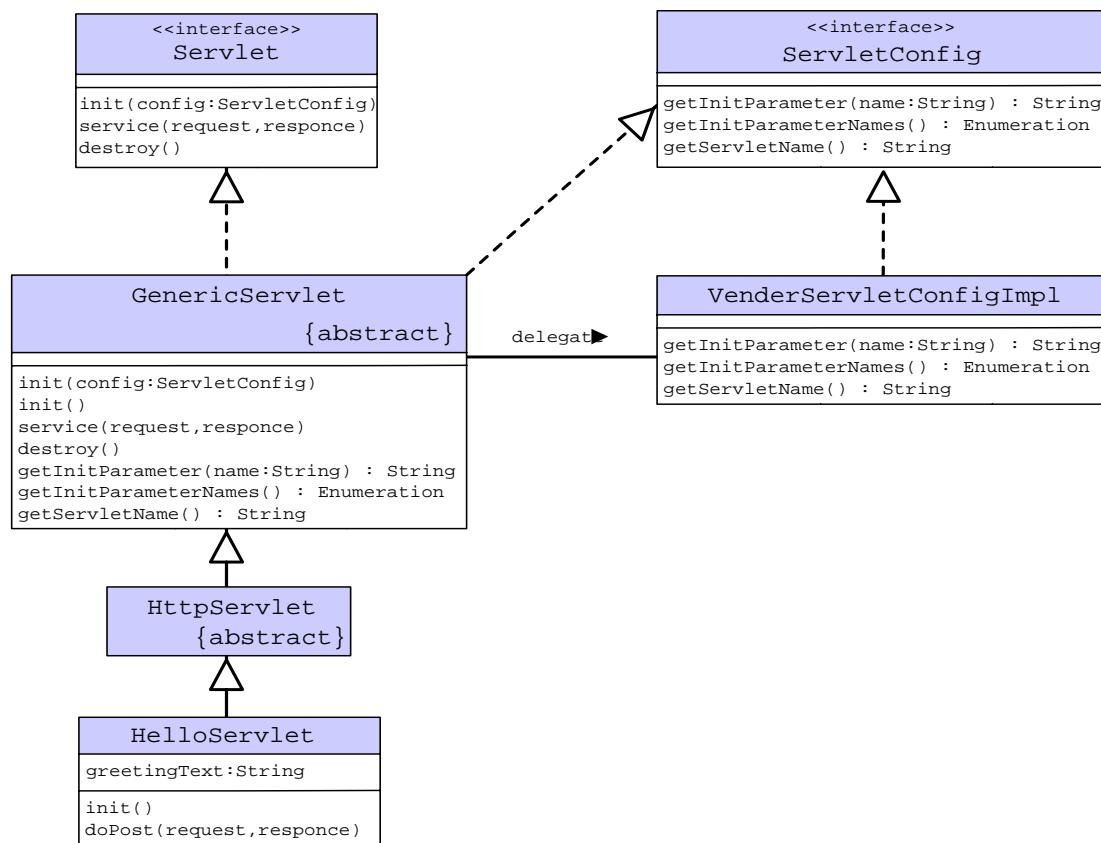
At any given moment, the Web container will only have a single servlet instance for a servlet definition specified in the deployment descriptor. However, over the life span of the Web container the servlet instance may be created and destroyed any number of times. The Web container controls this life cycle process.

# Servlet Configuration

When a servlet is initialized, a common task is to load any external resources into memory. Typically, the directory and file path information for these resources is stored as servlet initialization parameters. The Web container uses a configuration object to pass the initialization parameters to the servlet at runtime. This section describes that process.

## The ServletConfig API

The Servlet specification provides an interface called `ServletConfig` to access servlet configuration information. This information includes the servlet's name and the initialization parameters configured in the deployment descriptor. This is shown in Figure 5-2.



**Figure 5-2** Class Diagram of the `ServletConfig` Interface

The GenericServlet class implements the ServletConfig interface, which provides direct access to the configuration information. The Web container calls the init(config) method, which is handled by the GenericServlet class. This method stores the config object (delegate) and then calls the init() method. You should override the init() method rather than the init(config) method. The getInitParameter method provides the servlet with access to the initialization parameters for that servlet instance. These parameters are declared in the deployment descriptor.

## Initialization Parameters

Servlet initialization parameters are name-value pairs that are declared in the deployment descriptor. The names and values are arbitrary strings. An example initialization parameter is shown in Code 5-1.

**Code 5-1** An Example Servlet Definition With Initialization Parameters

```
11   <servlet>
12     <servlet-name>EnglishHello</servlet-name>
13     <servlet-class>sl314.web.HelloServlet</servlet-class>
14     <init-param>
15       <param-name>greetingText</param-name>
16       <param-value>Hello</param-value>
17     </init-param>
18   </servlet>
19
20   <servlet>
21     <servlet-name>FrenchHello</servlet-name>
22     <servlet-class>sl314.web.HelloServlet</servlet-class>
23     <init-param>
24       <param-name>greetingText</param-name>
25       <param-value>Bonjour</param-value>
26     </init-param>
27   </servlet>
```

In this example, the HelloServlet class is used to create two servlet definitions: EnglishHello and FrenchHello. Each of these servlet definitions is configured with a greetingText initialization parameter. The FrenchHello servlet uses “Bonjour” as its greeting phrase.

The HelloServlet class can now be written to read the greetingText initialization parameter when the servlet instance is initialized. That parameter can then be used to create a parameterized response. The HelloServlet class is shown in Code 5-2.

**Code 5-2** Configurable Hello World Servlet

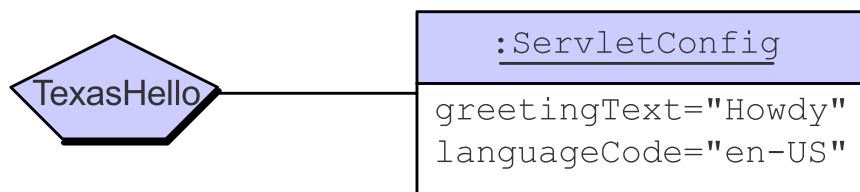
```
11 public class HelloServlet extends HttpServlet {  
12  
13     private String greetingText;  
14  
15     public void init() {  
16         greetingText = getInitParameter("greetingText");  
17         // send a message that we have the init param  
18         System.out.println(">> greetingText = " + greetingText + " ``");  
19     }  
20  
21     private static final String DEFAULT_NAME = "World";  
22  
23     public void doPost(HttpServletRequest request,  
24                         HttpServletResponse response)  
25             throws IOException {  
26  
27         // Determine the specified name (or use default)  
28         String name = request.getParameter("name");  
29         if ( (name == null) || (name.length() == 0) ) {  
30             name = DEFAULT_NAME;  
31         }  
32  
33         // Specify the content type is HTML  
34         response.setContentType("text/html");  
35         PrintWriter out = response.getWriter();  
36  
37         // Generate the HTML response  
38         out.println("<HTML>");  
39         out.println("<HEAD>");  
40         out.println("<TITLE>Hello Servlet</TITLE>");  
41         out.println("</HEAD>");  
42         out.println("<BODY BGCOLOR='white'>");  
43         out.println("<B>" + greetingText + ", " + name + "</B>");  
44         out.println("</BODY>");  
45         out.println("</HTML>");  
46     }  
47 }
```

A single servlet definition may include more than one initialization parameter by including multiple init-param elements in the deployment descriptor. For example, you could have configured the Hello World servlet with two parameters: the greetingText and a languageCode. This is shown in Code 5-3.

**Code 5-3** An Example With Multiple Initialization Parameters

```
11 <servlet>
12   <servlet-name>TexasHello</servlet-name>
13   <servlet-class>s1314.web.HelloServlet</servlet-class>
14   <init-param>
15     <param-name>greetingText</param-name>
16     <param-value>Howdy</param-value>
17   </init-param>
18   <init-param>
19     <param-name>languageCode</param-name>
20     <param-value>en-US</param-value>
21   </init-param>
22 </servlet>
```

This servlet definition will configure a servlet instance at runtime that includes these two initialization parameters. This configuration illustrated in Figure 5-3.



**Figure 5-3** The TexasHello Configuration

## Summary

This module presented the fundamental life cycle of a servlet instance as well as the technique for declaring servlet initialization parameters.

The Web container controls the life cycle of a servlet instance using these methods:

- The `init` method is called when the instance is created.
- The `service` method is called to process all requests to that servlet.
- The `destroy` method is called when the container wants to remove the servlet from service.

The `ServletConfig` object stores the initialization parameters that are configured in the deployment descriptor. The `getInitParameter` method is used to retrieve initialization parameters.

## Certification Exam Notes

This module presented several objectives for Section 1, "The Servlet Model," of the Sun Certification Web Component Developer certification exam:

- 1.3 For each of the following operations, identify the interface and method name that should be used:
  - Retrieve a servlet initialization parameter
- 1.5 Given a life cycle method: init, service, or destroy, identify correct statements about its purpose or about how and when it is invoked.

This module also presented the following objective from Section 2, "The Structure and Deployment of Modern Servlet Web Applications:"

- 2.2 Match the name with a description of purpose or functionality, for each of the following deployment descriptor elements:
  - Initialization parameters

## Module 6

---

# Sharing Resources Using the Servlet Context

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the purpose and features of the servlet context
- Develop a context listener that manages a shared Web application resource

## Relevance



**Discussion** – The following questions are relevant to understanding how to use shared global resources across all servlet definitions within a Web application:

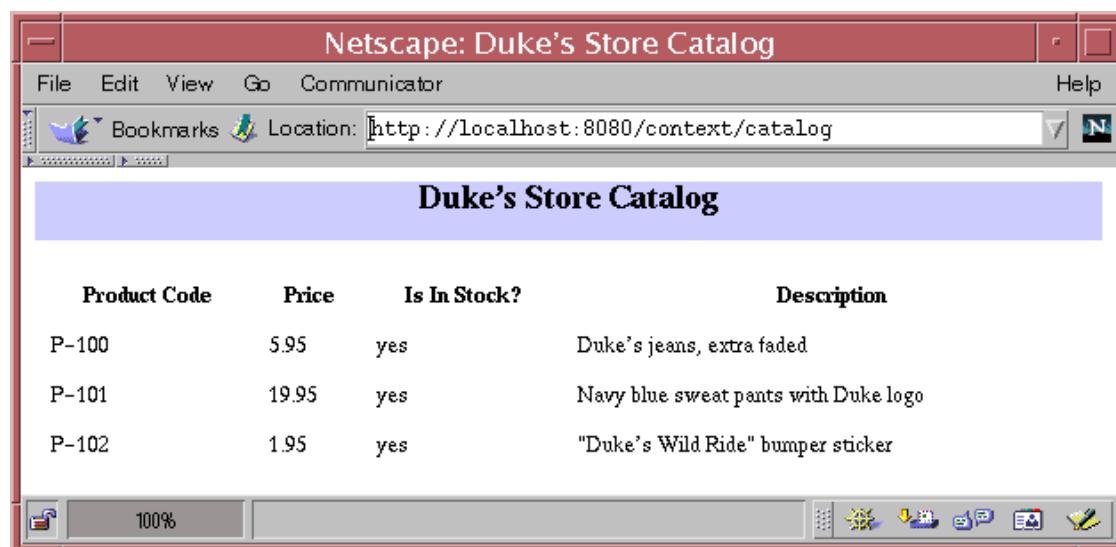
- What types of resources would you want to share across multiple servlets in a Web application?
  
- How can you store data and objects that are shared by all servlets in a Web application?

# The Web Application

A Web application is a self-contained collection of static and dynamic resources: HTML pages, media files, data and resource files, servlets (and JSP pages), and other auxiliary Java technology classes and objects. The Web application deployment descriptor is used to specify the structure and services used by a Web application. A `ServletContext` object is the runtime representation of the Web application.

## Duke's Store Web Application

The example in this module illustrates a simple storefront. The resource shared in this Web application is a catalog of products. There is only one page implemented in this system: the catalog page (see Figure 6-1). However, you could easily imagine a purchasing servlet and a checkout servlet that would need access to this shared resource.



**Figure 6-1** Duke's Store Web Application

**Note** – A better example of a shared resource than accessing a flat file is sharing a database connection pool. You will look at the design issues for creating connection pools in Module 12, “Integrating Web Applications With Databases.”

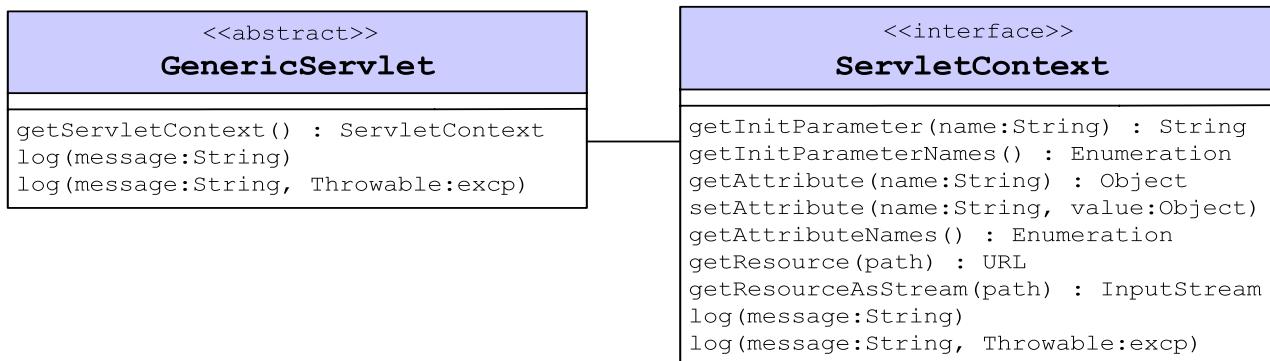


## The ServletContext API

The Duke's Store Web application makes use of four important capabilities of the ServletContext API:

- Read-only access to application scoped initialization parameters
- Read-only access to application-level file resources
- Read-write access to application scoped attributes
- Logging functionality

Every servlet within a given Web application has access to the ServletContext object for that Web application using the `getServletContext` method that is inherited from the GenericServlet class. This ServletContext API is illustrated in Figure 6-2.



**Figure 6-2** The ServletContext API

## Context Initialization Parameters

Just as each servlet definition may have one or more initialization parameters, a Web application may have one or more initialization parameters. These are also configured in the deployment descriptor using the context-param element tag. In the Duke's Store example, the catalog flat file is stored within the file structure of the Web application (hidden under the WEB-INF directory). The exact name is specified in a context initialization parameter. This example is shown in Code 6-1.

**Code 6-1** Example Context Initialization Parameter Declaration

```

3 <web-app>
4
5     <display-name>SL-314 Serlvet Context Example: Duke's Store</display-name>
6     <description>
7         This Web Application demonstrates application-scoped variables (in the
8             servlet context) and WebApp lifecycle listeners.
9     </description>
10
11    <!-- Initialize Catalog -->
12    <context-param>
13        <param-name>catalogFileName</param-name>
14        <param-value>/WEB-INF/catalog.txt</param-value>
15    </context-param>
```

Context initialization parameters are accessed using the getInitParameter method on the ServletContext object. For example, the catalogFileName initialization parameters is retrieved by Code 6-2.

**Code 6-2** Retrieving a Context Initialization Parameter

```

18 ServletContext context = sce.getServletContext();
19 String catalogFileName = context.getInitParameter("catalogFileName");
```

---

**Note –** In the first half of the module you will see only code snippets that highlight the particular topic. In the second half of this module you will see the complete code example.



## Access to File Resources

The ServletContext object provides read-only access to file resources through the `getResourceAsStream` method that returns a raw `InputStream` object. Access to text files should be decorated by the appropriate input/output (I/O) readers. In the example, the catalog file is read in from a stream at runtime (see Code 6-3).

### Code 6-3 Accessing File Resources

```
18 ServletContext context = sce.getServletContext();
19 String catalogFileName = context.getInitParameter("catalogFileName");
20 InputStream is = null;
21 BufferedReader catReader = null;
22
23 try {
24     is = context.getResourceAsStream(catalogFileName);
25     catReader = new BufferedReader(new InputStreamReader(is));
```

## Writing to the Web Application Log File

The ServletContext object provides write-only access to the log file:

- The `log(String)` method writes a message to the log file.
- The `log(String, Throwable)` method writes a message and the stack trace of the exception or error to the log file.

Web containers must support a separate log file for each Web application.

In the example, the Web application logs the fact that the catalog file was read in and that the program has initialized the `ProductList` object. This is shown in Code 6-4.

### Code 6-4 Writing to a Log File

```
43
44     context.log("The ProductList has been initialized.");
45
```

## Accessing Shared Runtime Attributes

The `ServletContext` object provides read-write access to attributes shared across all servlets through the `getAttribute` and `setAttribute` methods.

In the example, the Web application needs to store the `ProductList` catalog object in the `ServletContext` object to make the catalog available to all servlets in the Web application. This is shown in Code 6-5.

### Code 6-5 Storing an Application Scoped Attribute

```
41 // Store the catalog as an application-scoped attribute  
42 context.setAttribute("catalog", catalog);
```

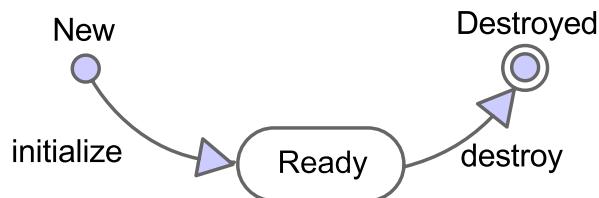
When the catalog has been stored in the `ServletContext` object, any servlet can access that attribute using the `getAttribute` method. In the example, the `ShowProductList` servlet accesses the catalog to generate the HTML table of products. This is shown in Code 6-6.

### Code 6-6 Retrieving an Application Scoped Attribute

```
13 public class ShowProductList extends HttpServlet {  
14  
15     public void doGet(HttpServletRequest request,  
16                         HttpServletResponse response)  
17         throws IOException {  
18     ServletContext context = getServletContext();  
19     ProductList catalog = (ProductList) context.getAttribute("catalog");  
20     Iterator items = catalog.getProducts();  
21  
22     // Specify the content type is HTML  
23     response.setContentType("text/html");  
24     PrintWriter out = response.getWriter();
```

## The Web Application Life Cycle

Just as servlets have a life cycle, so do Web applications. When the Web container is started, each Web application is initialized. When the Web container is shut down, each Web application is destroyed. This life cycle is illustrated in Figure 6-3.



**Figure 6-3** Web Application Life Cycle State Diagram

You can create “listener” objects that are triggered by these events. In the Duke’s Store example, a context listener is used to initialize the catalog resource.

You can create implementations of the `ServletContextListener` interface to listen to the Web application life cycle events. Associated with this listener interface is the `ServletContextEvent` class, which provides access to the `ServletContext` object. This is illustrated in Figure 6-4.



**Figure 6-4** The Web Application Lifecycle Listener API

The `ServletContextListener` interface is typically used to initialize heavy-weight, shared resources; for example, a JDBC™ connection pool or a business object (like a catalog) that should be constructed once and reused throughout the Web application.

## Duke's Store Example

In the Duke's Store example, a context listener called `InitializeProductList` is used to read the catalog file (specified in a context initialization parameter) as a file resource. The catalog data in the file is stored as a list of products, one product per line with a bar character (|) as a field delimiter. When the `ProductList` object is populated from the file, the listener stores that object as an attribute in the `ServletContext` object. This is shown in Code 6-7 on page 6-10.

### Code 6-7 The InitializeProductList Example

```
15 public class InitializeProductList implements ServletContextListener {  
16  
17     public void contextInitialized(ServletContextEvent sce) {  
18         ServletContext context = sce.getServletContext();  
19         String catalogFileName = context.getInitParameter("catalogFileName");  
20         InputStream is = null;  
21         BufferedReader catReader = null;  
22  
23         try {  
24             is = context.getResourceAsStream(catalogFileName);  
25             catReader = new BufferedReader(new InputStreamReader(is));  
26             String productString;  
27             ProductList catalog = new ProductList();  
28  
29             // Parse the catalog file to construct the product list  
30             while ( (productString = catReader.readLine()) != null ) {  
31                 StringTokenizer tokens = new StringTokenizer(productString, "|");  
32                 String code = tokens.nextToken();  
33                 String price = tokens.nextToken();  
34                 String quantityStr = tokens.nextToken();  
35                 int quantity = Integer.parseInt(quantityStr);  
36                 String description = tokens.nextToken();  
37                 Product p = new Product(code, price, quantity, description);  
38                 catalog.addProduct(p);  
39             }  
40  
41             // Store the catalog as an application-scoped attribute  
42             context.setAttribute("catalog", catalog);  
43  
44             context.log("The ProductList has been initialized.");  
45  
46             // Log any exceptions  
47         } catch (Exception e) {  
48             context.log("Exception occurred while parsing catalog file.", e);  
49  
50             // Clean up resources  
51         } finally {  
52             if ( is != null ) {  
53                 try { is.close(); } catch (Exception e) {}  
54             }  
55             if ( catReader != null ) {  
56                 try { catReader.close(); } catch (Exception e) {}  
57             }  
58         }  
59     }  
}
```

## Configuring Servlet Context Listeners

You can create as many context listeners as you need in your Web application. The Web container creates a single instance of each context listener. As each Web application life cycle event (initialization or destruction) occurs, the Web container calls the appropriate event method (`contextInitialized` or `contextDestroyed`) on each listener object.

The context listeners must be configured in the deployment descriptor for the Web container to realize that these listeners exist. This is done using the `listener` element tag. This is shown in Code 6-8.

**Code 6-8** Example Context Listener Declaration

```

3 <web-app>
4
5   <display-name>SL-314 Serlvet Context Example: Duke's Store</display-name>
6   <description>
7     This Web Application demonstrates application-scoped variables (in the
8       servlet context) and WebApp lifecycle listeners.
9   </description>
10
11  <!-- Initialize Catalog -->
12  <context-param>
13    <param-name>catalogFileName</param-name>
14    <param-value>/WEB-INF/catalog.txt</param-value>
15  </context-param>
16  <listener>
17    <listener-class>sl314.dukestore.InitializeProductList</listener-class>
18  </listener>
19  <!-- END: Initialize Catalog -->
```

The `listener` elements must come after any `context-param` elements and before any servlet definitions. You can declare any number of listeners. The order in the deployment descriptor determines the order in which the listeners are invoked by the Web container; at shutdown, that order is reversed. During Web application startup, all listeners will be invoked and completed before any servlet requests are processed.

---

**Note** – Context listeners are new to the Servlet specification version 2.3. Some Web container vendors might not yet support this feature.



## Summary

This module presented the `ServletContext` interface and how it is used to share common resources between all servlets within a Web application. The `ServletContext` object reflects the Web application as a runtime object and provides the following capabilities:

- Access to context initialization parameters using the `getInitParameter` method
- Access to file resources using the `getResourceAsStream` method
- Logging using the `log(message)` and `log(message, exception)` methods
- Access to shared attributes using the `getAttribute` and `setAttribute` methods

The `ServletContextListener` interface provides you with a tool to respond to Web application life cycle events.

# Certification Exam Notes

This module presented most of the objectives for Section 3, “The Servlet Container Model,” of the Sun Certification Web Component Developer certification exam:

- 3.1 Identify the uses for and the interfaces (or classes) and methods to achieve the following features:
  - Servlet context initialization parameters
  - Servlet context listener
  - Servlet context attribute listener
  - Session attribute listeners
- 3.2 Identify the Web application deployment descriptor element name that declares the following features:
  - Servlet context initialization parameters
  - Servlet context listener
  - Servlet context attribute listener
  - Session attribute listeners
- 3.3 Distinguish the behavior of the following in a distributable Web application:
  - Servlet context initialization parameters
  - Servlet context listener
  - Servlet context attribute listener
  - Session attribute listeners

For objectives 3.1 and 3.2, the “servlet context attribute listener” and the “session attribute listeners” topics are not presented in this course. Objective 3.3 is not described at all in this course. Review the Servlet specification (v2.3) for more details.

Also presented in this module is:

- 1.4 Identify the interface and method to access values and resources and to set object attributes within the following three Web scopes: *request, session, context*.



# Developing Web Applications Using the MVC Pattern

---

## Objectives

Upon completion of this module, you should be able to:

- List the limitations of a “simple” Web application
- Develop a Web application using a variation on the Model-View-Controller (MVC) pattern

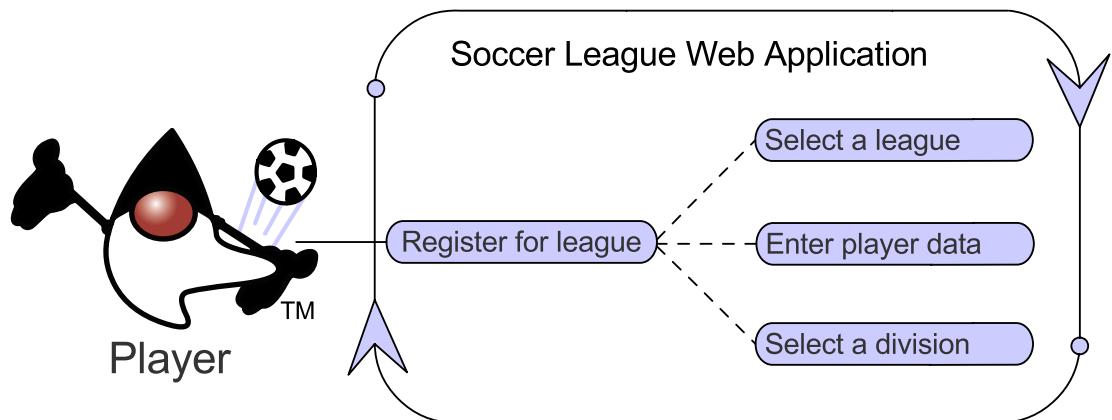
## Activities of a Web Application

A Web application interacts with a user on a Web browser through HTTP requests. Some of these requests might require shared resources, but at the lowest level the Web application must process each HTTP request using a servlet. The processing usually includes the following steps:

- Verify HTML form data (if any)
- Send an Error page if data fails verification checks
- Process the data, which may store persistent information
- Send an Error page if processing fails
- Send a Response page if processing succeeds

# The Soccer League Example

The example used in this module is of a Soccer League Web site. The main Use Case is player registration. This is illustrated in Figure 7-1.



**Figure 7-1** Soccer League Registration Use Case

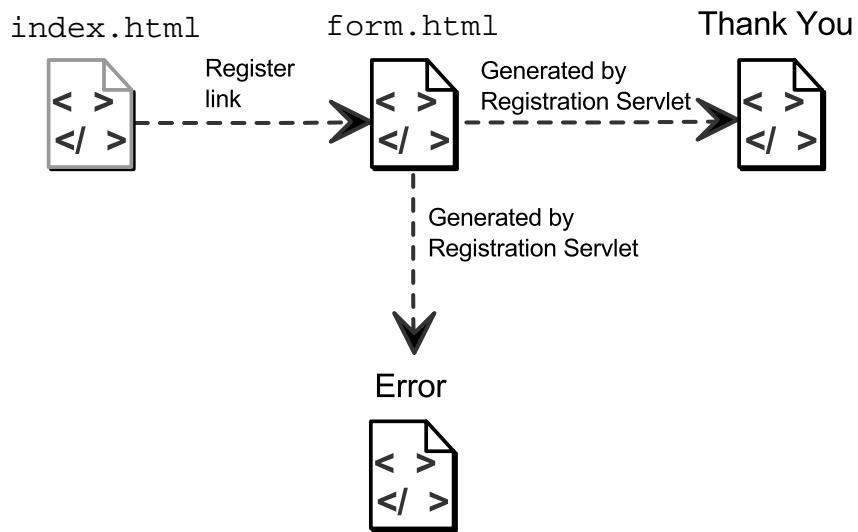
A player selects the league (year and season), enters her name and address information, and selects the division (amateur, semi-pro, or professional) that she want to play in. The Web application must:

- Verify the league, player, and division data in the HTML form
- Store the player and registration information
- Send the appropriate page (Error or Thank You)

To do its work the registration servlet must keep track of a list of valid leagues, which is stored in a flat file.

## Page Flow of the Soccer League Example

An important aspect of any Web application is the sequence of pages that the user can visit during the Use Case. In the registration Use Case, the user starts at the main page (`index.html`) and selects the “Register” link. The Web container returns the static HTML registration form page shown in Figure 7-3 on page 7-5. The user fills out the registration form and selects the Submit button. In the Web container this activates the registration servlet. Based on the results of that activity, the servlet either generates an Error page or a Thank You page. This page flow is illustrated in Figure 7-2.



**Figure 7-2** Soccer League Page Flow

The registration HTML form looks like Figure 7-3.

The screenshot shows a registration form titled "Sports League Registration Form". It is divided into three main sections: "Select League", "Enter Player Information", and "Select Division".

**Select League:** Fields for "League" (dropdown menu), "Season" (dropdown menu set to "Spring"), and "Year" (dropdown menu set to "2001").

**Enter Player Information:** Fields for "Name" (text input containing "Bryan Basham"), "Address" (text input containing "4747 Bogus Drive"), "City" (text input containing "Boulder"), "Province" (dropdown menu set to "CO"), and "Postal code" (text input containing "80303").

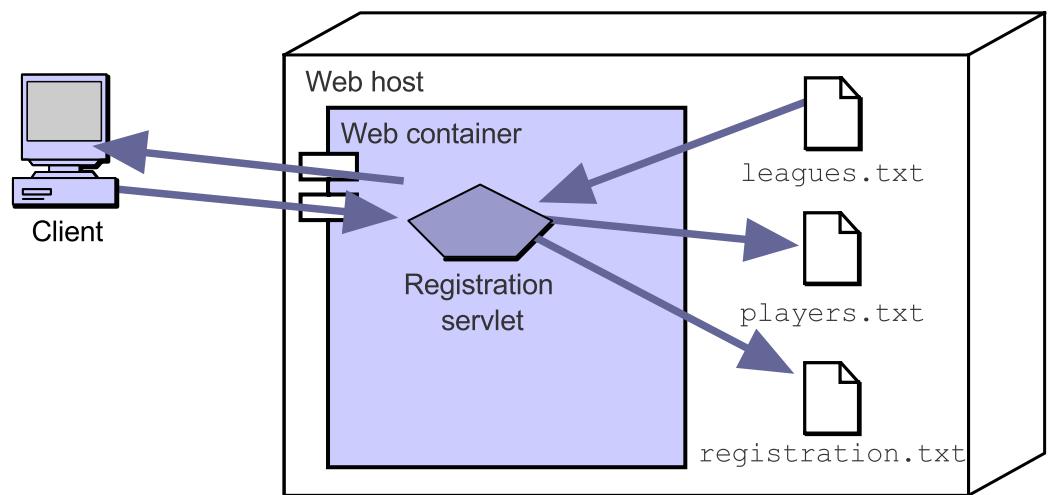
**Select Division:** Field for "Division" (dropdown menu set to "Amateur").

A "Register" button is located at the bottom left of the form area.

Figure 7-3     Soccer League Registration Form

## Architecture of the Soccer League Example

The registration servlet needs access to three flat files. The `leagues.txt` file stores the data for each league that has been created. The `players.txt` file stores the player name and address information. The `registration.txt` file stores the associations between players and the leagues that they register for (as well as the division). This Web application architecture is illustrated in Figure 7-4.



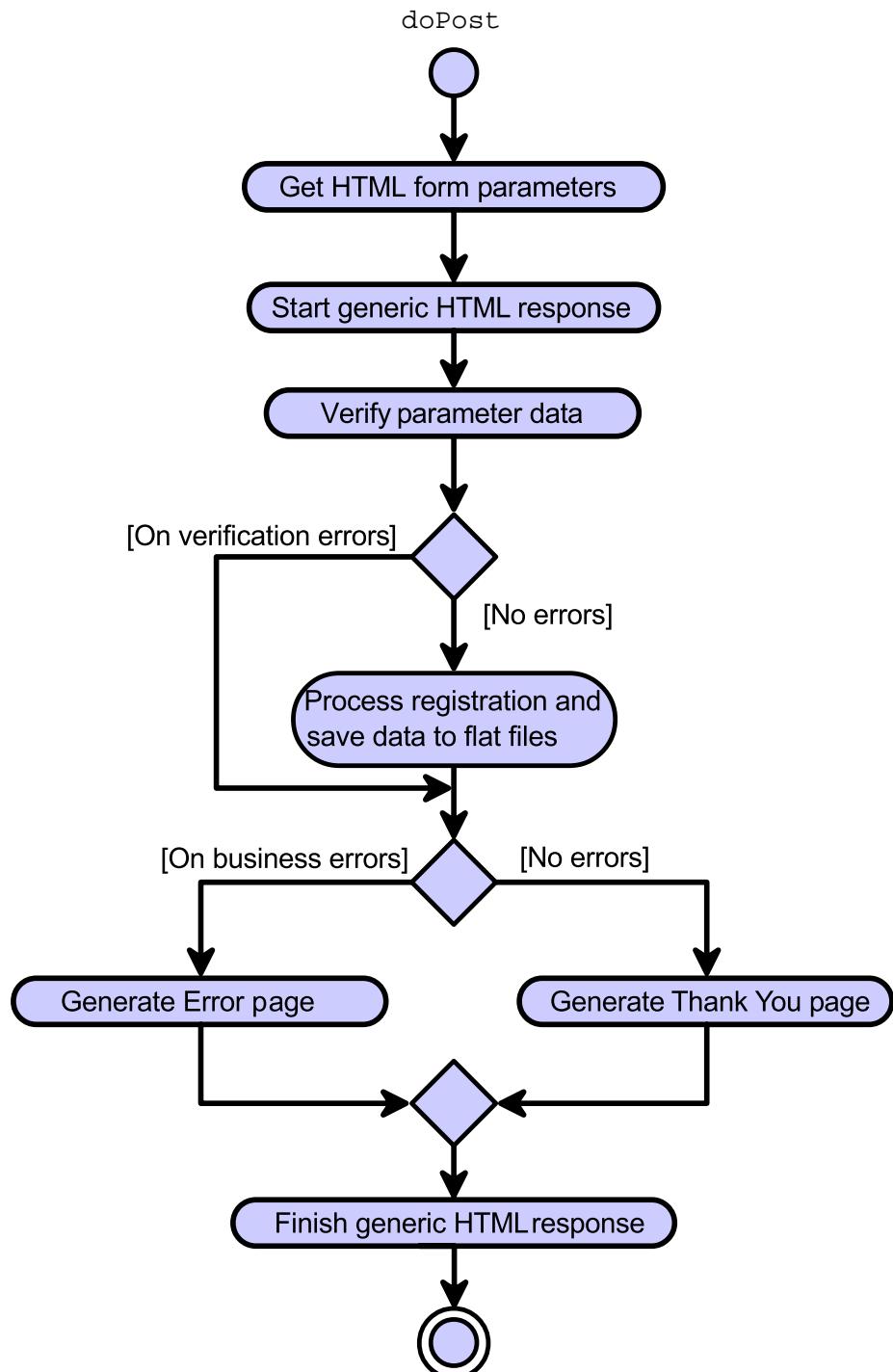
**Figure 7-4** Soccer League Deployment Diagram

## Activity Diagram of the Soccer League Example

The registration servlet must process each request by using the following steps:

- Retrieve and verify the HTML form parameters
- Perform the registration by saving data to the flat files
- Generate the appropriate response (either an Error page or a Thank You page)

One possible activity flow of the `doPost` method for this servlet is illustrated in Figure 7-5.



**Figure 7-5** Soccer League Registration Activity Diagram

## Discussion of the Simple Web Application



The Soccer League Web application design has several critical limitations. The most important limitation is that the `doPost` method is doing all of the work.

- This is poor modularization of the activity. How would you redesign the registration servlet to provide better modularity?
- How would you add new Use Cases to this Web application? Suppose you wanted to add an administrative Use Case to create new leagues for players? How would you code that servlet?
- In this simple example, flat files are the persistent storage mechanism. What would happen with the current version of this servlet if the project moved to a relational database?

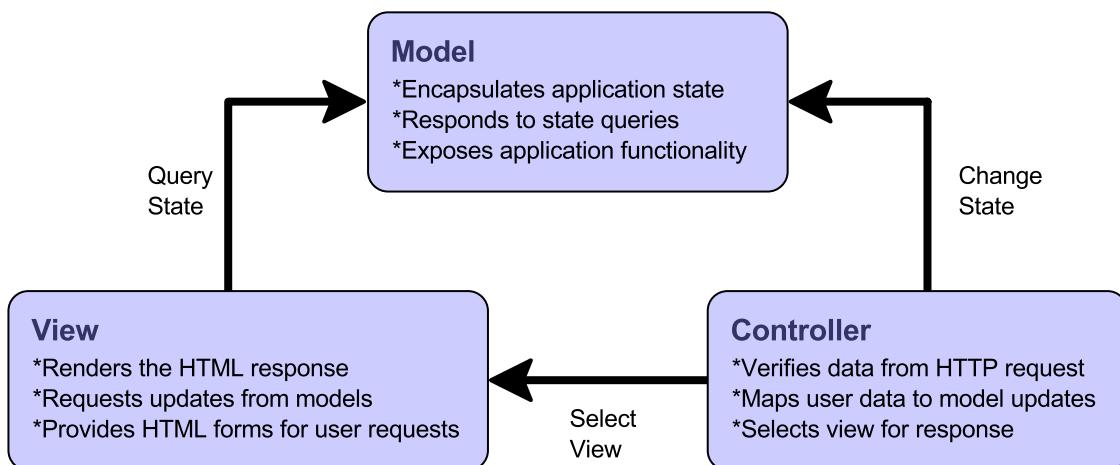
In the next section, the registration servlet is redesigned using an MVC-like pattern.

# Model-View-Controller for a Web Application

The Model-View-Controller pattern has its roots in GUI development in the Smalltalk environment, but this pattern is applicable in many languages and many situations. The main purpose of the pattern is to separate three kinds of responsibilities or tasks of an application. This is called “separation of concerns.” The three responsibilities are:

- Model – The business services and domain objects of the application
- View – The “window” into the application that the computer presents to the user
- Controller – The logic that accepts user actions, performs some business operation, and selects the next View for the user

The relationships among these application elements is illustrated in Figure 7-6.



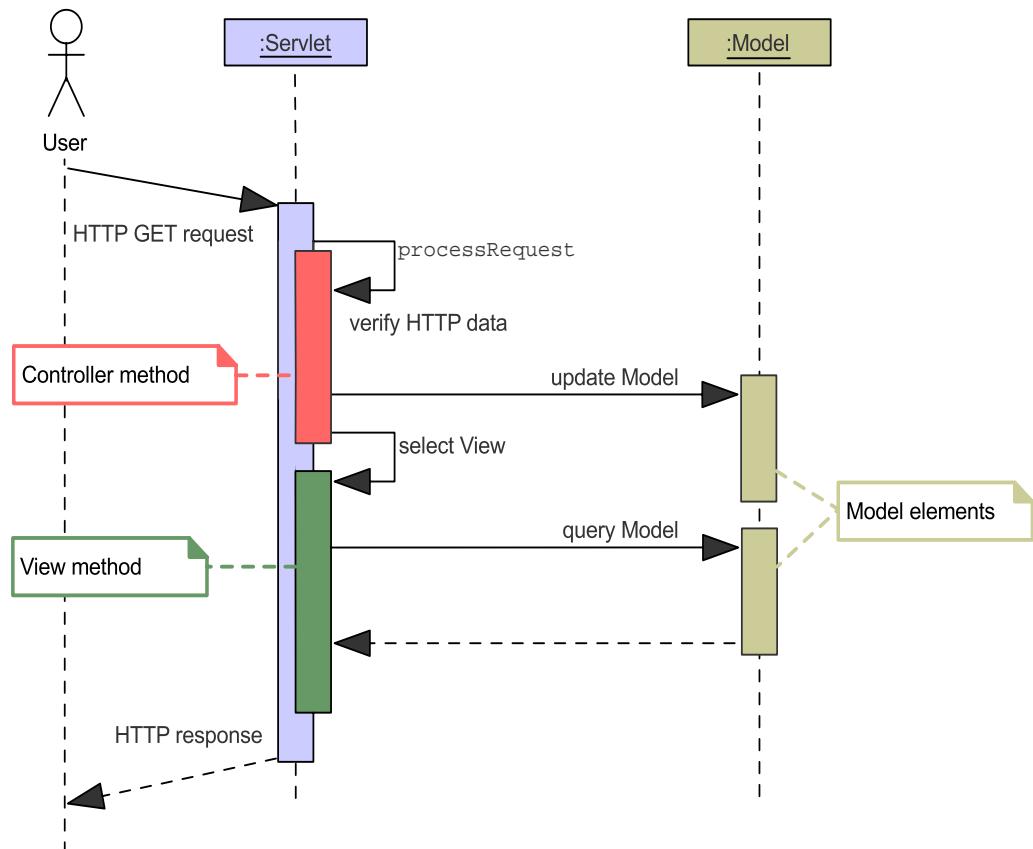
**Figure 7-6**      Model-View-Controller on the Web Tier



**Note** – In an actual MVC implementation, the Views would be implemented as JSP pages. You will learn about JSP pages later in this course. The MVC design pattern will be revisited in Module 15, “Developing Web Applications Using the Model 2 Architecture.”

## Sequence Diagram of MVC in the Web Tier

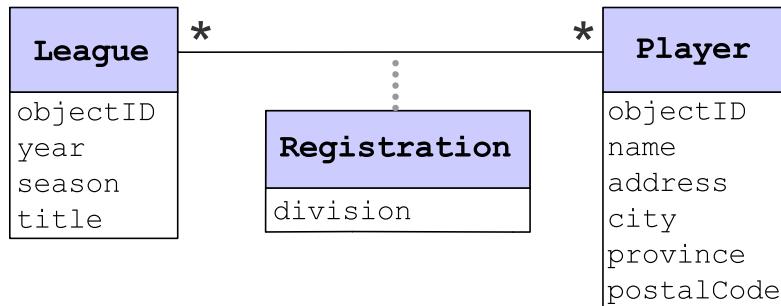
In this module, the Model elements are implemented as separate Java technology classes. The Controller and View elements are implemented as separate methods in a single servlet class. The interaction between the Web user, the servlet, and the Model is illustrated with a sequence diagram in Figure 7-7.



**Figure 7-7** Sequence Diagram of MVC on the Web Tier

## Soccer League Application: The Domain Model

The Soccer League application requires two independent domain objects: League and Player. The application also includes a dependent object: Registration. These objects hold the data relevant to these business entities. The relationship between the Soccer League domain objects is illustrated in Figure 7-8.



**Figure 7-8** The Soccer League Domain Model

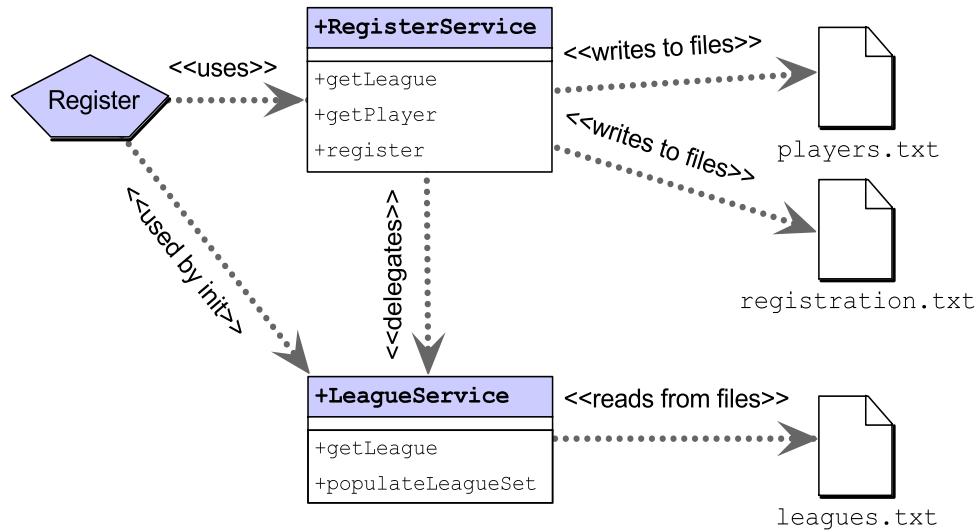
Any number of leagues and players can exist in the system. A league contains a collection of players and a player may register for more than one league. When a player registers for a league, that player must specify the division in which they want to play.

## Soccer League Application: The Services Model

The domain objects do not perform any business operations. In this design, service classes were created to perform these behaviors. In the registration example, the RegisterService class needs to support the following capabilities:

- Look up a league object based on the year and season entered by the user in the registration form (the `getLeague` method)
- Create or retrieve a player object (the `getPlayer` method)
- Register the player for the league (the `register` method)

Because you might need to look up league objects in many different Use Cases in the Soccer League Web application, a separate LeagueService class encapsulates that behavior. The RegisterService class uses a league service object as a delegate in the call to the getLeague method. The registration servlet uses the RegisterService class to perform the business logic for the Registration Use Case. The relationship between these components is illustrated in Figure 7-9.



**Figure 7-9** The Soccer League Services Model

## Soccer League Application: The Big Picture

The RegistrationServlet class has a processRequest method which acts as the Controller. This method retrieves the form data from the registration HTML form page and verifies that all of the data has been entered. The processRequest method uses the RegisterService object to perform the various business operations. Based on the success of these operations the processRequest method calls either the generateErrorResponse or the generateThankYouResponse method. These methods act as the Views of the application. These relationships are illustrated in Figure 7-10.

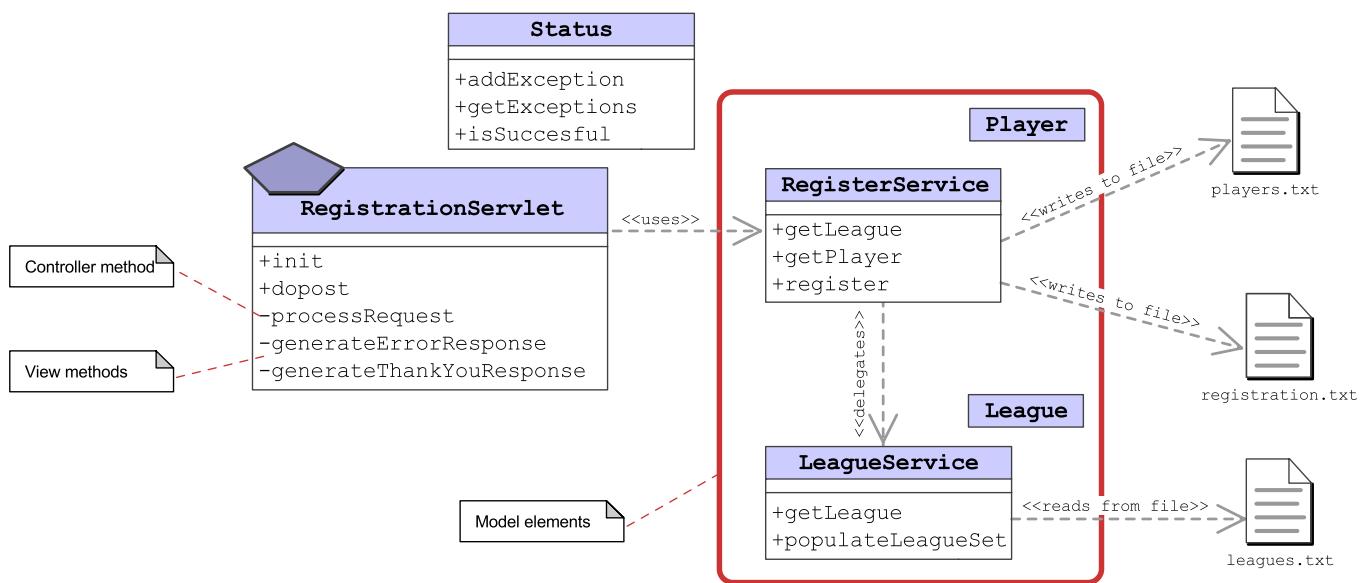


Figure 7-10 MVC Relationships in the Registration Use Case

More than one operation can fail, and the Status object is designed to hold all of these exceptions. For example, if the user fails to enter his name, or if the league that he selected does not exist, or if the register method fails (cannot write to the file, for example), then an Exception object is stored in the Status object. This class is a utility class that does not belong in the domain model. It is used in the Controller aspects of the application.

## Soccer League Application: The Controller

The processRequest method performs the Controller functions. It retrieves and verifies the form data. It performs business operations using the Model services. Based on the results of these operations, the processRequest method calls a View method to generate the next HTML response.

For example, if an error occurs during data verification (including looking up the league), then an Error page must be generated. This is shown in Code 7-1.

**Code 7-1** Example Controller Code

```
77     League league = regService.getLeague(year, season);
78     if ( league == null ) {
79         status.addException(
80             new Exception("The league you selected does not yet exist;"
81                         + " please select another."));
82     }
83
84     // If any of the above verification failed, then return the
85     // 'Error Page' View and return without proceeding with the
86     // rest of the business logic
87     if ( ! status.isSuccessful() ) {
88         generateErrorResponse(request, response);
89         return;
90     }
```

The Thank You View displays the player's name and the title of the league that he registered for. The Controller method (processRequest) passes these objects to the View methods by storing those objects as attributes in the HttpServletRequest object. An attribute is stored using the setAttribute method. This is shown in Code 7-2.

**Code 7-2** Controller Sends Data to the View

```
101    regService.register(league, player, division);
102    request.setAttribute("league", league);
103    request.setAttribute("player", player);
104
105    // The registration process was successful,
106    // Generate the 'Thank You' View
107    generateThankYouResponse(request, response);
```



---

**Note –** Passing objects using request attributes might not seem like the most efficient mechanism, but this is the best practice. This will be explained in more detail in Module 15, “Developing Web Applications Using the Model 2 Architecture.”

---

## Soccer League Application: The Views

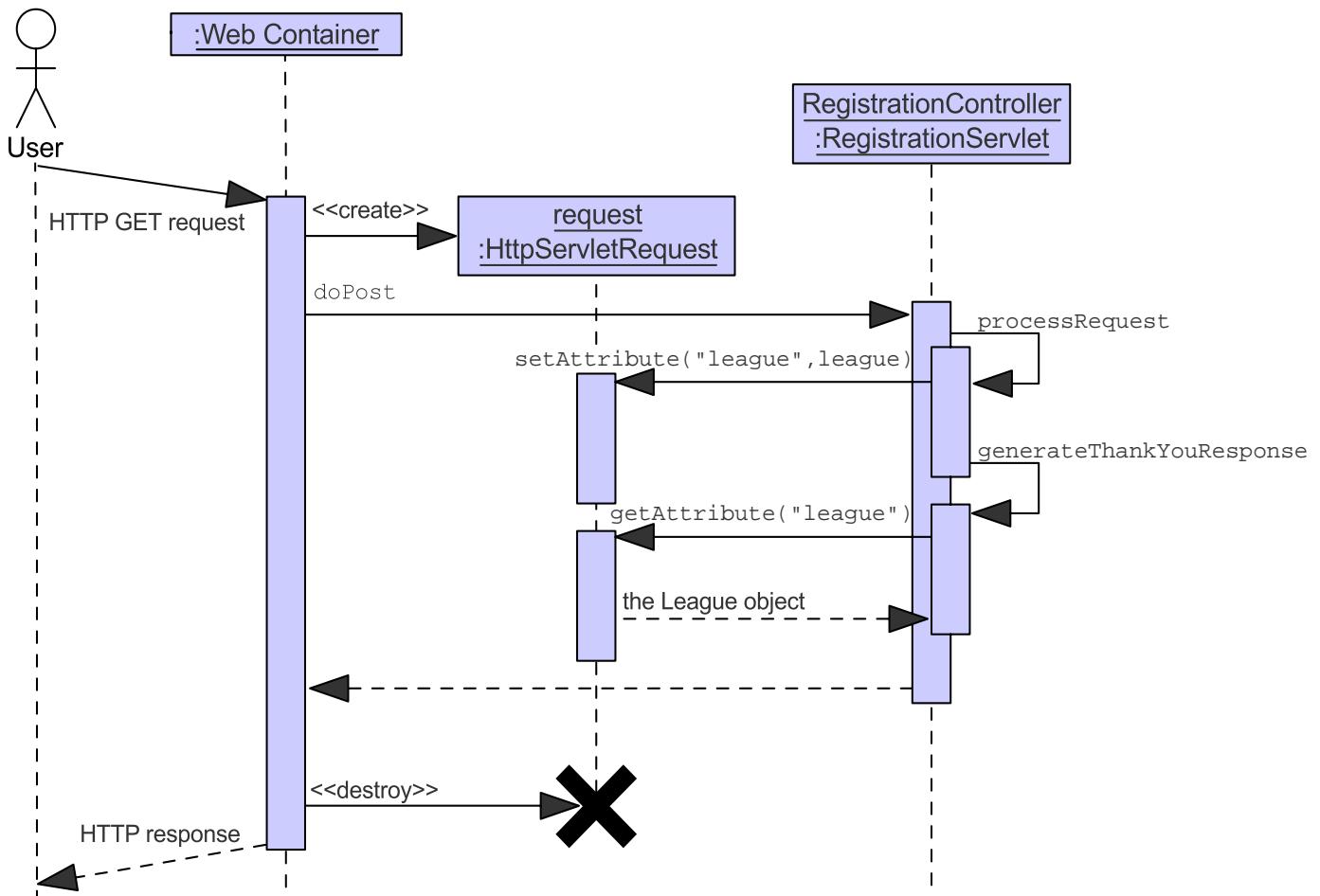
The generateXyzResponse methods act as the dynamic Views of the Web application. These methods are responsible for generating a dynamic HTML response. For example, the generateThankYouResponse method displays a simple “Thank you, <player name>, for registering for the <league title> league” message. The player and league information is passed to the View method using the request object. An attribute is retrieved using the getAttribute method. This is shown in Code 7-3.

**Code 7-3**      The View Retrieves Attributes

```
120 public void generateThankYouResponse(HttpServletRequest request,
121                                     HttpServletResponse response)
122     throws IOException {
123
124     // Specify the content type is HTML
125     response.setContentType("text/html");
126     PrintWriter out = response.getWriter();
127
128     League league = (League) request.getAttribute("league");
129     Player player = (Player) request.getAttribute("player");
```

## The Request Scope

To review, the Controller method passes data to a View method using attributes on the request object. This is called the *request scope*. Formally, the request scope is period of time during of processing a single HTTP request. Attributes stored in the request scope exist only for the duration of the request and response cycle because the request object is destroyed after the service method returns. This is illustrated with a sequence diagram in Figure 7-11.



**Figure 7-11** Sequence Diagram of Using the Request Scope

# Summary

This module presented Web application design issues. In particular, this module described and demonstrates how to develop a Web application using a modified version of the Model-View-Controller pattern. The main purpose of using this pattern in the Web tier is to cleanly separate “roles and responsibilities.” The elements of the MVC pattern are as follows:

- The Model represents the business services and domain objects.
- The Views of the Web application are a rendering of the user’s state within the application. The View retrieves domain objects using the `getAttribute` method.
- The Controller takes user actions (specifically, HTTP requests) and processes the events by manipulating the Model and selecting the appropriate next View. The Controller passes domain objects to the View using the `setAttribute` method on the request object.

Passing data using the request object is called using the request scope.

## Certification Exam Notes

This module presented several of the objectives for Section 13, "Web Tier Design Patterns," of the Sun Certification Web Component Developer certification exam:

- 13.1 Given a scenario description with a list of issues, select the design pattern (*Value Objects*, *MVC*, Data Access Object or Business Delegate) that would best solve those issues.
- 13.2 Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: *Value Objects*, *MVC*, Data Access Object, Business Delegate.

This module only gives you a hint about the MVC pattern. The full explanation is presented in Module 15, "Developing Web Applications Using the Model 2 Architecture." The Value Object pattern was not mentioned explicitly in this module. What was called simply a "domain object" in this module is equivalent to a Value Object. Read the J2EE Blueprints (<http://java.sun.com/j2ee/blueprints/>) for more information about the MVC and Value Object patterns.

The Data Access Object pattern is presented in Module 12, "Integrating Web Applications With Databases," and the Business Delegate pattern is presented in Module 20, "Integrating Web Applications With Enterprise JavaBeans Components."

Also presented in this module is:

- 1.4 Identify the interface and method to access values and resources and to set object attributes within the following three Web scopes: *request*, *session*, *context*.

# Developing Web Applications Using Session Management

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the purpose of session management
- Design a Web application that uses session management
- Develop servlets using session management
- Describe the cookies implementation of session management
- Describe the URL-rewriting implementation of session management

## Relevance



**Discussion** – The following questions are relevant to understanding what session management is all about:

- What mechanism do you currently use for maintaining communications across requests?
  
- How much additional development is needed to use that communication mechanism?

## Additional Resources



**Additional resources** – The following reference provides additional information on the topics described in this module:

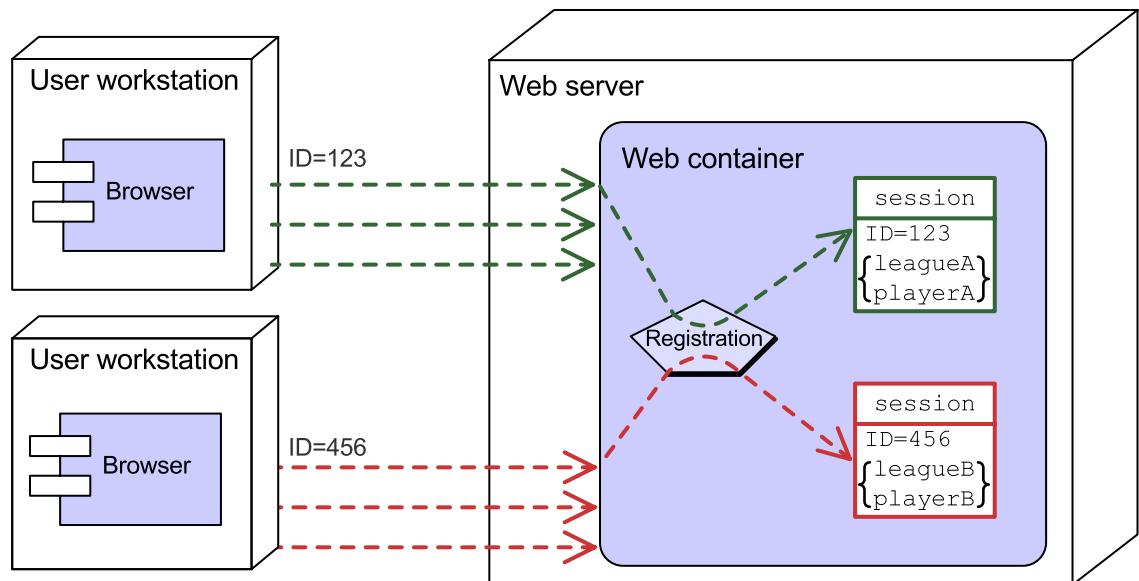
- HTTP State Management Mechanism. [Online]. Available at: <http://www.ietf.org/rfc/rfc2109.txt>

# HTTP and Session Management

HTTP is a stateless protocol. Each request and response message connection is independent of any others. This is significant because from one request to another (from the same user) the HTTP server forgets the previous request. Therefore, the Web container must create a mechanism to store session information for a particular user.

## Sessions in a Web Container

The Servlet specification mandates that the Web container must support session objects, which store attributes that are unique to a specific client, but exist across multiple HTTP requests. This is called the *session scope*. This is illustrated in Figure 8-1.



**Figure 8-1** The Web Container Manages Session Attributes

Each client is given a unique session ID that is used by the Web container to identify the session object for that user.

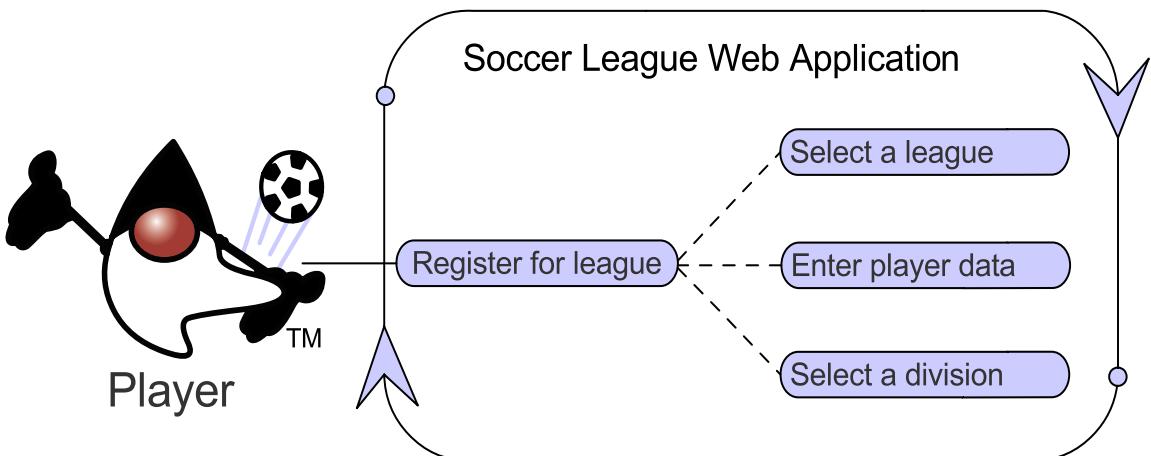
# Web Application Design Using Session Management

There are many ways to design a Web application using session management. This module presents a technique with three main design steps:

1. Design multiple, interacting views for a Use Case.
2. Create one servlet Controller for every Use Case in your Web application.
3. Use a hidden HTML parameter to indicate which action is being performed at a given stage in the session.

## Example: Registration Use Case

In the Soccer League example there is only one Use Case: player registration. This Use Case can be broken down into three activities: select a league, enter player information, and select a division. This is illustrated in Figure 8-2.

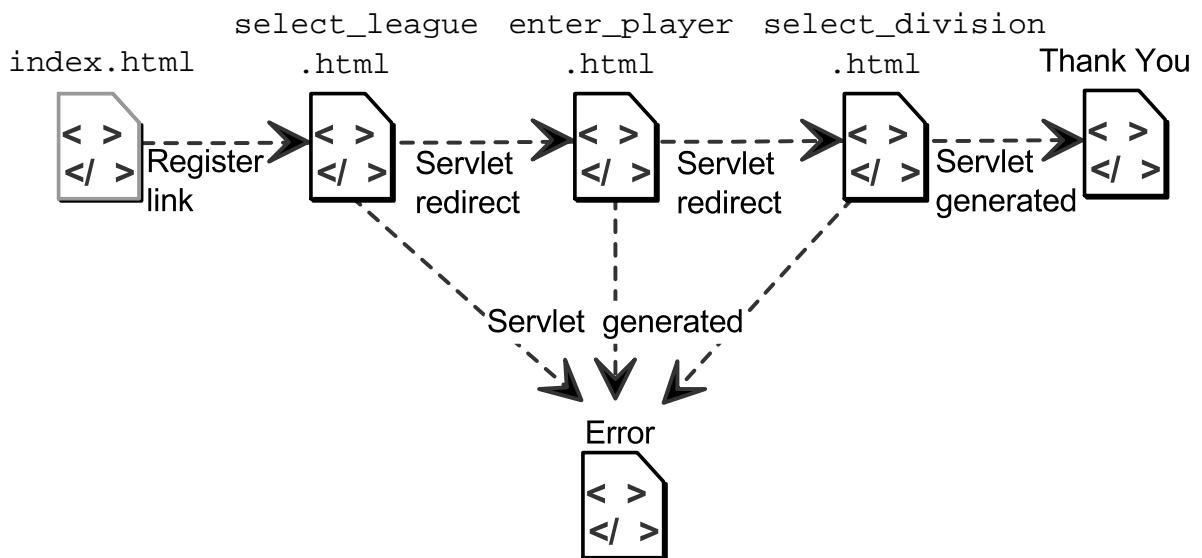


**Figure 8-2** Registration Use Case Diagram

This Use Case will be implemented by a single servlet Controller.

## Example: Multiple Views for Registration

Three HTML forms are used to collect the data for the three activities in the Registration Use Case. This is illustrated in Figure 8-3.



**Figure 8-3** Page Flow Diagram for the Registration Use Case

The flow of the Registration Use Case follows these steps:

1. The user starts at the main page (index.html) and selects the “Register” link. This returns the select\_league.html page.
2. When the user submits the “select league” form and the form is successfully processed by the servlet, the league object is saved in the session object and the enter\_player.html page is returned.
3. When the user submits the “enter player” form and the form is successfully processed, the player object is saved in the session object and the select\_division.html page is returned.
4. When the user submits the “select division” form and the form is successfully processed, the player has been registered and the Thank You page is generated.
5. If the servlet is unsuccessful at processing any of these forms, then an Error page is generated and the user must “back up” to edit and resubmit the form.

## Example: Enter League Form

Because you are using only one servlet Controller for this Use Case, that servlet must know which activity to process for each HTTP request. To determine which activity to process, you include a hidden HTML input tag named action with a value that indicates which activity must be processed for this form.

For example, the select\_league.html form has a hidden action field that specifies the “SelectLeague” activity. This is shown in Code 8-1.

**Code 8-1** Enter League HTML Form With Action Field

```
23 <FORM ACTION="register" METHOD="POST">
24
25 <INPUT TYPE="hidden" NAME="action" VALUE="SelectLeague">
```

The doPost method of the RegistrationServlet class dispatches the HTTP request to the activity-specific Controller methods based on the value of the hidden action field. This is shown in Code 8-2.

**Code 8-2** RegistrationServlet Controller Code

```
18 public class RegistrationServlet extends HttpServlet {
19
20     public static final String ACTION_SELECT_LEAGUE      = "SelectLeague";
21     public static final String ACTION_ENTER_PLAYER       = "EnterPlayer";
22     public static final String ACTION_SELECT_DIVISION   = "SelectDivision";
23
24     public void doPost(HttpServletRequest request,
25                         HttpServletResponse response)
26             throws IOException {
27
28         String action = request.getParameter("action");
29
30         if ( action.equals(ACTION_SELECT_LEAGUE) ) {
31             processSelectLeague(request, response);
32
33         } else if ( action.equals(ACTION_ENTER_PLAYER) ) {
34             processEnterPlayer(request, response);
35
36         } else if ( action.equals(ACTION_SELECT_DIVISION) ) {
37             processSelectDivision(request, response);
38         }
39     }
```

## Notes

# Web Application Development Using Session Management

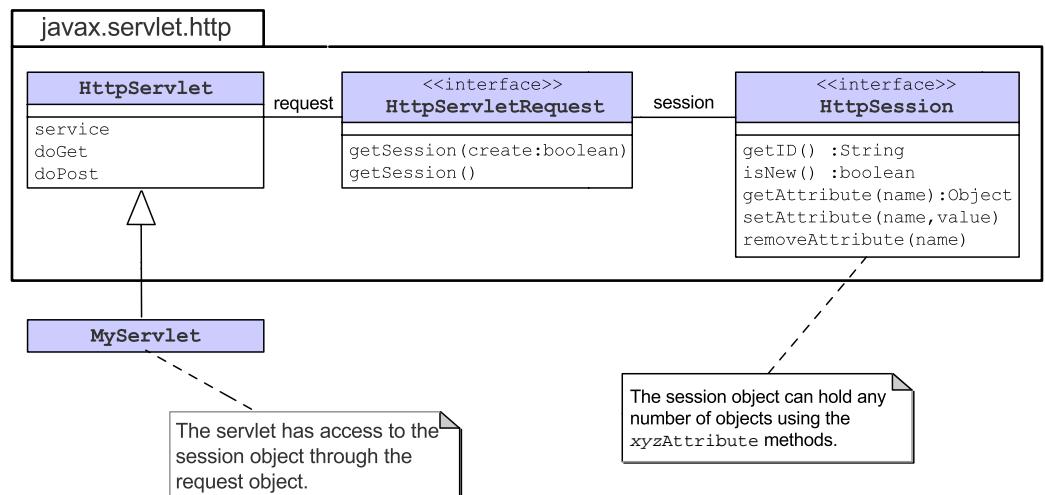
Each activity-specific control method must store attributes (name/object pairs) that are used by other requests within the session. For example, the `processSelectLeague` method must store the league object for later use. The Servlet specification provides a way to store attributes in the session scope.

Any control method can access an attribute that has already been set by processing a previous request. For example, the `processSelectDivision` method must access the league and player objects to complete the registration process.

At the end of the session, the servlet *might* destroy the session object. This is a controversial topic. It will be discussed in more detail later in this section.

## The Session API

The Servlet specification provides an `HttpSession` interface that allows you to store session attributes. You can store, retrieve, and remove attributes from the session object. The servlet has access to the session object through the `getSession` method of the `HttpServletRequest` object. This API is represented in Figure 8-4.



**Figure 8-4** The Session API

## Retrieving the Session Object

The session object is retrieved from the request object that is passed to the servlet by the Web container. This is shown in Code 8-3.

**Code 8-3**      Retrieving the Session Object

```
44 public void processSelectLeague(HttpServletRequest request,
45                               HttpServletResponse response)
46     throws IOException {
47     // Create the HttpSession object
48     HttpSession session = request.getSession();
```

The getSession method returns the current session associated with this request, or if the request does not have a session, the getSession method creates one. You can test whether the session object has just been created using the isNew method. If the session object already exists, then every call to the getSession method will return the same object.



**Note** – Only one session object will be created for a given client within a single Web application. However, several Use Cases in the same Web application may share the session object. This has design implications that will be discussed later in this module.

---

## Storing Session Attributes

When you have the session object, you can use the `setAttribute` method to store data in the session scope. For example, the `processSelectLeague` control method stores the league object that was specified by the form data selected by the user. This is shown in Code 8-4.

**Code 8-4**      The `processSelectLeague` Control Method

```
44  public void processSelectLeague(HttpServletRequest request,
45          HttpServletResponse response)
46          throws IOException {
47      // Create the HttpSession object
48      HttpSession session = request.getSession();
49
50      // Create business logic objects
51      RegisterService registerSvc = new RegisterService();
52
53      // Create the status object and store it in the request for use
54      // by the 'Error Page' View (if necessary)
55      Status status = new Status();
56      request.setAttribute("status", status);
57
58      // Extract HTML form parameters
59      String season = request.getParameter("season");
60      String year = request.getParameter("year");
61
62      // Verify 'Select League' form fields
63      if ( season.equals("UNKNOWN") ) {
64          status.addException(new Exception("Please select a league
65          season."));
65      }
66      if ( year.equals("UNKNOWN") ) {
67          status.addException(new Exception("Please select a league
68          year."));
68      }
69
70      // Retrieve the league object
71      League league = registerSvc.getLeague(year, season);
72      if ( league == null ) {
73          status.addException(
74              new Exception("The league you selected does not yet exist;"
75                          + " please select another."));
75      }
76  }
```

```
78 // Generate the "Error Page" response and halt
79 if ( ! status.isSuccessful() ) {
80     generateErrorResponse(request, response);
81     return;
82 }
83
84 // Store the league object in the session
85 session.setAttribute("league", league);
86
87 // Select the next tView: "Enter Player" form
88 response.sendRedirect("enter_player.html");
89 }
```

Line 71 uses the registration service object to retrieve the league object that was specified by the form data from the user's request. If there were no errors, then the league object is stored in the session object on Line 85.



**Note** – Line 88 uses an `HttpServletResponse` method that you have not seen yet. The `sendRedirect` method tells the Web container to redirect the HTTP request to a different HTML page. Because the "Enter Player" View is a static HTML page, you can use the `sendRedirect` method rather than generating the static content.

---

## Naming Session Attributes

As noted earlier, session objects are shared across multiple Use Cases in a Web application. Attribute names should be considered carefully. In this example, the league object used in the Registration Use Case is simply named "league." Another Use Case in the Soccer League Web application might exist that uses a league object for a completely different purpose. If that Use Case servlet uses the same name, "league," then these two servlets might interfere with each other if the same user happens to be dealing with both Use Cases within the same Web session. It is often better to give a session attribute a more explicit name; for example, "registration.league" and "registration.player."

## Accessing Session Attributes

You can use the `getAttribute` method to retrieve data in the session object. For example, the `processSelectDivision` control method retrieves the league and player objects (Lines 156 and 157) to complete the registration process by calling the `register` method on the register service object (Line 184). This is shown in Code 8-5.

**Code 8-5**      The `processSelectDivision` Control Method

```
149  public void processSelectDivision(HttpServletRequest request,
150                  HttpServletResponse response)
151          throws IOException {
152      // Retrieve the HttpSession object
153      HttpSession session = request.getSession();
154
155      // Retrieve the domain object from the session
156      League league = (League) session.getAttribute("league");
157      Player player = (Player) session.getAttribute("player");
158
159      // Create business logic objects
160      RegisterService registerSvc = new RegisterService();
161
162      // Create the status object and store it in the request for use
163      // by the 'Error Page' View (if necessary)
164      Status status = new Status();
165      request.setAttribute("status", status);
166
167      // Extract HTML form parameters
168      String division = request.getParameter("division");
169
170      // Verify 'Select Division' form fields
171      if ( division.equals("UNKNOWN") ) {
172          status.addException(new Exception("Please select a division."));
173      }
174
175      // If there were verification problems,
176      // then generate the "Error Page" response and halt
177      if ( ! status.isSuccessful() ) {
178          generateErrorResponse(request, response);
179          return;
180      }
181  }
```

```
182 // Now delegate the real work to the RegisterService object
183 try {
184     registerSvc.register(league, player, division);
185
186     // Catch any error and send it to the 'Error Page' View
187 } catch (Exception e) {
188     status.addException(e);
189     generateErrorResponse(request, response);
190     return;
191 }
192
193 // The registration process was successful,
194 // generate the 'Thank You' View
195 generateThankYouResponse(request, response);
196 // This user's session is complete.
197 session.invalidate();
198 }
```

The Views of the Web application might also need access to the session attributes (Lines 213 and 214). For example, the “Thank You” View method uses the title of the league object and the name of the player object in the response (Lines 231 and 232). This is shown in Code 8-6.

**Code 8-6**      The generateThankYouResponse View Method

```
203 public void generateThankYouResponse(HttpServletRequest request,
204                                     HttpServletResponse response)
205     throws IOException {
206     // Retrieve the HttpSession object
207     HttpSession session = request.getSession();
208
209     // Specify the content type is HTML
210     response.setContentType("text/html");
211     PrintWriter out = response.getWriter();
212
213     League league = (League) session.getAttribute("league");
214     Player player = (Player) session.getAttribute("player");
215
216     // Generate the HTML response
217     out.println("<HTML>");
218     out.println("<HEAD>");
219     out.println("<TITLE>Registration: Thank You</TITLE>");
220     out.println("</HEAD>");
221     out.println("<BODY BGCOLOR='white'>");
222     out.println("<TABLE BORDER='0' WIDTH='600'>");
223     out.println("<TR>");
224     out.println("  <TD COLSPAN='2' BGCOLOR='#CCCCCC' ALIGN='center'>");
225     out.println("    <H3>Registration: Thank You</H3>");
226     out.println("  </TD>");
227     out.println("</TR>");
228     out.println("</TABLE>");
229     out.println();
230     out.println("<BR>");
231     out.println("Thank you, " + player.getName() + ", for registering"
232               "in the <B>" + league.getTitle() + "</B> league.");
233     out.println();
234     out.println("</BODY>");
235     out.println("</HTML>");
236 }
```

## Destroying the Session

When the Web application is completed with a session, the servlet can (effectively) destroy the session using the `invalidate` method (Line 197 in Code 8-5).

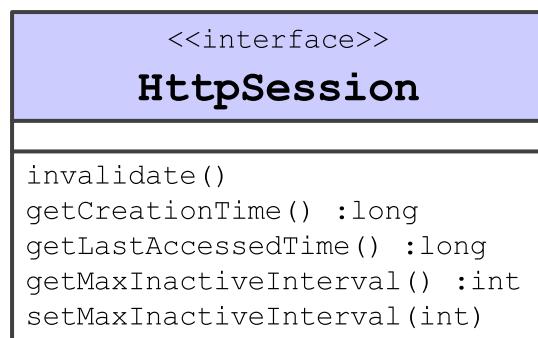
There are two other mechanisms for destroying a session; both are managed by the Web container. First, you can configure a time-out parameter in the deployment descriptor. The value of the `session-timeout` element must be a whole number that represents the number of minutes that a session can exist if the user has left the session inactive. The `session-timeout` element is located just under the root `web-app` element and just after the `servlet-mapping` elements. An example `session-timeout` element is shown in Code 8-7.

**Code 8-7** Session Time-out Configuration

```
35 <session-config>
36   <session-timeout>10</session-timeout>
37 </session-config>
```

The Web container keeps track of the last time the user interacted with the Web application; this is known as the “inactive interval.” If a given session has been inactive for longer than the time-out parameter, then the Web container has the authority to invalidate that session. The time-out parameter specified in the deployment descriptor applies to all sessions within that Web application.

The second mechanism allows you to control the length of the inactive interval for a specific session object. You can use the `setMaxInactiveInterval` method to change the inactive interval (in seconds) for the session object. This is illustrated in Figure 8-5.



**Figure 8-5** Other Session Methods

## Issues With Destroying the Session

Destroying a session is a design decision, and there are many implications that need to be considered. The main issue is that the Web application might have several independent Use Cases that share the same session object. If you invalidate the session, you destroy the session object for the other Use Cases as well. The servlets handling these other Use Cases might lose data stored in the destroyed session. It might be better to remove only the attributes used in the current Use Case. The servlet must be designed to check that its attributes are null before starting the Use Case.

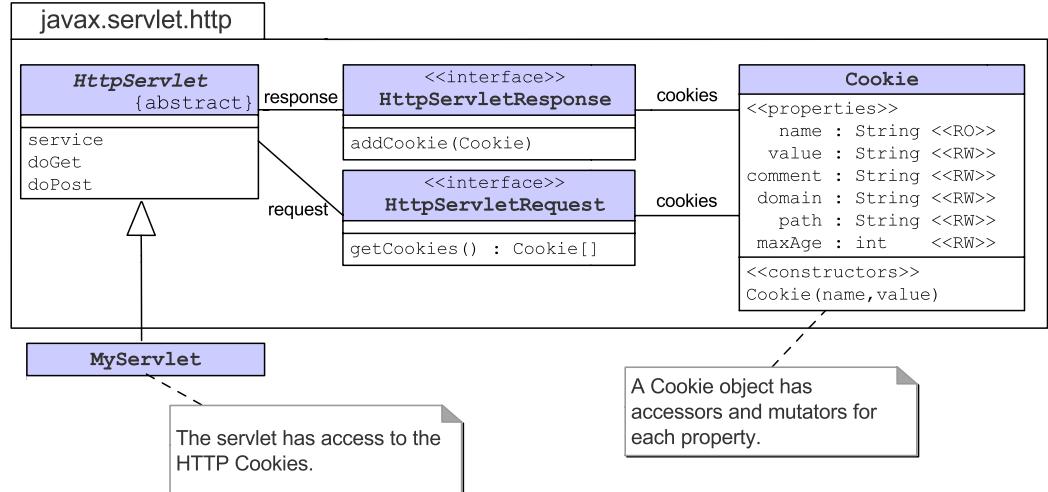
# Session Management Using Cookies

Internet Engineering Task Force (IETF) Request for Comment (RFC) #2109 creates an extension to HTTP that allows a Web server to store information on the client machine:

- Cookies are sent in a response from the Web server.
- Cookies are stored on the client's computer.
- Cookies are stored in a partition assigned to the Web server's domain name. Cookies can be further partitioned by a "path" within the domain.
- All Cookies for that domain (and path) are sent in every request to that Web server.
- Cookies have a lifespan and are flushed by the client browser at the end of that lifespan.

## The Cookie API

You can create Cookies using the `Cookie` class in the servlet API. You can add Cookies to the response object using the `addCookie` method. This sends the Cookie information over the HTTP response stream, and the information is stored on the user's computer (through the Web browser). You can access Cookies sent back from the user's computer in the HTTP request stream using the `getCookies` method. This method returns an array of all Cookies for the server domain on the client machine. This API is shown in Figure 8-6 on page 8-18.



**Figure 8-6** The Cookie API

## Using Cookies

Imagine that there is a visitor to your Web site and that you want to store her name so that the next time she visits the site you can personalize the screen. In your servlet, you could use the following code to store that Cookie:

```

String name = request.getParameter("firstName");
Cookie c = new Cookie("yourname", name);
response.addCookie(c);

```

Later when the visitor returns, your servlet can access the "yourname" Cookie using the following code:

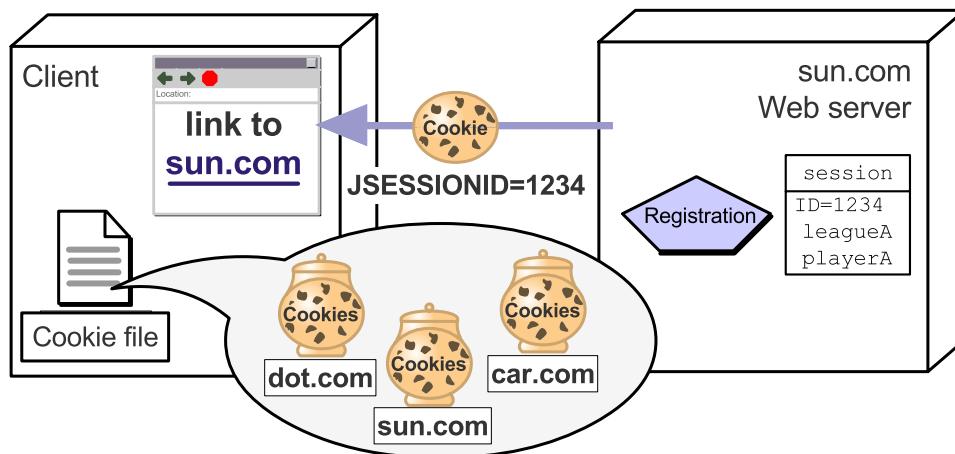
```

Cookie[] allCookies = request.getCookies();
for ( int i=0; i < allCookies.length; i++ ) {
    if ( allCookies[i].getName().equals("yourname") ) {
        name = allCookies[i].getValue();
    }
}

```

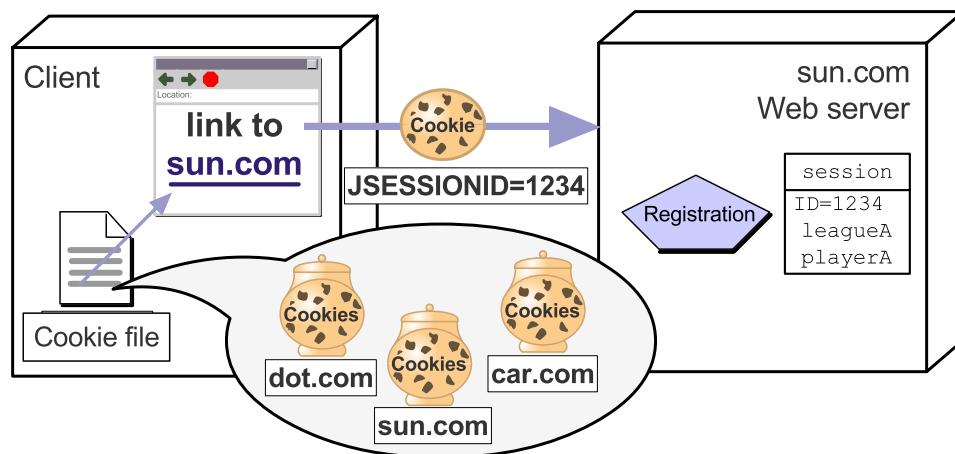
## Using Cookies for Session Management

HTTP Cookies can be used to perform session management. The Web container could store the session ID on the client machine. This is illustrated in Figure 8-7.



**Figure 8-7**      Web Container Stores the Session ID Cookie

While the session is still active, every HTTP request from the client includes the session ID Cookie that was stored on the client's machine. This is illustrated in Figure 8-8.



**Figure 8-8**      Web Container Retrieves the Session ID Cookie

## Session Management Using Cookies

---

When the `getSession` method is called, the Web container uses the session ID Cookie information to find the session object.



---

**Note** – The Servlet specification mandates that the name of the session ID Cookie be `JSESSIONID`.

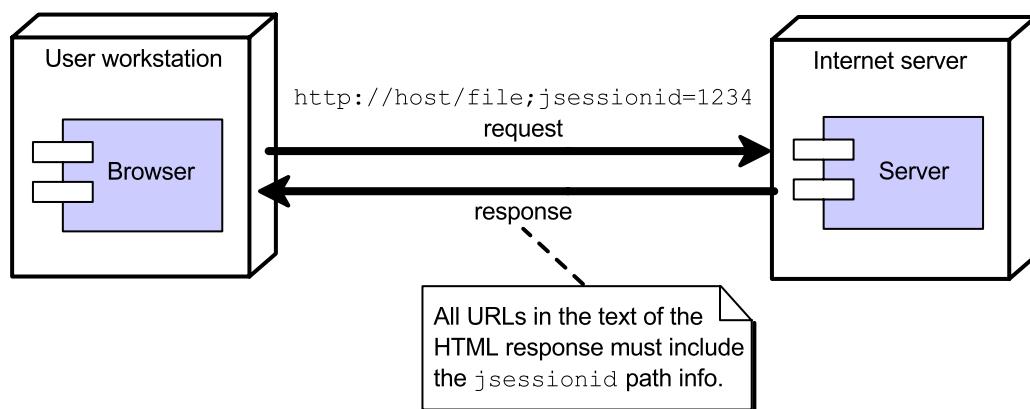
---

The Cookie mechanism is the default `HttpSession` strategy. You do not need to code anything special in your servlets to make use of this session strategy. Unfortunately, some browsers do not support Cookies or some users turn off Cookies on their browsers. If that happens, then the Cookie-based session management fails.

## Session Management Using URL-Rewriting

URL-rewriting is an alternative session management strategy that the Web container must support. Typically, a Web container attempts to use Cookies to store the session ID, but if that fails (that is, the user has Cookies turned off in the browser), then the Web container will try to use URL-rewriting.

URL-rewriting is implemented by attaching additional information (the session ID) onto the URL that is sent in the HTTP request stream. This is illustrated in Figure 8-9.



**Figure 8-9** Session Management Using URL-Rewriting

## Implications of Using URL-Rewriting

The URL-rewriting strategy is not as transparent as the Cookie strategy. Every HTML page and form that participates in a session *must* have the session ID information encoded into the URLs that are sent to the Web container. For example, during the Registration Use Case, the session is started when the user submits the “Select League” form. When the registration servlet requests the session object, the Web container issues a unique session ID. This ID must be added to the URL in the “Enter Player” page to the ACTION attribute in the FORM tag. This is shown in Code 8-8.

### Code 8-8      Encoding a URL

```
234 out.println("<FORM ACTION=' " + response.encodeURL("register") + "' METHOD='POST'>");  
235 out.println("");  
236 out.println("<INPUT TYPE='hidden' NAME='action' VALUE='EnterPlayer'>");
```

The process of URL-rewriting can be tedious. A static HTML page or form must now be dynamically generated to encode every URL (in FORM and A HREF tags). However, if you cannot verify that every user of your Web application will use Cookies, then you must consider that the Web container might need to use URL-rewriting.

## Guidelines for Working With Sessions

When dealing with session management, remember that:

- A servlet should create a session using the `getSession` method on the request object to save the state between HTTP requests.
- A servlet can determine if a session already exists by:
  - Checking for a null return use `getSession(false)`.
  - Calling `isNew` on the resulting session if you use `getSession(true)`.
- Any servlet can request that a session be created. A session is shared among all servlets in a Web application.
- Session attributes should be given appropriate names to avoid ambiguity.
- A session can be invalidated and become unusable.
- A session can time out due to browser inactivity.

## Summary

HTTP is a stateless protocol. The servlet API provides a session API. The following methods are useful for session management:

- Use either of the `getSession` methods on the request object to access (or create) the session object.
- Use the `setAttribute` method on the session object to store one or more name-object pairs.
- Use the `getAttribute` method on the session object to retrieve an attribute.
- Use the `invalidate` method on the session object to destroy a session. You can also use the Web container to destroy the session using a timeout declared in the deployment descriptor.

# Certification Exam Notes

This module presented most of the objectives for Section 5, "Designing and Developing Servlets Using Session Management," of the Sun Certification Web Component Developer certification exam:

- 5.1 Identify the interface and method for each of the following:
  - Retrieve a session object across multiple requests to the same or different servlets within the same Web application
  - Store objects into a session object
  - Retrieve objects from a session object
  - Respond to the event when a particular object is added to a session
  - Respond to the event when a session is created and destroyed
  - Expunge a session object
- 5.2 Given a scenario, state whether a session object will be invalidated.
- 5.3 Given that URL-rewriting must be used for session management, identify the design requirement on session-related HTML pages.

For objective 5.1, the following topics are not presented in this course:

- Respond to the event when a particular object is added to a session
- Respond to the event when a session is created and destroyed

Review the Servlet specification (v2.3) for more details on these topics.

Also presented in this module is:

- 1.3 For each of the following operations, identify the interface and method name that should be used:
  - Redirect an HTTP request to another URL
- 1.4 Identify the interface and method to access values and resources and to set object attributes within the following three Web scopes: *request*, *session*, *context*.



# Handling Errors in Web Applications

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the types of errors that can occur in a Web application
- Declare an HTTP error page using the Web application deployment descriptor
- Declare a Java technology exception error page using the Web application deployment descriptor
- Develop an error handling servlet
- Write servlet code to capture a Java technology exception and forward it to an error handling servlet
- Write servlet code to log exceptions

## Additional Resources



**Additional resources** – The following reference provides additional information on the topics described in this module:

- Hypertext Transfer Protocol – HTTP/1.1. [Online]. Available at: <http://www.ietf.org/rfc/rfc2616.txt>

# The Types of Web Application Errors

There are two types of errors in Web applications: HTTP errors and servlet exceptions.

## HTTP Error Codes

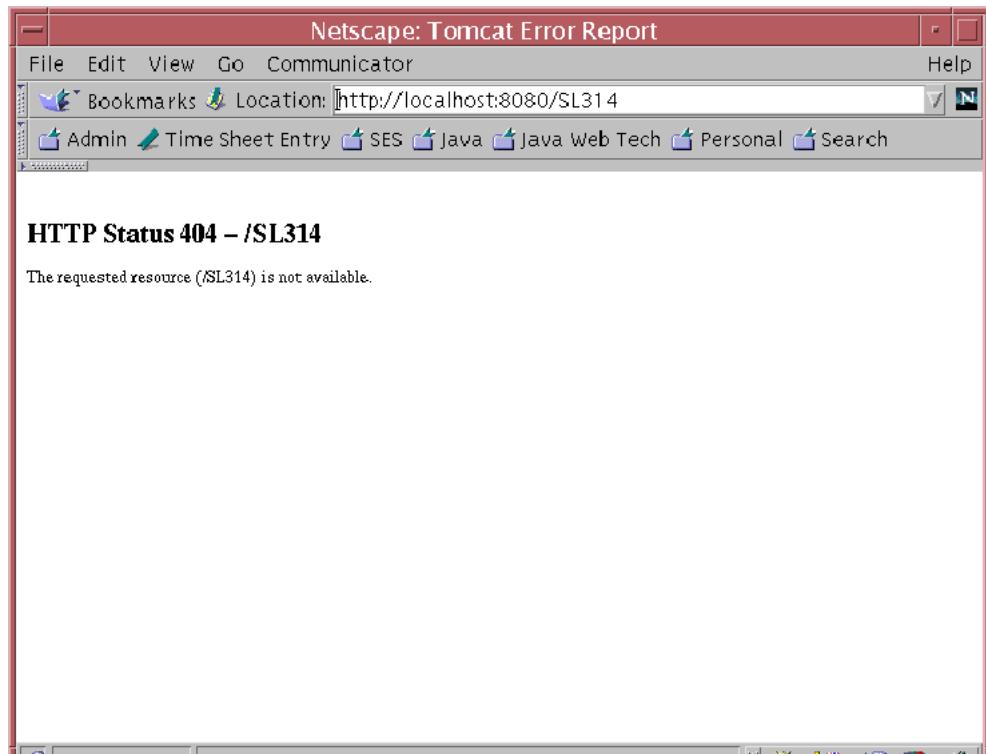
An HTTP response sent from the Web server to the client includes a status code, which tells the Web browser if the request was successful (the 200 status code) or unsuccessful. The status codes in the 400–500 range are used to indicate some error. Table 9-1 shows some examples.

**Table 9-1** HTTP Error Codes

400	Bad Request
401	Unauthorized
404	Not Found
405	Method Not Allowed
415	Unsupported Media Type
500	Internal Server Error
501	Not Implemented
503	Service Unavailable

## Generic HTTP Error Page

By default, the Web browser displays some message to the user. Often, this message is composed of HTML that is generated by the Web browser. This means that the Web server did not send any HTML message in the HTTP response. An example generic HTTP error page is shown in Figure 9-1.



**Figure 9-1** An Example Generic HTTP Error Page

## Servlet Exceptions

In addition to HTTP errors, a Java technology Web application can generate exceptions to indicate a problem with processing the HTTP request. A servlet can throw a `ServletException` to indicate to the Web container that an exception has occurred. An example of a servlet that throws an `ArithmaticException` is show in Code 9-1.

**Code 9-1**      A Servlet That Throws an Exception

```
12  public void doGet(HttpServletRequest request,
13      HttpServletResponse response)
14          throws ServletException {
15      int x = 0;
16      int y = 0;
17
18      try {
19          int z = x / y;
20      } catch (ArithmaticException ae) {
21          throw new ServletException(ae);
22      }
23 }
```



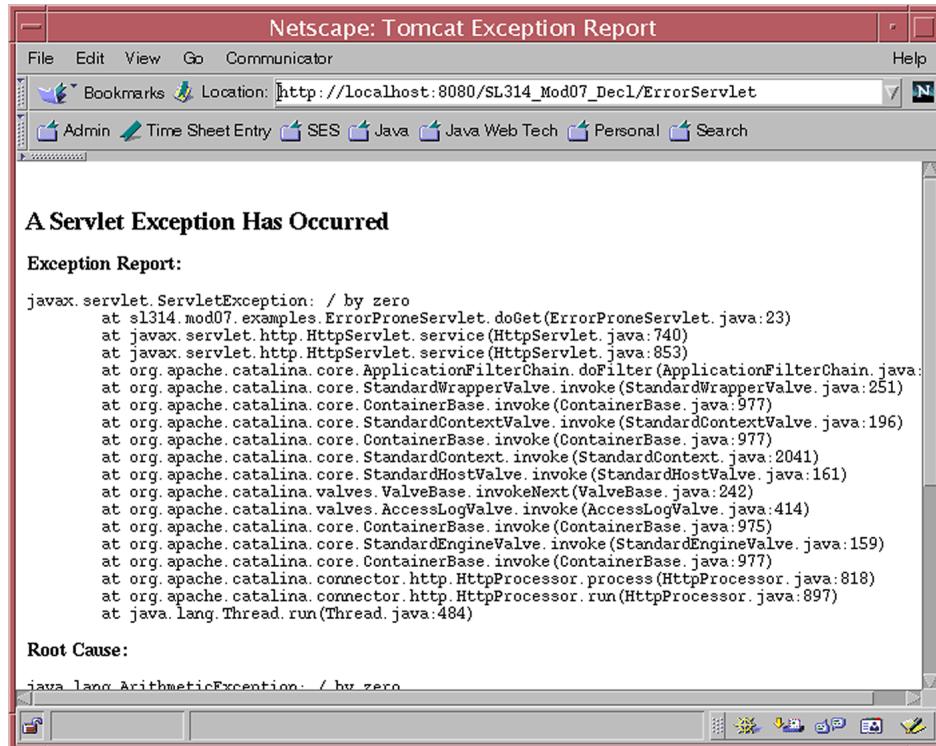
---

**Note** – The example shown in Code 9-1 is contrived and is used only for demonstration purposes. An `ArithmaticException` is a “non-checked exception” and need not be caught in a try-catch block. All non-check exceptions thrown by the service method are caught by the Web container, which issues a servlet exception on behalf of the servlet itself.

---

## Generic Servlet Error Page

The Web container will catch these exceptions and send an HTTP response with a 500 status code and an HTML response with the stack trace of the exception. An example generic servlet exception error page is shown in Figure 9-2.



**Figure 9-2** An Example Generic Servlet Exception Error Page

# Using Custom Error Pages

The generic error pages provided by the Web browser (for HTTP error codes) and the Web container (for servlet exceptions) are often ugly and not very informative to the end user. In this module, you will see how to create new error pages and how to activate these custom error pages. There are two ways to activate an error page within a Web application:

- Declarative – Use the deployment descriptor to declare error pages for specific situations (HTTP errors or Java technology exceptions), and let the Web container handle the forwarding to these pages.
- Programmatic – Handle the Java technology exceptions directly in your servlet code, and forward the HTTP request to the error page of your choice.

## Creating Error Pages

Error pages exist as static HTML or as dynamic servlets. Just like standard HTML pages, static error pages are located anywhere in the hierarchy of the Web application. Servlet-based error pages, however, can be given a URL mapping. An example of this is illustrated in Figure 9-3.

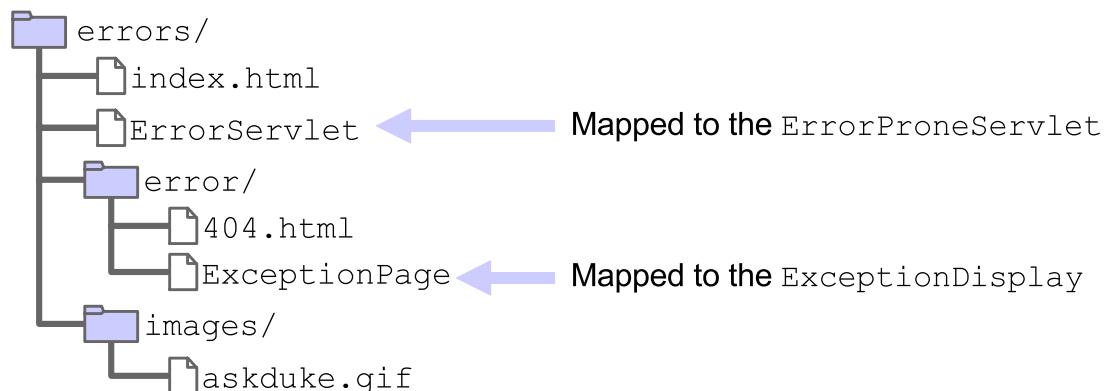


Figure 9-3     Error Pages in a Web Application

## Declaring HTTP Error Pages

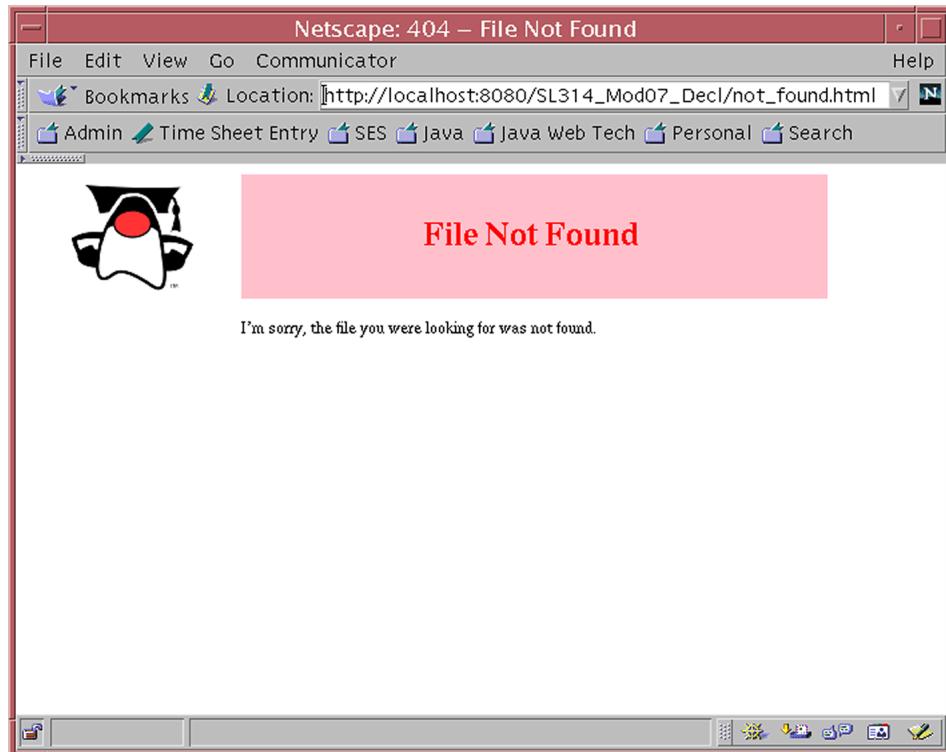
You can use the `error-page` element in the deployment descriptor to declare to the Web container that if an HTTP response is being sent back with a particular status code (for example, 404 “File Not Found”), then the HTML of the response is specified by the error page of your choice. The `error-page` element must include two subelements: `error-code`, the HTTP numeric status code, and `location`, the URL location of the custom error page. This is shown in Code 9-2.

**Code 9-2** An Example Custom HTTP Error Page Declaration

```
44 <servlet-name>ErrorProneService</servlet-name>
45 <url-pattern>/ErrorServlet</url-pattern>
46 </servlet-mapping>
47
```

## Example HTTP Error Page

Whenever a 404 error occurs in this Web application, the `/error/404.html` custom error page is sent in the response. The HTML in this page is rendered as shown in Figure 9-4.



**Figure 9-4** The Example Custom 404 Error Page

You can specify any number of error-page elements, but only one for a specific HTTP status code.

## Declaring Servlet Exception Error Pages

You can also use a deployment descriptor to handle servlet exceptions. Using the error-page element, you can instruct the Web container to forward specific exception types to the error page of your choice. To do this, use the exception-type subelement to identify the fully qualified exception class name. This is shown in Code 9-3.

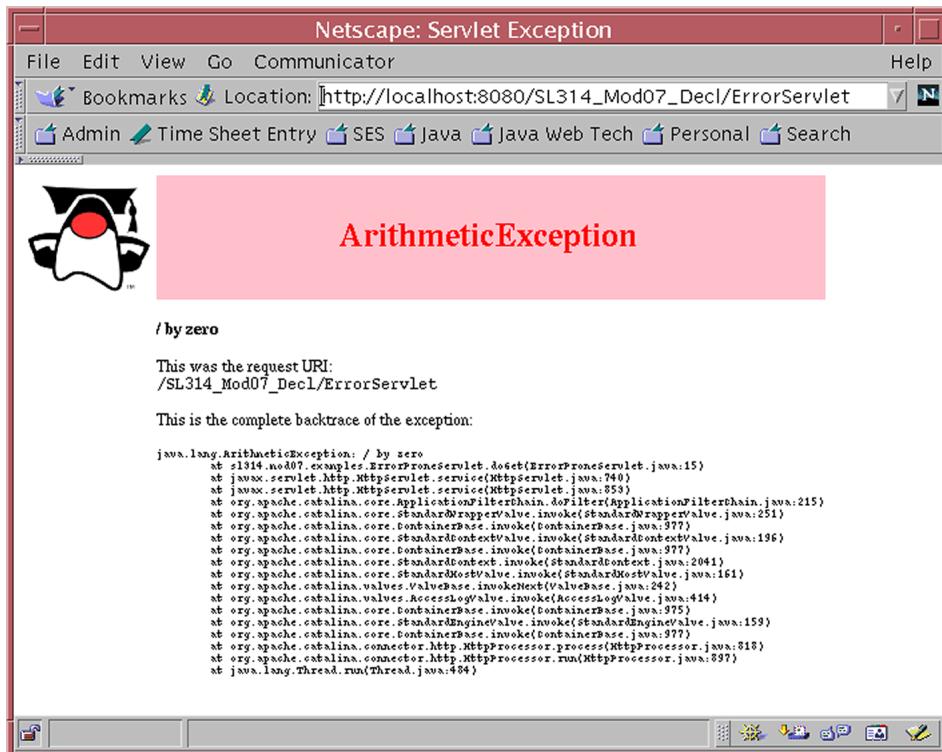
**Code 9-3** An Example Custom Servlet Exception Error Page Declaration

```
49 <servlet-name>NullPointerService</servlet-name>
50   <url-pattern>/NullPointerServlet</url-pattern>
51 </servlet-mapping>
52
```

Again, you can specify any number of error-page elements, but only one for a specific Java exception class. Also, you may use a superclass, like `java.lang.Exception`, to capture a range of exceptions.

## Example Servlet Error Page

The example servlet in A Servlet That Throws an Exception on page 9-5 throws an `ArithmaticException`. The error-page declaration in An Example Custom Servlet Exception Error Page Declaration on page 9-9 tells the Web container to catch this exception and forward the HTTP request to the `/error/ExceptionPage` custom error page. The HTML in this page is rendered as shown in Figure 9-5.



**Figure 9-5** The Example Custom Servlet Exception Error Page

Look closely at the `doGet` method in A Servlet That Throws an Exception on page 9-5. The method declares that the method throws the `ServletException` (in the `javax.servlet` package). Because this method is being overridden from the `HttpServlet` class, you cannot add new exceptions to the `throws` clause in the declaration. Therefore, every non-checked exception that can be thrown in the body of the servlet code must be caught in a try-catch block. When caught, the exception is wrapped in a new `ServletException` and then that exception is thrown out of the method. This is a common practice in servlet programming.

# Developing an Error Handling Servlet

As mentioned earlier, the custom error page can be a static HTML page or a dynamic servlet that is called by the Web container to respond to an error. In this section, you will see how to create an error handling servlet.

The /error/ExceptionPage error page is a servlet. This URL location is mapped to the ExceptionDisplay servlet class. Custom error pages that are implemented as servlets should override both the doGet and doPost methods. The Web container will activate the error page servlet with the same HTTP method activated the original servlet. If the response should be identical, then these methods can simply dispatch to a common method. This is shown in Code 9-4.

**Code 9-4**      The ExceptionDisplay Servlet Class

```
12 public final class ExceptionDisplay extends HttpServlet {  
13  
14     public void doGet(HttpServletRequest request,  
15                         HttpServletResponse response)  
16         throws IOException {  
17         generateResponse(request, response);  
18     }  
19  
20     public void doPost(HttpServletRequest request,  
21                         HttpServletResponse response)  
22         throws IOException {  
23         generateResponse(request, response);  
24     }  
25 }
```

Before the error page servlet is activated, the Web container adds two attributes to the request scope:

- `javax.servlet.error.exception` – This attribute holds the actual exception object thrown by the original servlet. If the servlet threw a `ServletException`, then this attribute is the original exception embedded in the `ServletException` object. This original exception is also called the root cause.
- `javax.servlet.error.request_uri` – This attribute holds a `String` of the request URL of the servlet in which the error occurred. That is, the page or resource that the user had originally requested.

The `ExceptionDisplay` servlet uses these two request attributes to dynamically create the display shown in Figure 9-5 on page 9-10. The name of the Java technology exception class is shown in the banner of the page and the request URL is listed in the main body of the response page. Access to these request attributes is handled by the `getAttribute` method. This is shown in Code 9-5.

**Code 9-5**      The `ExceptionDisplay` Servlet Class (continued)

```
26 public void generateResponse(HttpServletRequest request,
27                               HttpServletResponse response)
28 throws IOException {
29
30     response.setContentType("text/html");
31     PrintWriter out = response.getWriter();
32
33     Throwable exception
34     = (Throwable) request.getAttribute("javax.servlet.error.exception");
35     String expTypeFullName
36     = exception.getClass().getName();
37     String expTypeName
38     = expTypeFullName.substring(expTypeFullName.lastIndexOf(".") + 1);
39     String request_uri
40     = (String) request.getAttribute("javax.servlet.error.request_uri");
41
```

The `ExceptionDisplay` servlet uses the content of the `exception` and `request_uri` attributes to generate the HTML response for the error page. This code is shown in Code 9-6.

**Code 9-6** The `ExceptionDisplay` Servlet Class (continued)

```

42     out.println("<HTML>");
43     out.println("<HEAD>");
44     out.println("<TITLE>Servlet Exception</TITLE>");
45     out.println("</HEAD>");
46     out.println("<BODY BGCOLOR='white'>");
47     out.println("<TABLE BORDER='0' CELLSPACING='0' CELLPADDING='0' "
48 WIDTH='600'>");
49     out.println("<TR>");
50     out.println("  <TD ALIGN='center' VALIGN='center'>");
51     out.println("    <IMG SRC='images/askduke.gif' ALT='Ask Duke!'>");
52     out.println("  </TD>");
53     out.println("  <TD BGCOLOR='pink' ALIGN='center' VALIGN='center'>");
54     out.print("    <FONT SIZE='5' COLOR='red'><B>");
55     out.print(expTypeName);
56     out.println("</B></FONT>");
57     out.println("  </TD>");
58     out.println("</TR>");
59     out.println("<TR HEIGHT='15'><TD HEIGHT='15'><!-- vertical space --></TD></TR>");
60     out.println("<TR>");
61     out.println("  <TD></TD>");
62     out.println("  <TD>");
63     out.print("    <B>");
64     out.print(exception.getMessage());
65     out.println("    </B><BR><BR>");
66     out.println("This was the request URI: <BR>");
67     out.println("<CODE>" + request_uri + "</CODE><BR><BR>");
68     out.println("This is the complete stack trace of the exception:");
69     out.println("<FONT SIZE='1'><PRE>");
70     exception.printStackTrace(out);
71     out.println("</PRE></FONT>");
72     out.println("  </TD>");
73     out.println("</TR>");
74     out.println("</TABLE>");
75     out.println("</BODY>");
76     out.println("</HTML>");
```

}

## Programmatic Exception Handling

Declarative exception handling is powerful and easy to use, but it might not be appropriate for all situations. Programmatic exception handling is another technique. Programmatic exception handling only applies to Java technology exceptions thrown by servlets; it is not appropriate for handling HTTP errors.

In programmatic exception handling, you write your servlet code to catch all exceptions and handle them directly, as opposed to letting the Web container do this for you.

To handle exceptions programmatically, wrap all of your error-prone business logic in a try-catch block. This code is shown in Code 9-7 on page 9-15. In the catch clause, you can forward the request to the exception handling error page using a RequestDispatcher object (Line 33). The request dispatcher object is retrieved from the servlet context using the getNamedDispatcher method (Line 29). The string passed to the getNamedDispatcher method must be the name of the exception handling servlet defined in the deployment descriptor. You can also pass request attributes to the exception handling servlet (Lines 30–32).

**Code 9-7** A Servlet That Throws a NullPointerException

```
14 public final class ErrorProneServlet extends HttpServlet {  
15  
16     public void doGet(HttpServletRequest request,  
17                         HttpServletResponse response)  
18         throws IOException, ServletException {  
19         String string = null;  
20  
21         try {  
22             // Attempt to access the first character of a null String object  
23             string.charAt(0);  
24  
25             // Catch exceptions and forward to the Exception Handler servlet  
26         } catch (Exception e) {  
27             ServletContext context = getServletContext();  
28             RequestDispatcher errorPage  
29                 = context.getNamedDispatcher("ExceptionHandler");  
30             request.setAttribute("javax.servlet.error.exception", e);  
31             request.setAttribute("javax.servlet.error.request_uri",  
32                                 request.getRequestURI());  
33             errorPage.forward(request, response);  
34         }  
35     }  
36 }  
37 }
```

## Exception Handling Servlet Declarations

The exception handling servlet must be declared in the Web application deployment descriptor. You do not need to specify a URL mapping for exception handling servlets, however. The servlet definition for the `ExceptionHandler` servlet is shown in Code 9-8. That name is used in the call to the `getNamedDispatcher` method (see Line 29 in Code 9-7 on page 9-15).

**Code 9-8**      Servlet Exception Error Page Definition

```
22 <servlet>
23   <servlet-name>ExceptionHandler</servlet-name>
24   <description>
25     The servlet that handles displaying any exception
26     thrown from another servlet.
27   </description>
28   <servlet-class>sl314.web.ExceptionDisplay</servlet-class>
29 </servlet>
```

Do not create a URL mapping for the exception handler servlets. When you create a URL mapping, the mapping is exposed to the users of the Web application. The user could try to activate the exception handler servlet by entering the URL directly into the Web browser. Creating a servlet definition without a URL mapping for the exception handling servlet ensures that it will be hidden from the user.

## Trade-offs for Declarative Exception Handling

The *advantage* of declarative exception handling is that it is easy to implement. All you need to do is add one or more error-page declarations to the deployment descriptor.

The *disadvantages* of declarative exception handling are:

- You must create a URL mapping for every exception handling servlet. This exposes your exception handling servlets to the user. The user might activate the servlet by entering the URL into the Web browser, which might have some undesirable effects on the Web application.
- It is often too generic.

When one servlet throws an `SQLException` (for example), that situation might not have the same meaning as another servlet throwing an `SQLException`. If you use declarative error handling, the Web application cannot usefully distinguish these two errors for the user. The user sees only one relatively generic error page for all SQL exceptions thrown across the whole Web application.

## Trade-offs for Programmatic Exception Handling

The *advantages* of programmatic exception handling are:

- It keeps the handler code close to the Controller.

It is often preferable to co-locate the exception handling code with the code that produces the exception. This helps developers (especially new developers maintaining older servlet code) see clearly how a given exception is being handled.
- It makes dealing with exceptions explicit.

Declarative exception handling hides the error handling details from the servlet code. Programmatic exception handling makes those details explicit.
- The handler can be customized to the situation.

As mentioned earlier, an SQL exception in one servlet may have a different meaning than it does in another servlet. Programmatic exception handling allows the developer to customize the error page View that the user gets from each servlet.

The *disadvantages* of programmatic exception handling are:

- It requires more code to implement.

To support programmatic exception handling, you must wrap all the error-prone code of a servlet within a try-catch block. In each catch block, you must use a request dispatcher to forward the HTTP request to the appropriate error page handler.

The request dispatcher code in the catch clause is not much more code than in the declarative exception handling paradigm, because you need to use a try-catch block and wrap the exceptions in a `ServletException` object in the declarative paradigm.

---

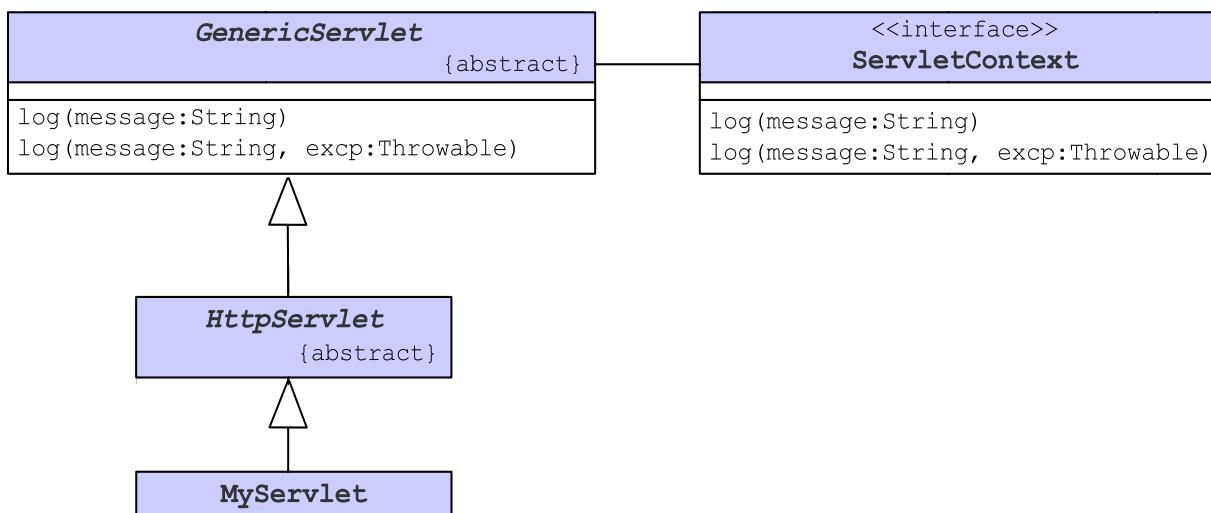
**Note** – A single Web application can use a combination of declarative and programmatic exception handling.

---



# Logging Exceptions

Whenever a servlet throws an exception, it is a good idea to write the exception to a log file. This logging feature is built into the `GenericServlet` and `ServletContext` classes. You can use either the `log(String)` or `log(String, Throwable)` methods. This is illustrated in Figure 9-6.



**Figure 9-6** The Logging API

## Summary

This module presented methods for handling errors in Web applications. This is the important information to know about declarative and programmatic error handling:

- There are two types of errors in a Web application: HTTP errors and Java technology exceptions.
- You can use the `error-page` deployment descriptor element to declare error pages for both types.
- You must wrap checked exceptions in a `ServletException` object in your `doXYZ` methods.
- An error handling servlet has access to two request attributes (`javax.servlet.error.exception` and `javax.servlet.error.request_uri`).
- For programmatic error handling, you can use a `RequestDispatcher` object to forward the HTTP request directly to the error handling servlet.

## Certification Exam Notes

This module presented most of the objectives for Section 4, "Designing and Developing Servlets to Handle Server-side Exceptions," of the Sun Certification Web Component Developer certification exam:

- 4.1 For each of the following cases, identify correctly constructed code for handling business logic exceptions, and match that code with correct statements about the code's behavior: return an HTTP error using the `sendError` response method; and return an HTTP error using the `setStatus` method.
- 4.2 Given a set of business logic exceptions, identify the following: the configuration that the deployment descriptor uses to handle each exception; how to use a `RequestDispatcher` to forward the request to an error page; and specify the handling declaratively in the deployment descriptor.
- 4.3 Identify the method used for the following: write a message to the Web application log; write a message and an exception to the Web application log.

Objective 4.1 is not presented at all in this course. Review the Servlet specification (v2.3) for more details.



# Configuring Web Application Security

---

## Objectives

Upon completion of this module, you should be able to:

- Explain the importance of Web security
- Use the deployment descriptor to configure authorization for a Web application resource
- Use the deployment descriptor to configure authentication of users of a Web application

## Relevance



**Discussion** – The following questions are relevant to understanding why Web security is important:

- Have you ever used a Web site in which you provided confidential information about yourself or your financial records (for example, a credit card number)? What precautions did the developers of that Web site use to protect your confidential data?
  
- Have you ever clicked an HTML link and then been prompted by your Web browser for a username and password? Why do you think that happened?

## Additional Resources



**Additional resources** – The following reference provides additional information on the topics described in this module:

- Stein, Lincoln D. *Web Security*. Reading: Addison-Wesley Longman, Inc., 1997.

## Web Security Issues

Security is critical to any Web application because the Web server is exposed to the Internet directly. Malicious or benign users, sometimes called crackers, might try to break into your Web server and access secure information or applications. This could easily jeopardize your business data as well as the identity and confidential information of your customers.

Web security is a challenging field. It is difficult to keep up-to-date on the latest security measures, and crackers are constantly finding holes in existing security measures. It is impossible to have perfect security, but minimal security is much better than no security. This module presents a few important security issues and discusses how the Web container can be used to enforce certain security measures. The Servlet specification does not describe all of the security issues presented in this module. There are notes in this text about security measures that are discussed in the specification and some that are not.

## Authentication

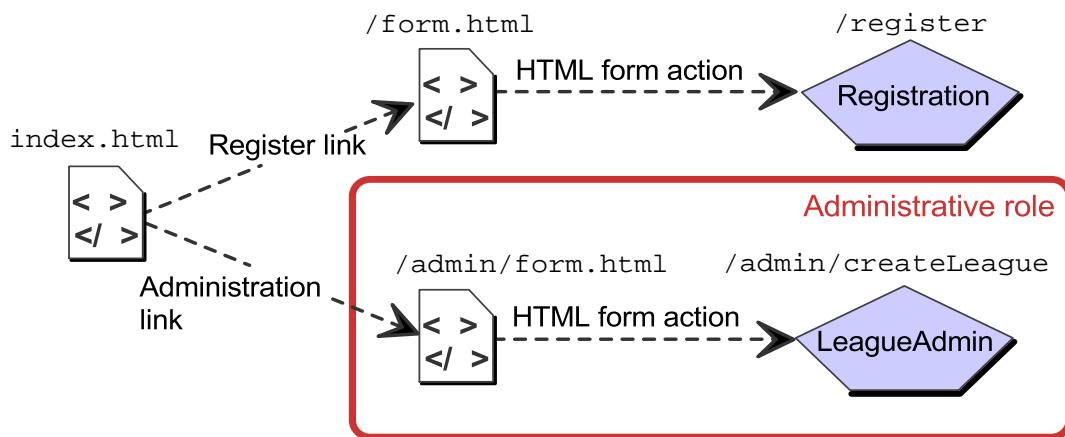
*Authentication* is the process of verifying the user's identity. This is usually done with a user name and password (sometimes called a passphrase).

Authentication is a security measure that can be configured in the Web container. You can select one of four authentication techniques:

- BASIC – The Web browser solicits the user name and password and sends the data to the Web server “in the open.”
- DIGEST – The Web browser solicits the user name and password and sends this data to the Web server, which has been encoded using an algorithm (such as MD5).
- FORM – The Web application supplies an HTML form that is sent to the Web browser.
- CLIENT-CERT – The Web container uses Secure Sockets Layer (SSL) to verify the user. The user must have an SSL certificate.

## Authorization

*Authorization* is the process of partitioning Web resources based on user roles. For example, the Soccer League Web application might include a service to league coordinators to allow them to create new leagues. The developer of this Web application might create an admin directory in the Web site hierarchy which includes the “create league” form and the servlet mapping for the “create league” servlet. This is illustrated in Figure 10-1.



**Figure 10-1** Soccer League Administration Security Domain

The mapping between a Web resource collection (a related set of Web pages and servlets) and a given user role is called a security domain.

Authentication and authorization are usually used together. Authentication is used to identify the user, and authorization is used to verify that the user belongs to the specified security role for the requested Web resource.

Authorization is a security measure that can be configured in the Web container.

## Maintaining Data Integrity

*Data integrity* is the result of securing the transmission of your confidential data using encryption. Data sent across the network is vulnerable in two ways:

- Data can be observed or intercepted.  
Maintaining data integrity guarantees confidentiality; only the proper recipient will be able to read the contents of the message.
- Data can be corrupted during transmission.  
Maintaining data integrity guarantees content integrity; no other party has intercepted the message and modified it before the data reaches the intended user.

You can protect data against both risks using the Secure Hypertext Transfer Protocol (HTTPS). The HTTPS protocol is the same as the HTTP protocol, but the underlying socket layer uses the SSL protocol to encrypt the HTTP request and response streams.

Data integrity is a security measure that can be configured in the Web container.

## Access Tracking

*Access tracking* (also called *auditing*) is the process of keeping records of every access to your Web application. You can audit the actions that occur in your Web application by writing them to log files using the following methods:

- `log(String)` – Prints the message to a log file
- `log(String, Throwable)` – Prints the message and the stack trace of the exception to a log file

These methods exist in both the `ServletContext` interface and the `GenericServlet` class.

## Dealing With Malicious Code

*Malicious code* is designed to damage the system on which it is running. In general, malicious code includes viruses, worms, Trojan horses, and so on. In Web applications, malicious code is code on the server that has the hidden intent of damaging the server.

Internet service providers need to partition Web applications to prevent malicious code attacks across Web applications. A Web container can isolate Web applications using a Java technology SecurityManager. This is not required in the Servlet specification; therefore, it is a “value added” feature for vendors.

---

**Note** – Dealing with malicious code is not a security measure that can be configured in the Web container.

---



## Dealing With Web Attacks

*Web attacks* are attempts to compromise a server by an outside individual or group. Web attacks include “denial of service,” cracking server passwords, downloading confidential files, and so on.

There is nothing in the Servlet specification to safeguard against Web attacks. You can prevent some Web attacks by using a firewall. For example, you can prevent all external TCP connections that are not HTTP or HTTPS requests.

Even installing a firewall is not sufficient protection against some Web attacks. For example, a denial of service attack might take the form of hundreds of HTTP requests per second sent by one or more attacking computers.

---

**Note** – Dealing with Web attacks is not a security measure that can be configured in the Web container.

---



## Declarative Authorization

To implement declarative authorization, you must:

- Identify the Web resource collections
- Identify the roles
- Map the Web resource collection to the roles
- Identify the users in each of those roles

### Web Resource Collection

A Web resource collection is a group of Web pages and servlet Universal Resource Identifier (URIs). Usually, a Web resource collection has a common theme: administration, distinct business uses, preferred customers, and so on. An example Web resource collection is illustrated in Figure 10-1 on page 10-4. The `form.html` page and `createLeague` servlet in the `admin` directory is the Web resource collection.

A Web resource collection is configured in the deployment descriptor. The `web-resource-collection` element includes two important subelements: `url-pattern` and `http-method`. The Soccer League administration Web resource collection is shown in Code 10-1.

**Code 10-1**      Soccer League Administration Web Resource Collection

```
48 <web-resource-collection>
49   <web-resource-name>League Admin</web-resource-name>
50   <description>
51     Resources accessible only to administrators.
52   </description>
53   <url-pattern>/admin/*</url-pattern>
54   <http-method>POST</http-method>
55   <http-method>GET</http-method>
56 </web-resource-collection>
```

The `url-pattern` element specifies a URL that is relative to the Web application context root. The `url-pattern` element can be a specific resource or a pattern that handles multiple resources. The `*` character is used as a wildcard in the URL pattern. In the example, Line 53 will match any static page, servlet mapping, and subdirectories under the `admin` directory in the Soccer League Web application. You can specify any number of `url-pattern` elements in a Web resource collection configuration.

The `http-method` element specifies the HTTP method for the requests that must be authorized by the Web container for this Web resource collection. You can specify any number of `http-method` elements in a Web resource collection configuration. It is a good practice to include both POST and GET HTTP methods.

## Declaring Security Roles

The next step is to configure the security roles of users that are authorized to access the Web resource collection. This is configured in a `security-constraint` element. The `web-resource-collection` element is embedded in the `security-constraint` element. Additionally, an `auth-constraint` element is included in the `security constraint`. The `auth-constraint` element includes a `role-name` subelement. In the Soccer League example, the League Administration Web resource collection is authorized for only the `administrator` role (Line 58). This is shown in Code 10-2 on page 10-9.

**Code 10-2** Soccer League Administration Security Constraint

```

47   <security-constraint>
48     <web-resource-collection>
49       <web-resource-name>League Admin</web-resource-name>
50       <description>
51         Resources accessible only to administrators.
52       </description>
53       <url-pattern>/admin/*</url-pattern>
54       <http-method>POST</http-method>
55       <http-method>GET</http-method>
56     </web-resource-collection>
57     <auth-constraint>
58       <role-name>administrator</role-name>
59     </auth-constraint>
60   </security-constraint>
61
62   <login-config>
63     <auth-method>BASIC</auth-method>
64   </login-config>
65
66   <security-role>
67     <description>A restricted-access user role.</description>
68     <role-name>administrator</role-name>
69   </security-role>

```

Each security role must be declared in a `security-role` element in the deployment descriptor (Lines 66–69).

## Security Realms

A *security realm* is a software component for matching users to roles. It also verifies the user's password. Every Web container must include a security realm, but this mechanism is *vendor-specific*.

There are many possible mechanisms:

- Flat-file (the `MemoryRealm` class in the Tomcat server)
- Database tables (the `JDBCRealm` class in the Tomcat server)
- Lightweight Directory Access Protocol (LDAP)
- Network Information System (NIS)

Security realms are used for both authentication (verifying the user's identity) and authorization (verifying that the user belongs to a specific security role).

## Declarative Authentication

Use the deployment descriptor to declare the Web application's authentication technique:

```
62      <login-config>
63          <auth-method>BASIC</auth-method>
64      </login-config>
```

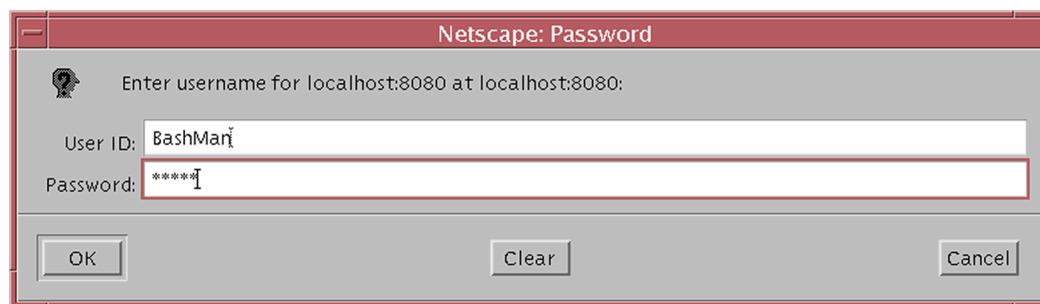
The auth-method element can take one of four values:

- BASIC – The Web browser solicits the user name and password and sends the data to the Web server “in the open.”
- DIGEST – The Web browser solicits the user name and password and sends a ‘digest’ (a mathematical algorithm, such as MD5) to the Web server.
- FORM – The Web application supplies an HTML form that is sent to the Web browser.
- CLIENT-CERT – The Web container uses SSL to verify the user. The user must have an SSL certificate.

CLIENT-CERT uses SSL and is the most secure of the four techniques, but requires the user to have an X-509 certificate. The BASIC, DIGEST, and FORM authentication methods send the user name and password data from the Web browser to the Web container using either weak or no encryption. These techniques can be combined with server-side SSL to become as secure as CLIENT-CERT.

## BASIC Authentication

The BASIC authentication method uses the built-in HTTP BASIC authentication protocol. There is nothing that you, as the Web component developer, need to do to take advantage of this mechanism. When the Web container decides to authenticate a user for a confidential request, the Web container sends an HTTP challenge back to the Web browser. The Web browser then must prompt the user for a user name and password. Netscape's authentication dialog box is shown in Figure 10-2.



**Figure 10-2**    Netscape's Authentication Dialog Box

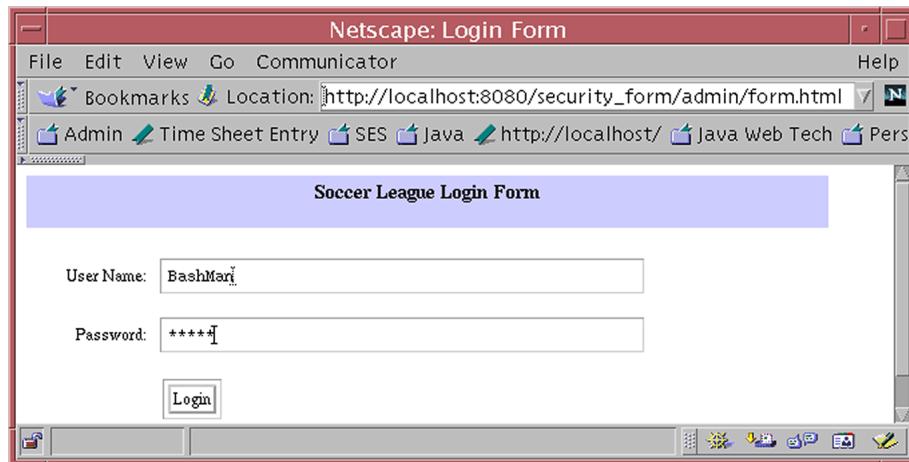
When the user enters a user name and password, the Web browser sends this data back to the Web container. The Web container verifies the data against the vendor-specific security realm. If the password is verified and the user is a member of the required security role, then the Web container activates the original HTTP request, which could be a static HTML page or the activation of a servlet.

**Form-based Authentication** BASIC HTTP authentication is usually considered a brute-force and ugly mechanism. Often, the Web design team wants to create an HTML page that presents a more elegant authentication entry form. The Servlet specification allows you to configure the Web application to perform an authentication using your own HTML pages. This is configured in the `login-config` element as shown in Code 10-3.

**Code 10-3** Soccer League Form-based Authentication

```
62 <login-config>
63   <auth-method>FORM</auth-method>
64   <form-login-config>
65     <form-login-page>/login/form.html</form-login-page>
66     <form-error-page>/login/error.html</form-error-page>
67   </form-login-config>
68 </login-config>
```

Now, when the user attempts to access a restricted Web resource, the Web container sends the login form (specified in the `form-login-page` element on Line 65) back to the Web browser. The example Soccer League Login Form is shown in Figure 10-3.



**Figure 10-3** Soccer League Authentication Form

This HTML form is special. The Servlet specification mandates that the ACTION attribute of the form must be the phrase j\_security\_check (all lowercase). Likewise, the user name and password input fields must be j\_username and j\_password, respectively. The HTML code of the Soccer League login form is shown in Code 10-4.

**Code 10-4**      HTML Code of the Soccer League Login Form

```
1  <HTML>
2
3  <HEAD>
4  <TITLE>Login Form</TITLE>
5  </HEAD>
6
7  <BODY BGCOLOR='white'>
8
9  <TABLE BORDER='0' CELLSPACING='0' CELLPADDING='5' WIDTH='600'>
10 <TR>
11    <TD BGCOLOR='#CCCCFF' ALIGN='center'>
12      <H3>Soccer League Login Form</H3>
13    </TD>
14  </TR>
15 </TABLE>
16
17 <FORM ACTION='j_security_check' METHOD='POST'>
18
19 <TABLE BORDER='0' CELLSPACING='0' CELLPADDING='5' WIDTH='600'>
20 <TR>
21   <TD ALIGN='right'>User Name:</TD>
22   <TD><INPUT TYPE='text' NAME='j_username' SIZE='50'></TD>
23 </TR>
24 <TR>
25   <TD ALIGN='right'>Password:</TD>
26   <TD><INPUT TYPE='password' NAME='j_password' SIZE='50'></TD>
27 </TR>
28 <TR>
29   <TD></TD>
30   <TD><INPUT TYPE='submit' VALUE='Login'></TD>
31 </TR>
32 </TABLE>
33
34 </FORM>
35
36 </BODY>
37
38 </HTML>
```

When the user fills in this form and selects the Submit button, the Web container intercepts the `j_security_check` action and handles the authentication. The rest of the authorization process is the same as before. There is nothing that you need to code to make the `j_security_check` action work. The Web container implements the servlet that executes when the `j_security_check` action is received.

## Summary

This module presented Web application security issues and measures. These are the key ideas:

- There are six main security issues: authorization, authentication, data integrity, tracking access, dealing with malicious code, and dealing with Web attacks.
- You can use the deployment descriptor to configure the authorization security domains in your Web application.
- You can use the deployment descriptor to configure the authentication technique for your Web application.

## Certification Exam Notes

This module presented all of the objectives for Section 6, "Designing and Developing Secure Web Applications," of the Sun Certification Web Component Developer certification exam:

6.1 Identify correct descriptions or statements about the security issues:

- Authentication
- Authorization
- Data integrity
- Auditing
- Malicious code
- Web site attacks

6.2 Identify the deployment descriptor element names, and their structure, that declare the following:

- A security constraint
- A Web resource
- The login configuration
- A security role

6.3 Given an authentication type, BASIC, DIGEST, FORM, and CLIENT-CERT, identify the correct definition of its mechanism.

# Understanding Web Application Concurrency Issues

---

## Objectives

Upon completion of this module, you should be able to:

- Describe why servlets need to be thread-safe
- Describe the attribute scope rules and the corresponding concurrency issues
- Describe the single thread model for servlets and the issues with this concurrency strategy
- Design a Web application for concurrency

## Additional Resources

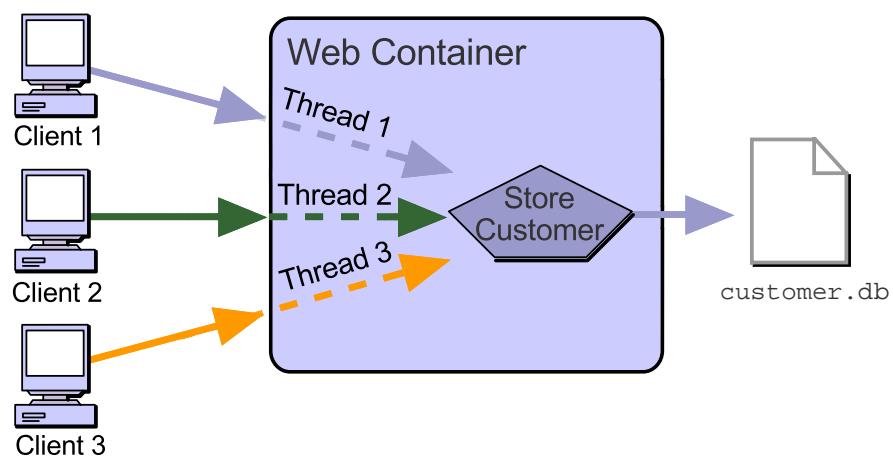


**Additional resources** – The following reference provides additional information on concurrency issues:

- Arnold, Ken, James Gosling, David Holmes. *The Java™ Programming Language, Third Edition*, Boston: Addison-Wesley, 2000.
- Lea, Doug. *Concurrent Programming in Java™ Second Edition*, Reading: Addison-Wesley, 2000.

# The Need for Servlet Concurrency Management

For every HTTP request that is processed by the Web container, a separate Java technology thread is used to handle the processing of that request. By default, the Web container will *only* make one object instance for every servlet definition (as declared in the deployment descriptor). Therefore, all threads that are processing requests to the same servlet definition are executed against the same object instance. This is illustrated in Figure 11-1.



**Figure 11-1** Multiple Simultaneous Requests to a Servlet

## Concurrency Management Example

The following example, the `StoreCustomer` servlet, is contrived to clearly demonstrate the problems of improper concurrency management. This servlet takes five CGI parameters (a person's name and address) and stores that data in a flat file (one line per person). This servlet works by writing each customer record on a single line in a flat file with each field in the customer record separated by the bar “|” character. The servlet uses a `FileWriter` object that is referenced by the `customerDataWriter` instance variable. This variable is initialized in the servlet's `init` method. The body of the `doGet` method is shown in Code 11-1 on page 11-5.

**Code 11-1**    Servlet That Is Not Thread-Safe

```
41 public void doGet(HttpServletRequest request,
42                     HttpServletResponse response)
43                     throws IOException {
44
45     // Extract all of the request parameters
46     String name = request.getParameter("name");
47     String address = request.getParameter("address");
48     String city = request.getParameter("city");
49     String province = request.getParameter("province");
50     String postalCode = request.getParameter("postalCode");
51
52     // Store the customer data to the flat-file
53     customerDataWriter.write(name, 0, name.length());
54     customerDataWriter.write('|');
55     customerDataWriter.write(address, 0, address.length());
56     customerDataWriter.write('|');
57     customerDataWriter.write(city, 0, city.length());
58     customerDataWriter.write('|');
59 // Simulate a suspension of the current thread.
60 try { Thread.sleep(1000); } catch (Exception e) { /* ignore */ }
61     customerDataWriter.write(province, 0, province.length());
62     customerDataWriter.write('|');
63     customerDataWriter.write(postalCode, 0, postalCode.length());
64     customerDataWriter.write('\n');
65     customerDataWriter.flush();
66
67     // Specify the content type is HTML
68     response.setContentType("text/html");
69     PrintWriter out = response.getWriter();
70
71     // Generate the HTML response
72     out.println("<HTML>");
73     out.println("<HEAD>");
74     out.println("<TITLE>Bad Threading Servlet</TITLE>");
75     out.println("</HEAD>");
76     out.println("<BODY BGCOLOR='white'>");
77     out.println("Customer, " + name + " stored.");
78     out.println("</BODY>");
79     out.println("</HTML>");
80 }
```

The problem with this servlet is that every request uses the same `FileWriter` object. One request might be suspended by the JVM in the middle of writing customer data to the flat file; and at the same time, another request might start writing another customer's data. This will corrupt both of these data records in the file.

The solution to this problem is to wrap a synchronized block around the critical code in the `doGet` method. In this example, the critical code is the code that writes to the flat file (Lines 53–65 in Code 11-1). This is the critical code because the `FileWriter` object (in the `customerDataWriter` attribute) is a shared resource. It is shared across all simultaneous requests on that servlet. Therefore, you would lock the `FileWriter` object in the `customerDataWriter` attribute. This is shown in Lines 53–67 in Code 11-2 on page 11-7.

**Code 11-2**      Servlet That Is Thread-Safe

```
41 public void doGet(HttpServletRequest request,
42                     HttpServletResponse response)
43             throws IOException {
44
45     // Extract all of the request parameters
46     String name = request.getParameter("name");
47     String address = request.getParameter("address");
48     String city = request.getParameter("city");
49     String province = request.getParameter("province");
50     String postalCode = request.getParameter("postalCode");
51
52     // Store the customer data to the flat-file
53     synchronized (customerDataWriter) {
54         customerDataWriter.write(name, 0, name.length());
55         customerDataWriter.write('|');
56         customerDataWriter.write(address, 0, address.length());
57         customerDataWriter.write('|');
58         customerDataWriter.write(city, 0, city.length());
59         customerDataWriter.write('|');
60     // Simulate a suspension of the current thread.
61     try { Thread.sleep(1000); } catch (Exception e) { /* ignore */ }
62     customerDataWriter.write(province, 0, province.length());
63     customerDataWriter.write('|');
64     customerDataWriter.write(postalCode, 0, postalCode.length());
65     customerDataWriter.write('\n');
66     customerDataWriter.flush();
67 }
68
69     // Specify the content type is HTML
70     response.setContentType("text/html");
71     PrintWriter out = response.getWriter();
72
73     // Generate the HTML response
74     out.println("<HTML>");
75     out.println("<HEAD>");
76     out.println("<TITLE>Good Threading Servlet</TITLE>");
77     out.println("</HEAD>");
78     out.println("<BODY BGCOLOR='white'>");
79     out.println("Customer, " + name + " stored.");
80     out.println("</BODY>");
81     out.println("</HTML>");
82 }
```

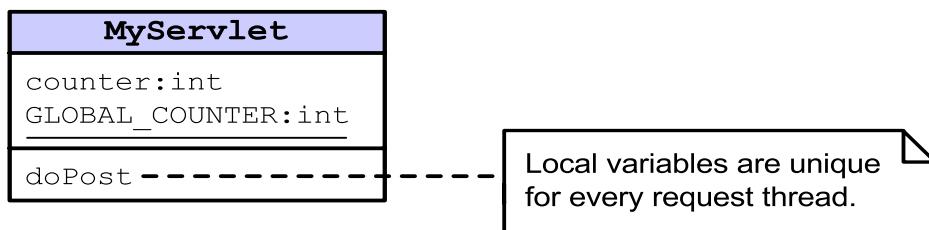
## Attributes and Scope

There are six scopes for attributes in a Web application:

- Local variables (also called the page scope)
- Instance variables
- Class variables
- Request attributes (also called the request scope)
- Session attributes (also called the session scope)
- Context attributes (also called the application scope)

## Local Variables

Local variables are attributes that exist within the scope of the service method (or methods called by the service method, such as the doPost method). An example of a local variable is illustrated in Figure 11-2. The values of these attributes exist within the stack of the thread that is executing the service method. These attributes *are* thread-safe.

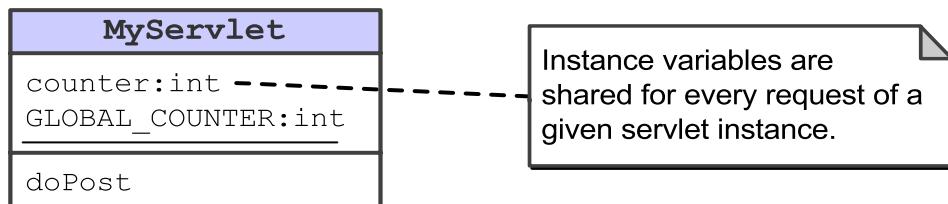


**Figure 11-2** The Local Variable Scope

The best use for local attributes is to store temporary data during the processing of a single request.

## Instance Variables

Instance variables are attributes of the servlet object created to represent a specific servlet definition. An example of an instance variable is illustrated in Figure 11-3. These attributes are shared across all HTTP requests that are processed by this servlet definition. These attributes *are not* thread-safe.



**Figure 11-3** The Instance Variable Scope

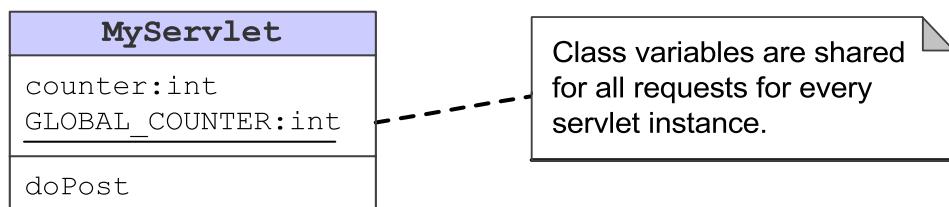
The best use for instance variables is to store the servlet definitions configuration parameters as read-only values. If you want to use an instance variable to store a read-write value, then you should synchronize the code that changes that value by locking the servlet instance (see Code 11-3).

**Code 11-3** Read-Write Access to Instance Variables

```
synchronized (this) {  
    this.counter++;  
}
```

## Class Variables

Class variables are attributes that exist statically within the servlet class. An example of a class variable is illustrated in Figure 11-4. A servlet class may be used by multiple servlet definitions within a Web application. Therefore, these attributes are shared across all HTTP requests for all servlet definitions that use this servlet class. These attributes *are not* thread-safe.



**Figure 11-4** The Class Variable Scope

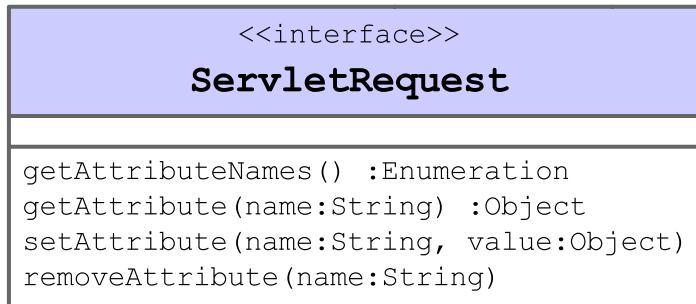
The best use for class variables is to share common data across multiple servlet definitions. It is best if these attributes are read-only (that is, they are declared as `static` and `final`). If you need a read-write attribute, then you should synchronize the code that changes that value by locking the class object representing the servlet class (see Code 11-4).

**Code 11-4** Read-Write Access to Class Variables

```
synchronized (MyServlet.class) {
    MyServlet.GLOBAL_COUNTER++;
}
```

## Request Scope

Request scoped attributes exist within the `ServletRequest` object that is passed to the `service` method. The API for the request scope is illustrated in Figure 11-5. These attributes *are* thread-safe.



**Figure 11-5** The Request Scope API

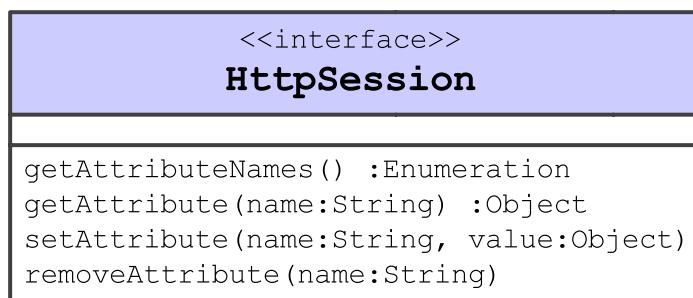
The best use for request scoped attributes is to pass data from the servlet Controller to the View. These attributes are used to generate a dynamic presentation by the View.

**Note** – The `request` attribute API is part of the generic `ServletRequest` interface. These methods are inherited by the `HttpServletRequest` interface.



## Session Scope

Session scoped attributes exist within the `HttpSession` object. The API for the session scope is illustrated in Figure 11-6. A session may exist across multiple HTTP requests for a single client. While it might appear that there are no concurrency issues with the session scope, that is not always the case. It is possible for the user to launch multiple Web browsers and interact with the Web application by initiating multiple, simultaneous requests. Therefore, multiple request threads may be accessing the session attributes concurrently. These attributes *are not* thread-safe.



**Figure 11-6** The Session Scope API

The best use for session scoped attributes is to store data that must be used across multiple requests. For example, a “shopping cart” attribute must exist throughout the user’s shopping and browsing requests as well as during check out.

Typically, a session attribute is set once and retrieved many times. The object of that attribute should be designed to be thread-safe. However, if you need to change the value of a session attribute frequently, then you should synchronize the code that changes that value by locking the session object. This is shown in Code 11-5.

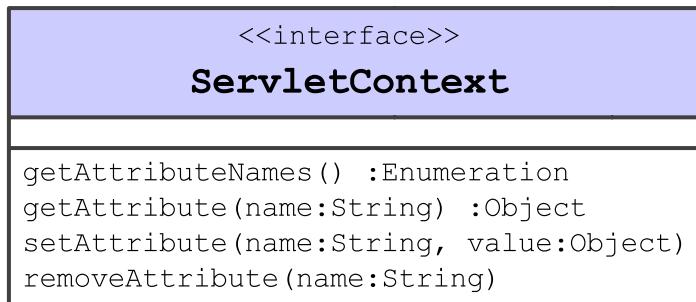
### Code 11-5     Read-Write Access to Session Attributes

```

HttpSession session = request.getSession();
synchronized (session) {
    session.setAttribute("counter", new Integer(counter+1));
}
  
```

## Application Scope

Application scoped attributes exist within the `ServletContext` object. The API for the application scope is illustrated in Figure 11-7. Application attributes are accessible by any servlet throughout the whole Web application. These attributes *are not* thread-safe.



**Figure 11-7** The Application Scope API

The best use for application scoped attributes is to store shared resources across the whole Web application. For example, JDBC platform connection pool or data source objects are resources that should be shared across all servlets.

Typically, an application attribute is set once and retrieved many times. The object of that attribute should be designed to be thread-safe. However, if you need to change the value of an application attribute frequently, then you should synchronize the code that changes that value by locking the servlet context object. This is shown in Code 11-6.

### Code 11-6     Read-Write Access to Application Attributes

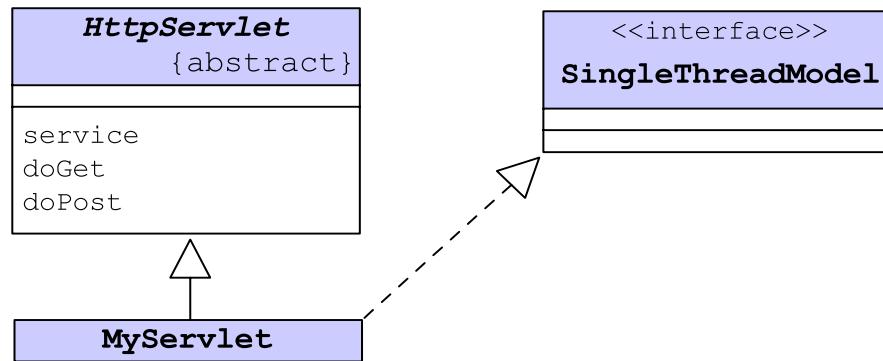
```
ServletContext context = getServletContext();
synchronized (context) {
    context.setAttribute("counter", new Integer(counter+1));
}
```

# The Single Threaded Model

This section describes a proposed technique for managing servlet concurrency. The section also discusses the limitations of this technique.

## The SingleThreadModel Interface

The Servlet specification ensures that if your servlet class implements SingleThreadModel (STM) interface, then only one request thread at a time executes the service method. The SingleThreadModel interface is illustrated in Figure 11-8.



**Figure 11-8** Using the SingleThreadModel Interface

The SingleThreadModel interface has no methods to implement. You can use STM to signal to the Web container that the servlet class must be handled specially.

## How the Web Container Might Implement the Single Threaded Model

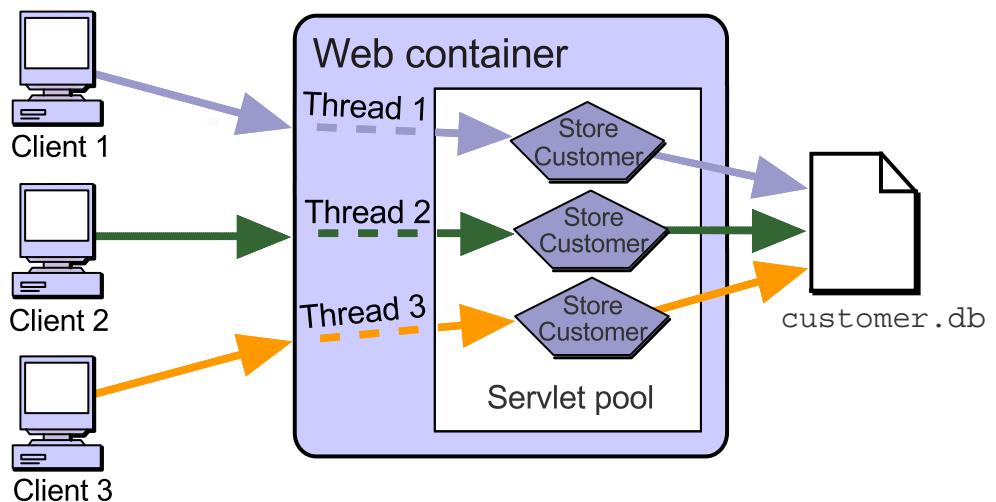
There are several ways that a Web container *might* implement the single threaded model:

- Queueing up all of the requests and passing them one at a time to a single servlet instance.
- Creating an infinitely large pool of servlet instances, each of which handles one request at a time.
- Creating a fixed-size pool of servlet instances, each of which handles one request at a time. If more requests arrive than the size of the pool, then some requests are postponed until some other request has finished.

These are suggestions made in the Servlet specification. The Web container vendor does not have to follow any of these techniques. Ultimately as a servlet developer, you do not know how the Web container handles STM servlets.

## STM and Concurrency Management

Using STM with your servlet does not guarantee that your servlet will be thread-safe. Consider the `StoreCustomer` servlet from the beginning of the module. That servlet used an instance variable to hold a reference to a `FileWriter` object. If this servlet implemented the STM interface, then it is possible that multiple servlet instances might be created with their own `FileWriter` object. So, access to that object is protected. Unfortunately, the `customer.db` file can still be corrupted at the operating system level when multiple, concurrent write operations occur on the same operating system file. This is illustrated in Figure 11-9.



**Figure 11-9** Multiple Simultaneous Requests to an STM Servlet

There are a few significant drawbacks of the STM mechanism:

- The implementation of the STM mechanism is vendor-specific. You cannot code your servlets with knowledge of a specific mechanism and know that your servlets will be able to port to another Web container vendor.
- The use of STM can significantly slow performance when the Web container uses the “one at a time” approach.
- The use of instance variables in STM servlets *is* thread-safe, but each servlet instance has its own copy of the variable.
- The use of STM does not control access to static (class scope) variables when the Web container uses the “servlet pool” approach. You must still use a `synchronized` block to access static variables of the servlet class.

## The Single Threaded Model

---

- The use of STM does not solve concurrency issues relative to accessing external, shared resources (such as the customer.db file).
- The use of STM does not solve concurrency issues for session and application scope attributes.

For all of these reasons, you should not use the `SingleThreadModel` interface.

# Recommended Approaches to Concurrency Management

Here are a few suggestions to help manage concurrency issues within your servlets:

- Whenever possible, use only local and request attributes.
- Use the `synchronized` syntax to control concurrency issues when accessing or updating frequently changing attributes and when using common resources (see Updating an Attribute in the Session Scope on page 11-19 and Reading an Attribute in the Session Scope on page 11-19).

## **Code 11-7** Updating an Attribute in the Session Scope

```
HttpSession session = session.getSession();
synchronized (session) {
    session.setAttribute("count", new Integer(count+1));
}
```

## **Code 11-8** Reading an Attribute in the Session Scope

```
HttpSession session = request.getSession();
synchronized (session) {
    countObj = (Integer) session.getAttribute("count");
}
count = countObj.intValue();
```

- Minimize the use of `synchronized` blocks and methods in your servlet class code. Never synchronize the whole `doGet` or `doPost` method.
- Synchronize on the common resource (such as a file stream object) and not on the servlet object. This can reduce the time that you wait for the resource.
- Use resource classes that have been properly designed for thread-safety. For example, you can assume that your JDBC technology-based driver vendor has designed a thread-safe `DataSource` class.

## Summary

This module presented servlet concurrency management issues. These are the key ideas:

- Using shared resources and multiple, concurrent requests can corrupt your data.
- Only local variables and request attributes are thread-safe; all other scopes are not thread-safe.
- Do not use the `SingleThreadModel` interface.
- Use the `synchronized` syntax to control concurrency issues when accessing or changing thread-unsafe attributes.
- Minimize the use of `synchronized` blocks and methods in your servlet class code.
- Use resource classes that have been properly designed for thread-safety.

# Certification Exam Notes

This module presented all of the objectives for Section 7, “Designing and Developing Thread-safe Servlets,” of the Sun Certification Web Component Developer certification exam:

- 7.1 Identify which attribute scopes are thread-safe: local variables, instance variables, class variables, request attributes, session attributes, context (application) attributes
- 7.2 Identify correct statements about differences between the multithreaded and single-threaded servlet models
- 7.3 Identify the interface used to declare that a servlet must use the single thread model



# Integrating Web Applications With Databases

---

## Objectives

Upon completion of this module, you should be able to:

- Understand what a database management system (DBMS) does
- Design a Web application to integrate with a DBMS
- Develop a Web application using a connection pool
- Develop a Web application using a data source and the Java Naming and Directory Interface (JNDI)

## Relevance



**Discussion** – The following questions are relevant to understanding the design decisions for integrating the Web tier with the database tier:

- Have you ever developed an application that integrates with the database (DB) tier? How did you develop the access logic to the RDBMS?
  
- Did you ever have to change the database design? How did that affect the various tiers in your application?

## Additional Resources



**Additional resources** – The following reference provides additional information on the topics described in this module:

- Alur, Deepak, John Crupi, Dan Malks, *Core J2EE Patterns*, Upper Saddle River, Prentice Hall PTR, 2001.

## Database Overview

A database is a collection of logically related data. A database is usually managed by a database management system (DBMS). This module uses the relational model.

In a relational database model, data is logically grouped into tables, where each table typically represents an entity or a relationship between entities. Tables are organized into rows and columns: A column represents a data attribute, and a row represents a record. Database entities represent objects in Java technology programs, and relationships represent connections between objects. For example, the league and player objects in the Soccer League application are represented by database entities, whereas the registration information is represented by a relationship between leagues and players.

Another concept that is fundamental to the relational database model is the idea that there should be a query language that allows the database to be managed. For an RDBMS, the language standard that supports database operations is called the Standard Query Language (SQL). An RDBMS allows you to perform four fundamental operations:

- Create a row in a table

This operation allows you to add a new object (entity) or relationship into the database.

- Retrieve one or more rows in a table

This operation allows you to retrieve one or more objects and their relationships to other objects.

- Update some data in a table

This operation allows you to modify the attributes of an entity or relationship.

- Delete one or more rows in a table

This operation allows you to remove one or more objects or relationships in the database.

## The JDBC API

JDBC is the Java technology API for interacting with a DBMS. The JDBC API includes interfaces that manage connections to the DBMS, statements to perform operations, and result sets that encapsulate the result of retrieval operations.



**Note –** This course does not teach the JDBC API. The Sun course SL-330 *Database Application Programming With Java™ Technology* presents the JDBC API.

---

You will learn techniques for designing and developing a Web application in which the JDBC technology code is encapsulated using the *Data Access Object (DAO)* design pattern.

When designing a Web application that uses a database, a major development consideration is how to manage database communication from the Web. One approach is to create a connection for each HTTP request that is processed. A servlet would connect to the database, perform its query, process the results, and close the connection when it completes the request.

This solution can lead to problems with speed and scalability. This solution can reduce the response time of each HTTP request, because the connection with the database takes time to establish. Furthermore, the servlet has to connect every time a client makes a request. Additionally, because databases generally support fewer simultaneous connections than a Web server, this solution limits the ability of the application to scale.

Another approach is to keep only a few connections that are shared among data access logic elements. In this technique, the application need only create a few connection objects at startup; therefore, the response time of any HTTP request does not incur the cost of creating a new connection object. This is called Connection Pooling.

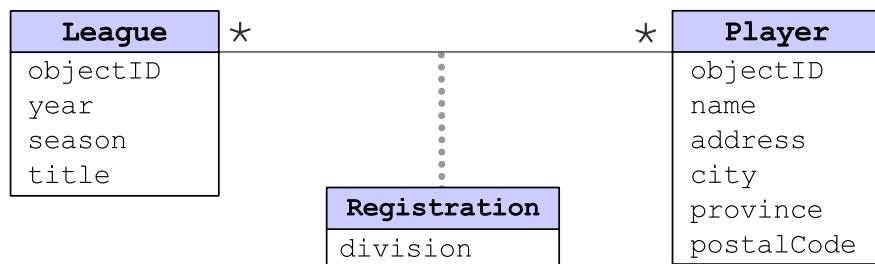
# Designing a Web Application That Integrates With a Database

To build a database application you must design the relationship between the Model objects and the corresponding database representation. To design the Model elements of an application you should perform the following tasks:

- Design the domain objects of your application
- Design the database tables that map to the domain objects
- Design the business services (the Model) to separate the database code into classes using the DAO pattern

## Domain Objects

Domain objects represent the real-world business entities of your application. The Soccer League example includes two main domain objects: the League that stores the year, season, and title of the league and the Player that stores the player's name and address. There is also an association called Registration that holds the division that the player is registering for within the league. This example domain model is shown in Figure 12-1.

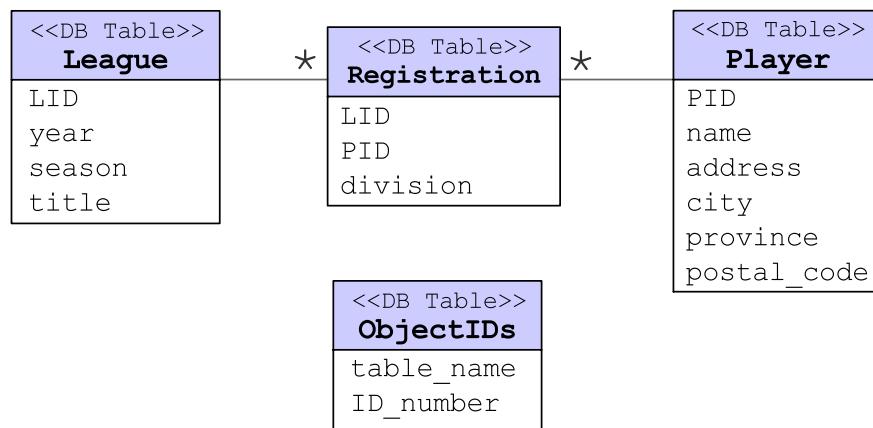


**Figure 12-1** Soccer League Domain Objects

The `objectID` has been added to the classes to provide a unique ID in the database table for each of these entities.

## Database Tables

In this case, the mapping between domain objects and the database schemata is clear. Each domain object has a corresponding DB table; for example, the League class maps to the League table. Also, the entity tables each have a unique integer ID field; for example, the LID field maps to the objectID attribute of the League objects. A resolution table holds the registration data. A row in this table identifies that a specific player has registered for a specific division within a league. Finally, the database keeps track of the next ID number for each table in the ObjectIDs table. This schemata is shown in Figure 12-2.



**Figure 12-2** Soccer League Schemata Design

Example data for each of these tables is shown in Figure 12-3.

**League**

LID	year	season	title
001	2001	Spring	Soccer League (Spring `01)
002	2001	Summer	Summer Soccer Fest 2001
003	2001	Fall	Fall Soccer League 2001
004	2004	Summer	The Summer of Soccer Love

**Registration**

LID	PID	division
001	047	Amateur
001	048	Amateur
002	048	Semi-Pro
002	049	Professional
003	048	Professional

**Player**

PID	name	address	city	province	postal_code
047	Steve Sterling	12 Grove Park Road	Manchester	Manchester	M4 6NF
048	Alice Hornblower	62 Woodside Lane	Reading	Berks	RG31 9TT
049	Wally Winkle	17 Chippenham Road	London	London	SW19 4FT

**ObjectIDs**

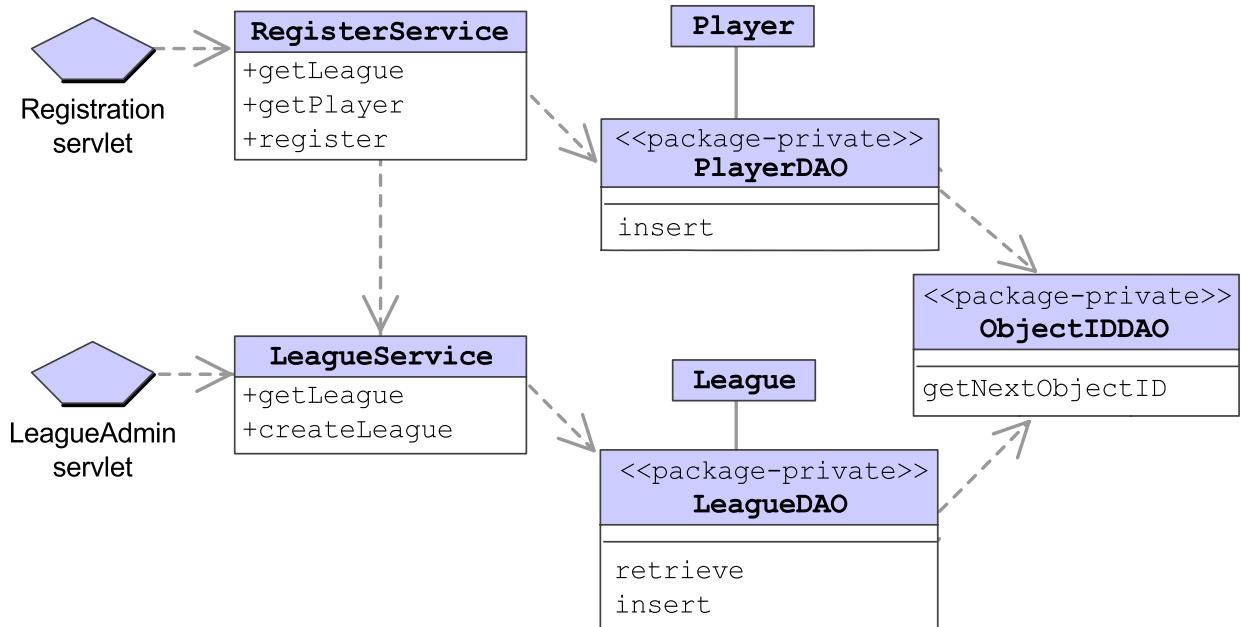
table_name	ID_number
League	005
Player	050

**Figure 12-3** Example Table Data for Soccer League

## Data Access Object Pattern

The DAO pattern makes it easier to maintain applications that use databases by separating the business logic from the data access (data storage) logic. The data access implementation (usually JDBC technology calls) is encapsulated in DAO classes. The DAO pattern permits the business logic and the data access logic to change independently. For example, if the DB schema changes, you would only need to change the DAO methods and not the business services or the domain objects.

The Model tier now includes the business services classes, domain object classes, and the DAO classes. The Web tier interacts directly with the business services and the domain objects. The DAO classes are hidden from the Web tier by making the DAO classes “package private.” The business services use the DAOs to perform the data access logic, such as inserting an object or retrieving an object. This is illustrated in the Soccer League Web application in Figure 12-4 on page 12-8.



**Figure 12-4** The DAO Pattern Applied to the Soccer League Model

When a registration request is processed, the Registration servlet uses the RegisterService object to perform the register business method. The register method uses the PlayerDAO object to insert the player object into the database, along with the other registration data. The PlayerDAO uses the ObjectIDDAO object to allocate the next player ID; this ID is used for the PID field when the player is inserted.



**Note** – The domain objects, like League, only hold data; that is, they only have accessor methods. These are often called Value Objects. Some domain objects require mutator methods; these are called Updateable Value Objects. For example, Player class represents an mutable domain object.

## Advantages of the DAO Pattern

The DAO pattern has the following advantages:

- Business logic and data access logic are now separate.  
The business services do not need to know how the data is stored.
- The data access objects promote reuse and flexibility in changing the system.

New business services can be constructed that reuse the data access logic in the DAO classes.

- Developers writing other servlets can reuse the same data access code.

Many Web applications have been written with the JDBC technology code embedded directly in the servlet classes. To create new Web tier functionality, the development team is forced to cut-and-paste the JDBC technology code from existing servlets into new servlets. This approach is error prone and it does not promote code reuse. The DAO pattern emphasizes code reuse by encapsulating the data access logic in one location, the DAO classes.

- This design makes it easy to change front-end technologies.

Using the DAO pattern, the Web tier components can be changed easily without impacting the existing data access logic.

- This design makes it easy to change back-end technologies.

Using the DAO pattern, the data resource tier can change independently of the front-end tiers. For example, you could change the DAO classes to integrate with an XML data source, Enterprise Information System (EIS) data source, or to some proprietary data source. This change would have no effect on the front-end tiers.

## Developing a Web Application That Uses a Connection Pool

The DAO pattern hides the details of the data access logic from the rest of the system. Typically, the DAOs use JDBC technology code to access data in a DBMS. To do this, the DAO classes require connections to perform their work. The DAO classes must acquire the connection objects independently from the rest of the system. In this section, you will see how to use a connection pool (CP) to allocate connections for the DAO classes.

To develop a Web application that uses a CP with the DAO pattern, consider the following tasks:

- Build or buy a CP subsystem
- Store the CP object in a global “name space”
- Design your DAOs to use the CP, retrieving it from the name space
- Develop a servlet context listener to initialize the CP and store it in the name space

## Connection Pool

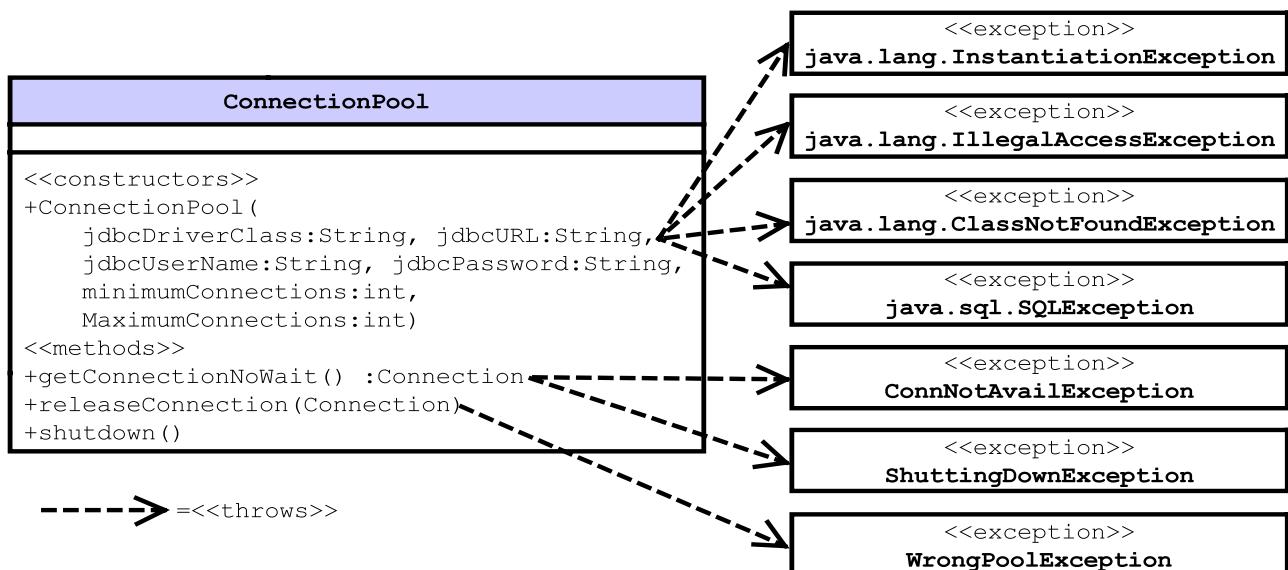
A CP is a subsystem that manages a collection of connection objects. Connection pool subsystems are often commercially available as freeware or through a JDBC technology vendor. You should consider acquiring a connection pool subsystem rather than building your own.

The CP subsystem typically has a method to retrieve a connection object and a method to release the connection back into the pool. There are design trade-offs involved in these methods, and some CP implementations support multiple methods to retrieve a connection. These are important considerations in evaluating a CP implementation for your application:

- The pool should have a minimum and maximum number of connections. At initialization time, the CP constructs the minimum number of connections before the Web application begins to process HTTP requests. As the load on the server increases, the CP might need to create new connections up to its specified maximum.

- If a connection is requested and the pool has allocated all existing connections but has not reached the maximum, then the CP constructs a new connection and adds it to the pool.
- When the maximum number of connections have already been allocated, the CP handles requests in one of two ways. It can wait until a connection becomes available or it can produce an exception to inform the caller that the system has no available connections at present. For Web application, it is often preferable to receive an exception so that the Web user does not have wait (possibly, indefinitely) for a database connection. The servlet can catch this exception and generate a “server too busy, please come by later” message.

An implementation of a CP is provided with this course. The fully qualified class name of the CP implementation is `s1314.util.sql.ConnectionPool`. The two main methods that are used in the examples for this course are: `getConnectionNoWait` and `releaseConnection`. The `getConnectionNoWait` method retrieves a `Connection` object but throws a `ConnNotAvailException` if no more connections are available within the pool. The `releaseConnection` method releases the connection back to the pool. This API is illustrated in Figure 12-5.



**Figure 12-5** An Example Connection Pool Design

## Storing the Connection Pool in a Global Name Space

To be used by your DAO classes, a single connection pool object must be globally accessible to the Web application. There are many strategies for creating a global name space. For the Soccer League Web application, you use a simple hand-coded `NamingService` class that implements the Singleton pattern. This Singleton pattern allows you to create named attributes in a manner similar to the `ServletContext` class. This API is shown in Figure 12-6.



**Figure 12-6** The `NamingService` API

**Note** – The Singleton design pattern ensures that a class has only one instance and that the instance is globally accessible. This is achieved by making the constructor of the Singleton class private and by providing a static method to retrieve a single instance, the `getInstance` method.



## Accessing the Connection Pool

The DAO classes can now retrieve the connection pool from the NamingService object. For example, part of the retrieve method in the LeagueDAO class is shown in Code 12-1. The connection pool is retrieved from the naming service (Lines 34–36) and the connection object is retrieved from the CP (Line 49).

**Code 12-1** Retrieving the Connection Pool in a DAO Class

```
27  /**
28   * This method retrieves a League object from the database.
29   */
30  League retrieve(String year, String season)
31      throws ObjectNotFoundException {
32
33      // Retrieve the connection pool from the global Naming Service
34      NamingService nameSvc = NamingService.getInstance();
35      ConnectionPool connectionPool
36          = (ConnectionPool) nameSvc.getAttribute("connectionPool");
37
38      // Database variables
39      Connection connection = null;
40      PreparedStatement stmt = null;
41      ResultSet results = null;
42      int num_of_rows = 0;
43
44      // Domain variables
45      League league = null;
46
47      try {
48          // Get a database connection
49          connection = connectionPool.getConnectionNoWait();
```

The data access logic for retrieving a League object is shown in Code 12-2. The Connection object is used to create a prepared SQL statement (Line 52). This statement is used to retrieve a row from the League table (Line 57) that matches the year and season parameters passed to the DAO retrieve method. The year and season form a secondary key into the League table, meaning that only one row in the table should match these two values. The while statement (Lines 60–74) constructs a League object from the data in the result set (Lines 70–73). The code also checks that one and only one row is returned (Lines 65–67). Finally, if no row was retrieved, then an ObjectNotFoundException is thrown (Lines 76–80).

### Code 12-2 Data Access Logic for Retrieving a League Object

```
51 // Create SQL SELECT statement
52 stmt = connection.prepareStatement(RETRIEVE_STMT);
53
54 // Initialize statement and execute the query
55 stmt.setString(1, year);
56 stmt.setString(2, season);
57 results = stmt.executeQuery();
58
59 // Iterator over the query results
60 while ( results.next() ) {
61     int objectID = results.getInt("LID");
62
63     // We expect only one row to be returned
64     num_of_rows++;
65     if ( num_of_rows > 1 ) {
66         throw new SQLException("Too many rows were returned.");
67     }
68
69     // Create and fill-in the League object
70     league = new League(objectID,
71                         results.getString("year"),
72                         results.getString("season"),
73                         results.getString("title"));
74 }
75
76 if ( league != null ) {
77     return league;
78 } else {
79     throw new ObjectNotFoundException();
80 }
81
```

The exception handling logic and the code to release the Connection object back to the connection pool is shown in Code 12-3. The DAO retrieve method must catch and handle any exceptions thrown in the try block (Lines 47–81). The finally clause is used to close the result set and statement objects (Lines 96–103) and to release the connection object back the connection pool (Lines 104–107). The RETRIEVE\_STMT is a private constant that holds the SQL prepared statement string (Lines 115 and 116).

### Code 12-3 Releasing the Connection to the Pool

```

82     // Handle any SQL errors
83     } catch (SQLException se) {
84         throw new RuntimeException("A database error occurred. " +
85         se.getMessage());
86
87         // Handle no available connection
88     } catch (ConnNotAvailException cnae) {
89         throw new RuntimeException("The server is busy, try again.");
90
91         // Handle server shutting down
92     } catch (ShuttingDownException sde) {
93         throw new RuntimeException("The server is being shutdown.");
94
95         // Clean up JDBC resources
96     } finally {
97         if ( results != null ) {
98             try { results.close(); }
99             catch (SQLException se) { se.printStackTrace(System.err); }
100        }
101        if ( stmt != null ) {
102            try { stmt.close(); }
103            catch (SQLException se) { se.printStackTrace(System.err); }
104        }
105        if ( connection != null ) {
106            try { connectionPool.releaseConnection(connection); }
107            catch (Exception e) { e.printStackTrace(System.err); }
108        }
109    }
110
111 /**
112 * The SQL query for a prepared statement to retrieve a League
113 * by the season and year fields.
114 */
115 private static final String RETRIEVE_STMT
116     = "SELECT * FROM League WHERE year=? AND season=?";

```

## Initializing the Connection Pool

The connection pool must be created once. The best time to initialize the connection pool is when the Web application starts. Use a servlet context listener to perform the initialization of the CP at startup and to shut down the CP when the Web application is shut down. In the Soccer League example, the `InitializeConnectionPool` class implements a `ServletContextListener` interface. Just part of the `initializeContext` method is shown in Code 12-4.

**Code 12-4** Initializing the Connection Pool

```
70 NamingService nameSvc = NamingService.getInstance();
71 connectionPool = new ConnectionPool(jdbcDriver, jdbcURL,
72                                     jdbcUserName, jdbcPassword,
73                                     minimumConnections,
74                                     maximumConnections);
75 nameSvc.setAttribute("connectionPool", connectionPool);
76 context.log("Connection pool created for URL=" + jdbcURL);
```

When the Web application shuts down, the connections in the pool should be released. This can be accomplished in the `destroyContext` method. This is shown in Code 12-5.

**Code 12-5** Shutting Down the Connection Pool

```
101 NamingService nameSvc = NamingService.getInstance();
102 ConnectionPool connectionPool
103     = (ConnectionPool) nameSvc.getAttribute("connectionPool");
104
105 // Shutdown the connection pool
106 connectionPool.shutdown();
107 context.log("Connection pool shutdown.");
```

## Developing a Web Application That Uses a Data Source

If you are developing your Web application in a J2EE compliant environment, then you should use a `DataSource` object instead of a connection pool. The `DataSource` interface is defined in the `javax.sql` extension package and is not part of the standard `java.sql` package. A data source acts like a connection pool, but connection objects can be closed rather than released back to the CP. You can usually buy a data source subsystem from your DBMS vendor or your J2EE platform vendor.

The `DataSource` object is stored using a JNDI service implementation. This is handled by the J2EE technology deployment environment. Your DAOs use a JNDI context to lookup the data source. The DAO uses the `DataSource` object to retrieve a `Connection` object. The DAO closes the connection when it has completed its work.

---

**Note** – The JNDI lookups are more expensive than looking up the CP object in the naming service used in the module's example. However, JNDI is a much more generic solution.

---



## Summary

This module presented how to integrate the Web tier with a database tier by using the Data Access Object pattern to separate the presentation and business logic from the data access logic. Here are the key ideas:

- A DBMS is often used to implement a robust persistence mechanism for large-scale applications.
- The JDBC API is used for interacting with a DBMS.
- The DAO pattern permits the separation of the business logic (and domain objects) from the data access logic.
- A connection pool is used to manage a pool of reusable Connection objects.
- A data source might also be used to manage a pool of reusable Connection objects. Data sources are part of the J2EE platform.

## Certification Exam Notes

This module presented some of the objectives for Section 13, "Web Tier Design Patterns," of the Sun Certification Web Component Developer certification exam:

- 13.1 Given a scenario description with a list of issues, select the design pattern (Value Objects, MVC, *Data Access Object*, or Business Delegate) that would best solve those issues.
- 13.2 Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Value Objects, MVC, *Data Access Object*, and Business Delegate.

The MVC pattern is presented in Module 15, "Developing Web Applications Using the Model 2 Architecture," and the Business Delegate pattern is presented in Module 20, "Integrating Web Applications With Enterprise JavaBeans Components."



# Developing JSP™ Pages

---

## Objectives

Upon completion of this module, you should be able to:

- Describe JavaServer Page (JSP) technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- Create and use JSP error pages
- Describe what the Web container does behind the scenes to process a JSP page

## Relevance



**Discussion** – The following questions are relevant to understanding JSP technology:

- What problems exist in generating an HTML response in a servlet?
  
- How do template page technologies (and JSP technology in particular) solve these problems?

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

- *JavaServer Pages Specification*. [Online]. Available: <http://java.sun.com/products/jsp/>.
- Hall, Marty. *Core Servlets and JavaServer Pages*. Upper Saddle River: Prentice Hall PTR, 2000

# JavaServer Page Technology

JavaServer Pages enable you to write standard HTML pages containing tags that run powerful programs based on the Java programming language. The goal of JSP technology is to support separation of presentation and business logic:

- Web designers can design and update pages without learning the Java programming language.
- Java technology programmers can write code without dealing with Web page design.

To give a first glimpse at JSP technology, this section compares the Hello World servlet with an equivalent JSP page. This version of the Hello World servlet retrieves the name of the user, which becomes part of the dynamic response. This code is shown in Code 13-1.

**Code 13-1**      Hello World Servlet

```
27 public void generateResponse(HttpServletRequest request,
28                               HttpServletResponse response)
29     throws IOException {
30
31     // Determine the specified name (or use default)
32     String name = request.getParameter("name");
33     if ( (name == null) || (name.length() == 0) ) {
34         name = DEFAULT_NAME;
35     }
36
37     // Specify the content type is HTML
38     response.setContentType("text/html");
39     PrintWriter out = response.getWriter();
40
41     // Generate the HTML response
42     out.println("<HTML>");
43     out.println("<HEAD>");
44     out.println("<TITLE>Hello Servlet</TITLE>");
45     out.println("</HEAD>");
46     out.println("<BODY BGCOLOR='white'>");
47     out.println("<B>Hello, " + name + "</B>");
48     out.println("</BODY>");
49     out.println("</HTML>");
50
51     out.close();
52 }
```

The equivalent JSP page is shown in Code 13-2.

**Code 13-2**    The hello.jsp Page

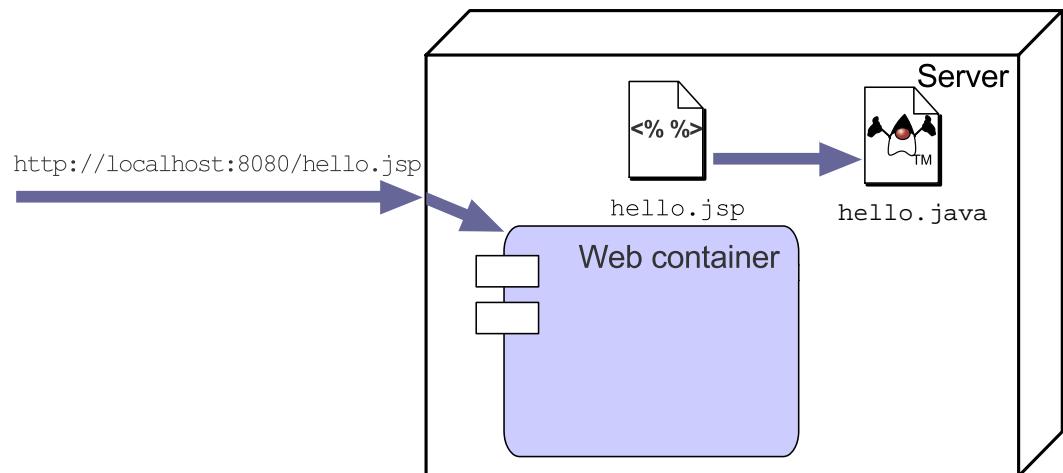
```
1  <%! private static final String DEFAULT_NAME = "World"; %>
2
3  <HTML>
4
5  <HEAD>
6  <TITLE>Hello JavaServer Page</TITLE>
7  </HEAD>
8
9  <%-- Determine the specified name (or use default) --%>
10 <%
11     String name = request.getParameter("name");
12     if ( (name == null) || (name.length() == 0) ) {
13         name = DEFAULT_NAME;
14     }
15 >
16
17 <BODY BGCOLOR='white'>
18
19 <B>Hello, <%= name %></B>
20
21 </BODY>
22
23 </HTML>
```

Most of the JSP page is HTML template text. The `<%` and `%>` tags on Lines 10 and 15 tell the Web container to execute the embedded Java technology code at runtime. Similarly, the `<%= name %>` element on Line 19 tells the Web container to place the string value of the name variable into the HTTP response at runtime.

In this module, JSP scripting elements and how the Web container processes a JSP page are described.

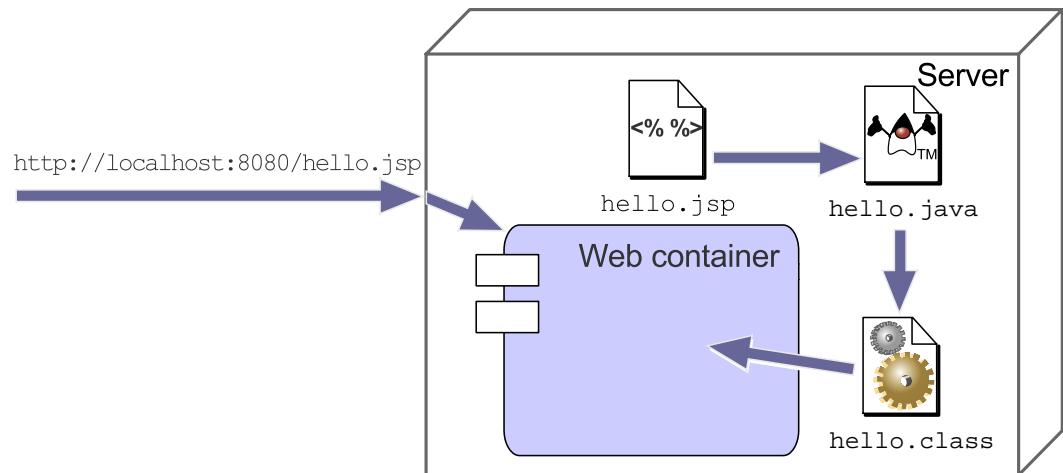
## How a JSP Page Is Processed

The first time a JSP page is requested, the Web container converts the JSP file into a servlet that can respond to the HTTP request. In the first step, the Web container translates the JSP file into a Java source file that contains a servlet class definition. This is illustrated in Figure 13-1.



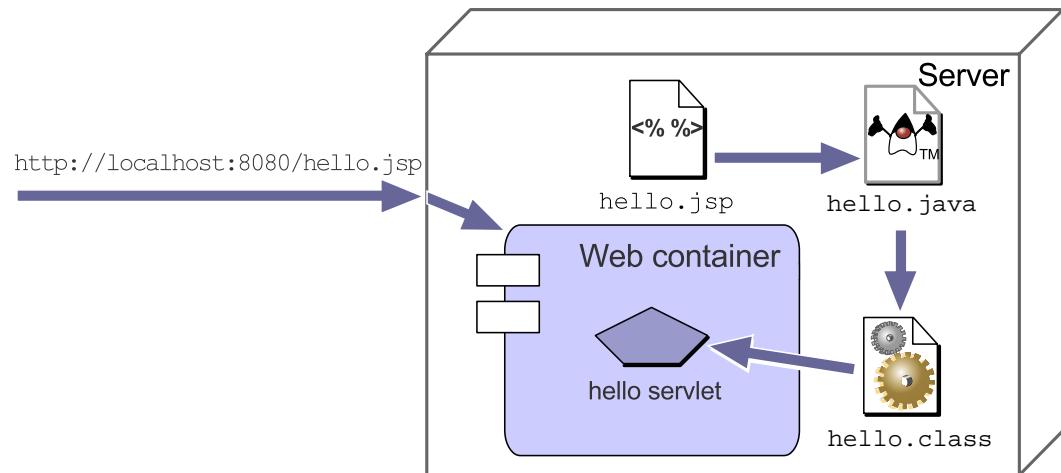
**Figure 13-1** JSP Processing: Translation Step

In the second step, the Web container compiles the servlet source code into a Java class file. This servlet class bytecode is then loaded into the Web container's JVM using a classloader. This is illustrated in Figure 13-2.



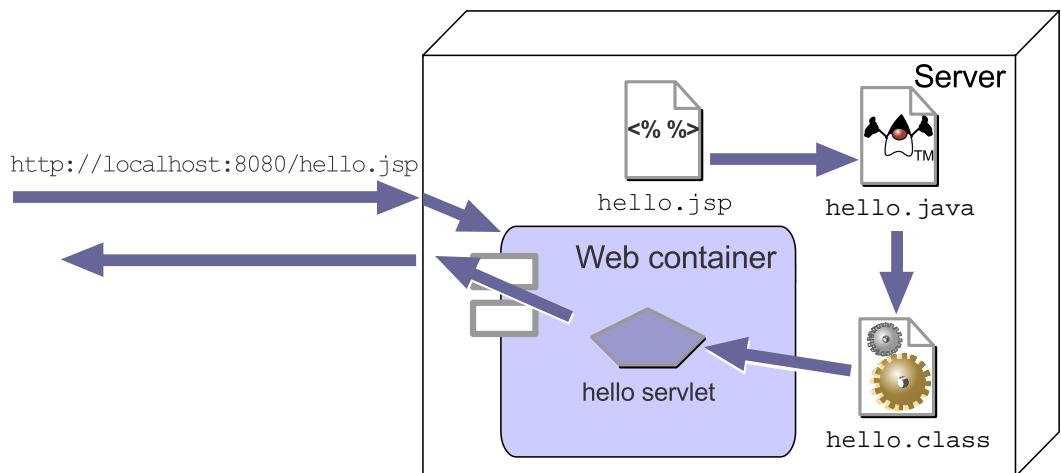
**Figure 13-2** JSP Processing: Compilation Step

In the third step, the Web container creates an instance of the servlet class and performs the initialization life cycle step by calling the special `jspInit` method. This is illustrated in Figure 13-3.



**Figure 13-3** JSP Processing: Initialization Step

Finally, the Web container can call the `_jspService` method for the converted JSP page so that it can respond to client HTTP requests. This is illustrated in Figure 13-4.



**Figure 13-4** JSP Processing: Service Step

When the Web container wants to remove the JSP servlet instance, it must call the `jspDestroy` method to allow the JSP page to perform any clean up it requires.



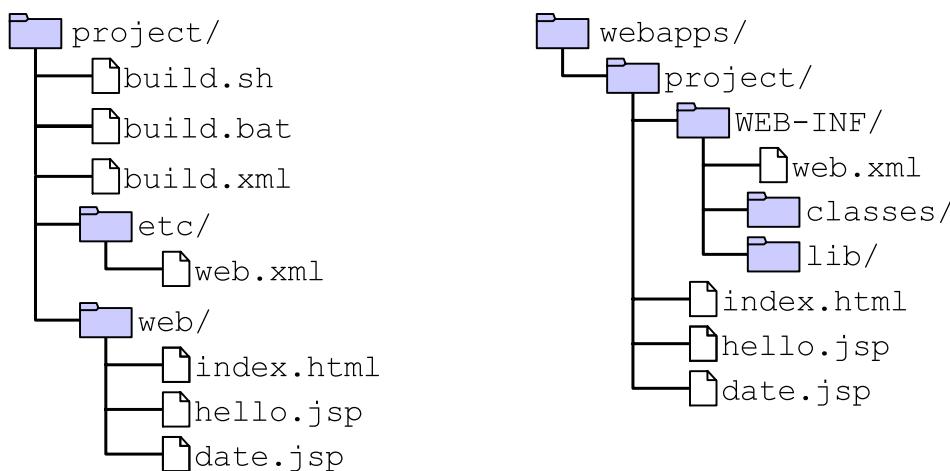

---

**Note** – The process of translating, compiling, loading, and initializing is performed every time the JSP file changes within the Web container's deployment environment. This makes developing JSP pages easier.

---

## Developing and Deploying JSP Pages

Unlike servlets, deploying JSP pages is as easy as deploying static pages. JSP pages are placed in the same directory hierarchy as HTML pages. In the development environment, JSP pages are in the `web` directory. In the deployment environment, JSP pages are placed at the top-level directory of the Web application. This is illustrated in Figure 13-5.



**Figure 13-5** JSP Pages in the Development and Deployment Hierarchies




---

**Note** – Just as HTML pages can exist in a complex directory hierarchy, JSP pages can also exist at any level in the Web application's hierarchy.

---

## JSP Scripting Elements

JSP scripting elements are embedded with the `<% %>` tags and are processed by the JSP engine during translation of the JSP page. Any other text in the JSP page is considered part of the response and is copied verbatim to the HTTP response stream that is sent to the Web browser.

```
<HTML>
<%-- scripting element --%>
</HTML>
```

There are five types of scripting elements:

- Comments                           `<%-- comment --%>`
- Directive tag                      `<%@ directive %>`
- Declaration tag                 `<%! decl %>`
- Scriptlet tag                     `<% code %>`
- Expression tag                   `<%= expr %>`

## Comments

Documentation is important to any application. There are three types of comments permitted in a JSP page:

- HTML comments

HTML comments are considered HTML template text. These comments are sent in the HTTP response stream. For example:

```
<!-- This is an HTML comment. It will show up in the response. -->
```

- JSP page comments

JSP page comments are only seen in the JSP page file itself. These comments are not included in the servlet source code during the translation phase, nor do they appear in the HTTP response. For example:

```
<%-- This is a JSP comment. It will only be seen in the JSP code.  
     It will not show up in either the servlet code or the response.  
--%>
```

- Java comments

Java comments can be embedded with scriptlet and declaration tags. These comments are included in the servlet source code during the translation phase, but do not appear in the HTTP response. For example:

```
<%  
/* This is a Java comment. It will show up in the servlet code.  
     It will not show up in the response. */  
%>
```

---

**Note** – You can also use Javadoc™ comments in declaration tags, for example `/** @author Jane Doe */`.

---



## Directive Tag

A directive tag provides information that will affect the overall translation of the JSP page. The syntax for a directive tag is:

```
<%@ DirectiveName [attr="value"]* %>
```

Three types of directives are currently specified in the JSP specification: page, include, and taglib. Here are a couple of examples:

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

```
<%@ include file="incl/copyright.html" %>
```

The page directive is described in detail later in this module. The include directive is described in Module 16, “Building Reusable Web Presentation Components,” and the taglib directive is described in Module 17, “Developing JSP Pages Using Custom Tags.”

## Declaration Tag

A declaration tag allows you to include members in the JSP servlet class, either attributes or methods. The syntax for a declaration tag is:

```
<%! JavaClassDeclaration %>
```

Here are a couple of examples:

```
<%! public static final String DEFAULT_NAME = "World"; %>

<%! public String getName(HttpServletRequest request) {
    return request.getParameter("name");
}
%>

<%! int counter = 0; %>
```

You can think of one JSP page as being equivalent to one servlet class. The declaration tags become declarations in the servlet class. You can create any type of declaration that is permissible in a regular Java technology class: instance variables, instance methods, class (or static) variables, class methods, inner classes, and so on. Be careful with using instance and class variables due to concurrency issues. A JSP page is multithreaded like a servlet.

You can use a declaration tag to override the `jspInit` and `jspDestory` life cycle methods. The signature of these methods has no arguments and returns `void`. For example:

```
<%!
public void jspInit() {
    /* initialization code here */
}

public void jspDestroy() {
    /* clean up code here */
}
%>
```

---

**Note** – It is illegal to override the `_jspService` method which is created by the JSP engine during translation.



## Scriptlet Tag

A scriptlet tag allows the JSP page developer to include arbitrary Java technology code in the `_jspService` method. The syntax for a declaration tag is:

```
<% JavaCode %>
```

Here are a couple of examples:

```
<% int i = 0; %>
```

In this example, the local variable `i` is declared with an initial value of 0.

```
<% if ( i > 10 ) { %>
I is a big number.
<% } else { %>
I is a small number
<% } %>
```

In this example, a conditional statement is used to select either the statement “I is a big number” or “I is a small number” to be sent to the HTTP response at runtime. For readability, you might want to exaggerate the scriptlet tags from the template text. For example:

```
<%
    if ( i > 10 ) {
%>
I is a big number.
<%
    } else {
%>
I is a small number
<%
    }
%>
```

Below is an iteration example. This JSP code creates a table in which the first column contains the numbers from 0 to 9 and the second column contains the squares of 0 through 9.

```
<TABLE BORDER='1' CELLSPACING='0' CELLPADDING='5'>
<TR><TH>number</TH><TH>squared</TH></TR>
<% for ( int i=0; i<10; i++ ) { %>
<TR><TD><%= i %></TD><TD><%= (i * i) %></TD></TR>
<% } %>
</TABLE>
```

## Expression Tag

An expression tag holds a Java language expression that is evaluated during an HTTP request. The result of the expression is included in the HTTP response stream. The syntax for a declaration tag is:

```
<%= JavaExpression %>
```

An expression is any syntactic construct that evaluates to either a primitive value (an int, float, boolean, and so on) or to an object reference. The value of the expression is converted into a string; this string is included in the HTTP response stream.

---

**Note** – A good rule of thumb is that an expression tag can hold any Java language expression that can be used as an argument to the `System.out.print` method.

---



Here are some examples:

```
<B>Ten is <%= (2 * 5) %></B>
```

This example shows an arithmetic expression. When this is evaluated, the number 10 is the result. The string “`<B>Ten is 10</B>`” is sent back in the HTTP response stream.

Thank you, `<I><%= name %></I>`, for registering for the soccer league.

This example shows that you can access local variables declared in the JSP page. If the name variable holds a reference to a String object, then that string is sent back in the HTTP response stream.

The current day and time is: `<%= new java.util.Date() %>`

This example shows that you can use an object in an expression tag. In this example, the Date object’s `toString` method is called; the string value returned is included in the HTTP response stream. All Java technology classes inherit or override the `toString` method. The JSP page uses this method to calculate the string representation of an object, which is included in the HTTP response stream.

## Implicit Variables

The Web container gives the JSP technology developer access to the following variables in scriptlet and expression tags. These variables represent commonly used objects for servlets that JSP developers might need to use. For example, you can retrieve HTML form parameter data by using the `request` variable, which represents the `HttpServletRequest` object. All implicit variables are shown in Table 13-1.

**Table 13-1** Implicit Variables in JSP Pages

Variable name	Description
<code>request</code>	The <code>HttpServletRequest</code> object associated with the request.
<code>response</code>	The <code>HttpServletResponse</code> object associated with the response that is sent back to the browser.
<code>out</code>	The <code>JspWriter</code> object associated with the output stream of the response.
<code>session</code>	The <code>HttpSession</code> object associated with the session for the given user of the request. This variable is only meaningful if the JSP page is participating in an HTTP session.
<code>application</code>	The <code>ServletContext</code> object for the Web application.
<code>config</code>	The <code>ServletConfig</code> object associated with the servlet for this JSP page.
<code>pageContext</code>	This object encapsulates the environment of a single request for this JSP page.
<code>page</code>	This variable is equivalent to the <code>this</code> variable in the Java programming language.
<code>exception</code>	The <code>Throwable</code> object that was thrown by some other JSP page. This variable is only available in a “JSP error page.”

The `pageContext`, `page`, and `exception` implicit variables are not commonly used. Thread-safety should be considered when accessing attributes in the `session` and `application` variables.

---

**Note** – Another name for “implicit variable” is “implicit object.”

---



## The page Directive

The page directive is used to modify the overall translation of the JSP page. For example, you can declare that the servlet code generated from a JSP page requires the use of the Date class:

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

You can have more than one page directive, but can only declare any given attribute once per page, except for the `import` attribute. You can place a page directive anywhere in the JSP file. It is good practice to make the page directive the first statement in the JSP file.

The page directive defines a number of page-dependent properties and communicates these to the Web container during translation time.

- The `language` attribute defines the scripting language to be used in the page. The value "java" is the only value currently defined and is the default.
- The `extends` attribute defines the (fully-qualified) class name of the superclass of the servlet class that is generated from this JSP page.

---

**Caution –** *Do not* use the `extends` attribute. Changing the superclass of your JSP pages might make your Web application non-portable.



- The `import` attribute defines the set of classes and packages that must be imported in the servlet class definition. The value of this attribute is a comma-delimited list of fully-qualified class names or packages. For example:  
`import="java.sql.Date, java.util.* , java.text.*"`
- The `session` attribute defines whether the JSP page is participating in an HTTP session. The value is either `true` (the default) or `false`.
- The `buffer` attribute defines the size of the buffer used in the output stream (a `JspWriter` object). The value is either `none` or `Nkb`. The default buffer size is 8 kilobytes (KBytes) or greater. For example: `buffer="8kb"` or `buffer="none"`.
- The `autoFlush` attribute defines whether the buffer output should be flushed automatically when the buffer is filled or whether an exception is thrown. The value is either `true` (automatically flush) or `false` (throw an exception). The default is `true`.

- The `isThreadSafe` attribute allows the JSP page developer to declare that the JSP page is thread-safe or not. If the value is set to `false`, then this attribute instructs JSP parser to write the servlet code such that only one HTTP request will be processed at a time. The default value is `true`.
- The `info` attribute defines an informational string about the JSP page.
- The `errorPage` attribute indicates another JSP page that will handle all runtime exceptions thrown by this JSP page. The value is a URL that is either relative to the current Web hierarchy or relative to the context root.

For example, `errorPage="error.jsp"` (this is relative to the current hierarchy) or `errorPage="/error/formErrors.jsp"` (this is relative to the Web application's context root).

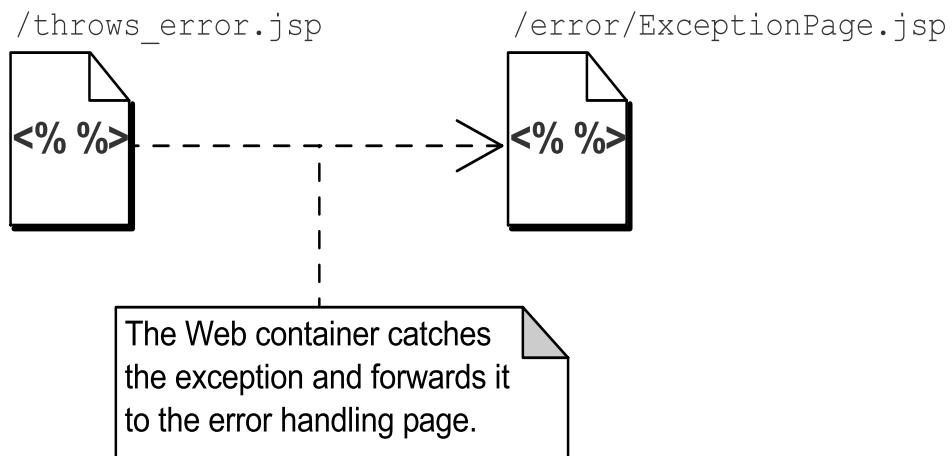
- The `isErrorPage` attribute defines that the JSP page has been designed to be the target of another JSP page's `errorPage` attribute. The value is either `true` or `false` (default). All JSP pages that "are an error page" automatically have access to the exception implicit variable.
- The `contentType` attribute defines the MIME type of the output stream. The default is `text/html`.
- The `pageEncoding` attribute defines the character encoding of the output stream. The default is ISO-8859-1. Other character encodings permit the inclusion of non-Latin character sets, such as Kanji or Cyrillic.

## JSP Page Exception Handling

Because you cannot “wrap” a try-catch block around your JSP page, the JSP specification provides a different mechanism for handling exceptions. You can specify an error page in your main JSP pages.

Usually, if a JSP page throws an exception, it is caught by the Web container and a generic “servlet exception” page is generated by the Web container. This is probably undesirable for the end user.

The page directive allows you to specify another JSP page to be activated whenever the JSP page throws an exception. This is illustrated in Figure 13-6.



**Figure 13-6** A JSP Error Page Example

There is a caveat about the use of the `errorPage` attribute: You can only specify a single error page for any given JSP page; therefore, you cannot have different error pages for different exceptions.

## Declaring an Error Page

The `errorPage` attribute of the `page` directive is used to declare the error page to be used by the Web container when *this* JSP page throws an exception. An example JSP page that throws an `ArithmaticException` is shown in Code 13-3.

**Code 13-3** The `throws_error.jsp` Page

```
1  <%@ page session="false" errorPage="error/ExceptionPage.jsp" %>
2  <%-- This page will cause an "divide by zero" exception --%>
3
4  <HTML>
5
6  <HEAD>
7  <TITLE>Demonstrate Error Pages</TITLE>
8  </HEAD>
9
10 <BODY BGCOLOR='white'>
11
12 <OL>
13 <%
14     for ( int i=10; i > -10; i-- ) {
15     %>
16     <LI><%= 100/i %>
17     <%
18     }
19     %>
20 </OL>
21
22 </BODY>
23
24 </HTML>
```

## Developing an Error Page

The `errorPage` attribute of the `page` directive is used to declare that this JSP page has access to the `exception` implicit variable. This is used by the Web container when another JSP page throws an exception. An example JSP error page is shown in Code 13-4.

**Code 13-4** The `ExceptionPage.jsp` Page

```
1  <%@ page session="false" isErrorPage="true"
2      import="java.io.PrintWriter" %>
3
4  <%-- Using the implicit variable EXCEPTION
5      extract the name of the exception --%>
6 <%
7     String expTypeFullName
8     = exception.getClass().getName();
9     String expTypeName
10    = expTypeFullName.substring(expTypeFullName.lastIndexOf(".") + 1);
11     String request_uri
12     = (String) request.getAttribute("javax.servlet.error.request_uri");
13 %>
14
15 <HTML>
16
17 <HEAD>
18 <TITLE>JSP Exception Page</TITLE>
19 </HEAD>
20
21 <BODY BGCOLOR='white'>
```

This JSP code is a lot like the `ExceptionPage` servlet discussed in Module 9, “Handling Errors in Web Applications.” The main difference is that the scriplet code uses the implicit variable `exception` to access the `exception` object thrown by the original JSP page (Line 8).

## Behind the Scenes

The servlet code produced from a translated JSP page is machine-produced and is difficult to read. An example servlet source code is shown in Code 13-5.

**Note** – The Tomcat server stores the JSP servlet source files in a directory called work.



**Code 13-5** The Translated Servlet for the hello.jsp Page

```
10 public class hello_jsp extends HttpJspBase {  
11  
12     // begin  
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(1,3);to=(1,56)]  
13     private static final String DEFAULT_NAME = "World";  
14     // end  
15  
16     static {  
17     }  
18     public hello_jsp( ) {  
19     }  
20  
21     private static boolean _jspx_initited = false;  
22  
23     public final void _jspx_init() throws org.apache.jasper.JasperException  
24     }  
25  
26     public void _jspService(HttpServletRequest request, HttpServletResponse  
27         throws java.io.IOException, ServletException {  
28  
29         JspFactory _jspxFactory = null;  
30         PageContext pageContext = null;  
31         ServletContext application = null;  
32         ServletConfig config = null;  
33         JspWriter out = null;  
34         Object page = this;  
35         String _value = null;
```

```
36     try {
37
38         if (_jspx_initited == false) {
39             synchronized (this) {
40                 if (_jspx_initited == false) {
41                     _jspx_init();
42                     _jspx_initited = true;
43                 }
44             }
45         }
46         _jspxFactory = JspFactory.getDefaultFactory();
47         response.setContentType("text/html;charset=ISO-8859-1");
48         pageContext = _jspxFactory.getPageContext(this, request, response,
49                                         "", false, 8192, true);
50
51         application = pageContext.getServletContext();
52         config = pageContext.getServletConfig();
53         out = pageContext.getOut();
54
55         // HTML // begin
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(0,27);to=(1,0)]
56             out.write("\r\n");
57
58         // end
59         // HTML // begin
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(1,58);to=(9,0)]
60             out.write("\r\n\r\n<HTML>\r\n\r\n<HEAD>\r\n<TITLE>Hello JavaServ
Page</TITLE>\r\n</HEAD>\r\n\r\n");
61
62         // end
63         // HTML // begin
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(9,55);to=(10,0)]
64             out.write("\r\n");
65
66         // end
67         // begin
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(10,2);to=(15,0)]
68
69             String name = request.getParameter("name");
70             if ( (name == null) || (name.length() == 0) ) {
71                 name = DEFAULT_NAME;
72             }
73         // end
74         // HTML // begin
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(15,2);to=(19,10)]
```

```
75         out.write("\r\n\r\n<BODY BGCOLOR='white'>\r\n\r\n<B>Hello, " );
76
77         // end
78         // begin
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(19,13);to=(19,19)]
79         out.print( name );
80         // end
81         // HTML // begin
[file="/usr/local/jakarta/tomcat/webapps/jsp/hello.jsp";from=(19,21);to=(24,0)]
82         out.write("</B>\r\n\r\n</BODY>\r\n\r\n</HTML>\r\n");
83
84         // end
85
86     } catch (Throwable t) {
87         if (out != null && out.getBufferSize() != 0)
88             out.clearBuffer();
89         if (pageContext != null) pageContext.handlePageException(t);
90     } finally {
91         if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext
92     }
93 }
94 }
```

## Debugging a JSP Page

There are basically three types of errors which can occur in JSP pages. Errors in JSP pages are usually found when you try to access them from the Web container. Therefore, it is good practice to test all of your JSP pages before deploying them. The three types of error are as follows:

- Translation time (parsing) errors

An error at translation time means that the Web container was not able to parse the scripting elements in the JSP page. For example, you might have an open directive tag (`<%!`) without a closing tag (`%>`). A screen shot of a translation error message from Tomcat is shown in Figure 13-7.

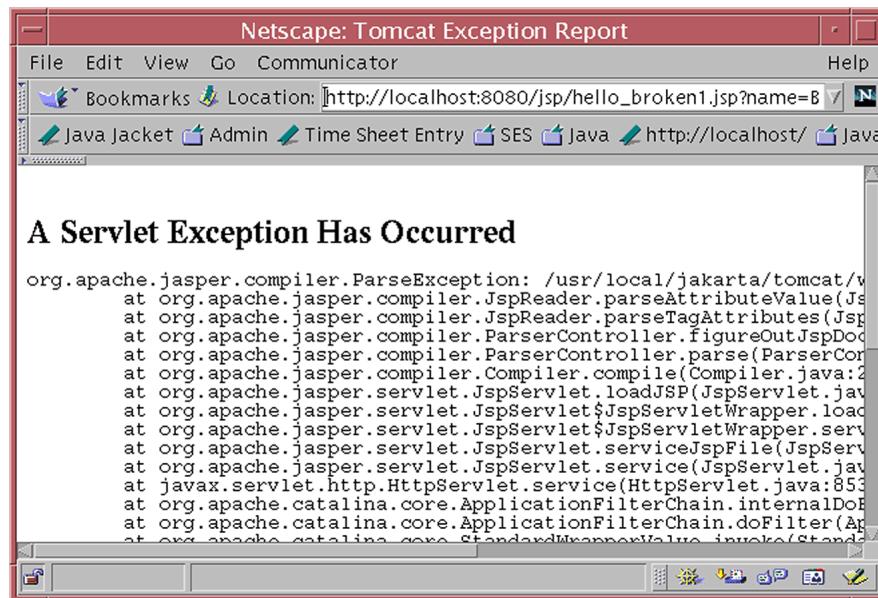
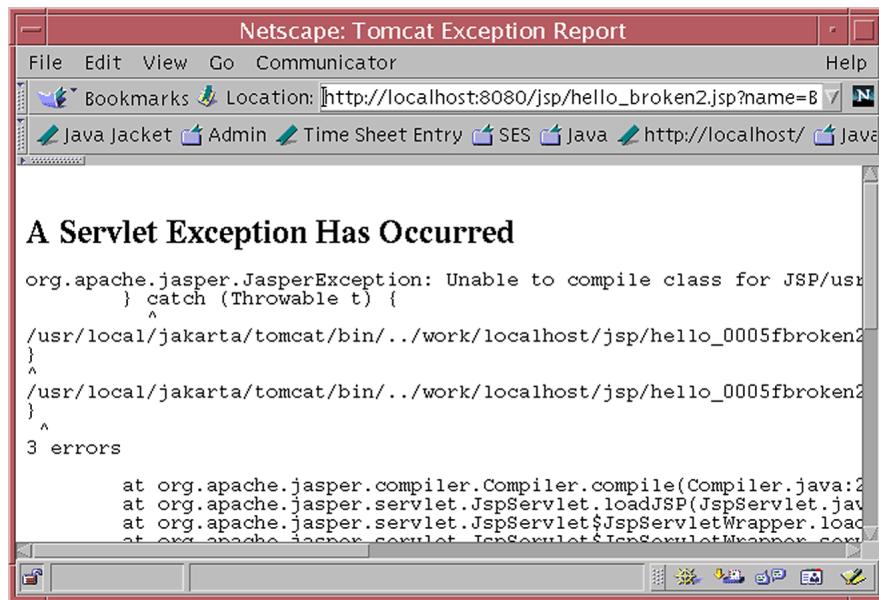


Figure 13-7 Example Translation Error

- Compilation time errors (errors in the servlet code)

An error at compilation time means that the Web container was not able to compile the Java servlet source code into Java bytecode. For example, you might use a conditional statement in scriptlet code that does not have a close curly-brace. A screen shot of a compilation error from Tomcat is shown in Figure 13-8.



**Figure 13-8** Example Compilation Error

**Note** – Be aware that the line numbers shown in these compilation error messages are relative to the generated servlet source code file, not relative to the JSP page file.



- Runtime errors (logic errors)

An error at runtime means that the dynamic code in the JSP page has a logical error which is caught at runtime. For example, you might attempt to manipulate a request parameter, but the field name in the HTML form is different than the name given to the `getParameter` method. The `getParameter` method will return null. Manipulating a null value throws a `NullPointerException`. A screen shot of a runtime error from Tomcat is shown in Figure 13-9.

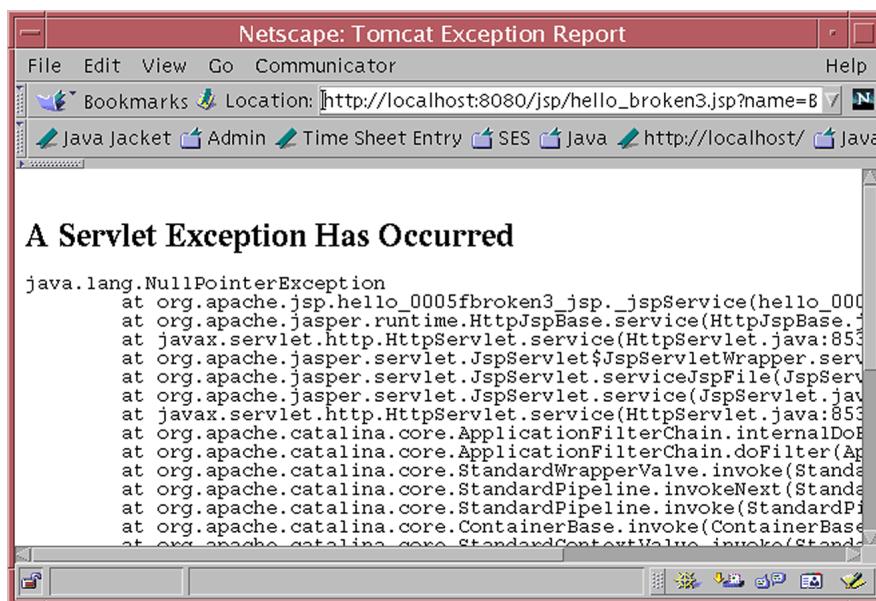


Figure 13-9 Example Runtime Error

**Note** – Be aware that the line numbers shown in these runtime error messages are relative to the generated servlet source code file, not relative to the JSP page file.



## Summary

A JSP page is like an HTML page with scripting elements that generate dynamic content. The JSP page is translated into a servlet class, which is then compiled and loaded as a servlet instance to process requests. There are five scripting elements:

- Comments <%-- *comment* --%>
- Directive tag <%@ *directive* %>
- Declaration tag <%! *decl* %>
- Scriptlet tag <% *code* %>
- Expression tag <%= *expr* %>

The page directive can be used to modify the translation of the JSP page.

# Certification Exam Notes

This module presented all of the objectives, except 8.3, for Section 8, "The JSP Model," of the Sun Certification Web Component Developer certification exam:

- 8.1 Write the opening and closing tags for the following JSP tag types: directive, declaration, scriptlet, expression.
- 8.2 Given a type of JSP tag, identify correct statements about its purpose or use.
- 8.3 Given a JSP tag type, identify the equivalent XML-based tag.
- 8.4 Identify the page directive attribute and values, that:
  - Imports a Java class into the JSP page
  - Declares that a JSP page exists within a session
  - Declares that a JSP page uses an error page
  - Declares that a JSP page is an error page
- 8.5 Identify and put in sequence the following elements of the JSP life cycle:
  - Page translation
  - JSP compilation
  - Load class
  - Create instance
  - Call `jspInit`
  - Call `_jspService`
  - Call `jspDestroy`
- 8.6 Match correct descriptions about purpose, function, or use with any of the following implicit objects: `request`, `response`, `out`, `session`, `config`, `application`, `page`, `pageContext`, `exception`.
- 8.7 Distinguish correct and incorrect scriptlet code for: a conditional statement and an iteration statement.



# Developing Web Applications Using the Model 1 Architecture

---

## Objectives

Upon completion of this module, you should be able to:

- Design a Web application using the Model 1 architecture
- Develop a Web application using the Model 1 architecture

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

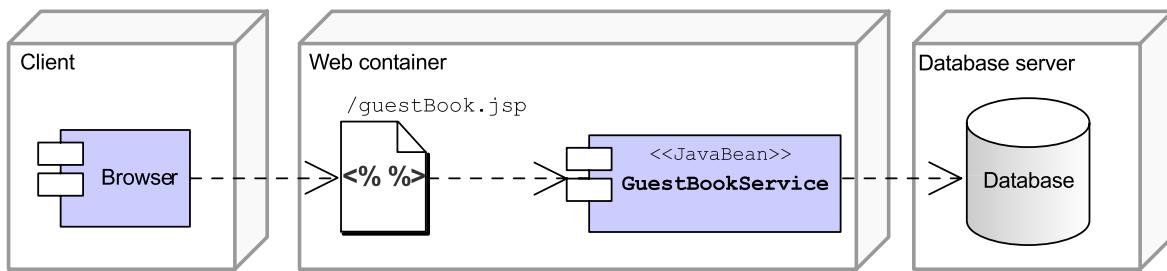
- *Java™ 2 Platform, Enterprise Edition Blueprints*. [Online]. Available: <http://java.sun.com/j2ee/blueprints/>.
- *JavaServer Pages: Building Dynamic Websystems*. [Online]. Available: <http://www.brainopolis.com/jsp/book/jspBookOutline.html>.
- Avedal, Karl, and other authors. *Professional JSP*. Birmingham, UK: WROX Press Ltd, 2000.
- *JavaServer Pages Fundamentals Short Course*. [Online] Available: <http://developer.java.sun.com/developer/onlineTraining/JSPIntro/contents.html#JSPIntro4>.

# Designing With Model 1 Architecture

The first half of this module describes how to design a Web application using the Model 1 architecture. In the second half of this module, you will see the implementation details.

In Model 1 architecture, JSP pages act as both the Views and the Controller for a Web application Use Case.

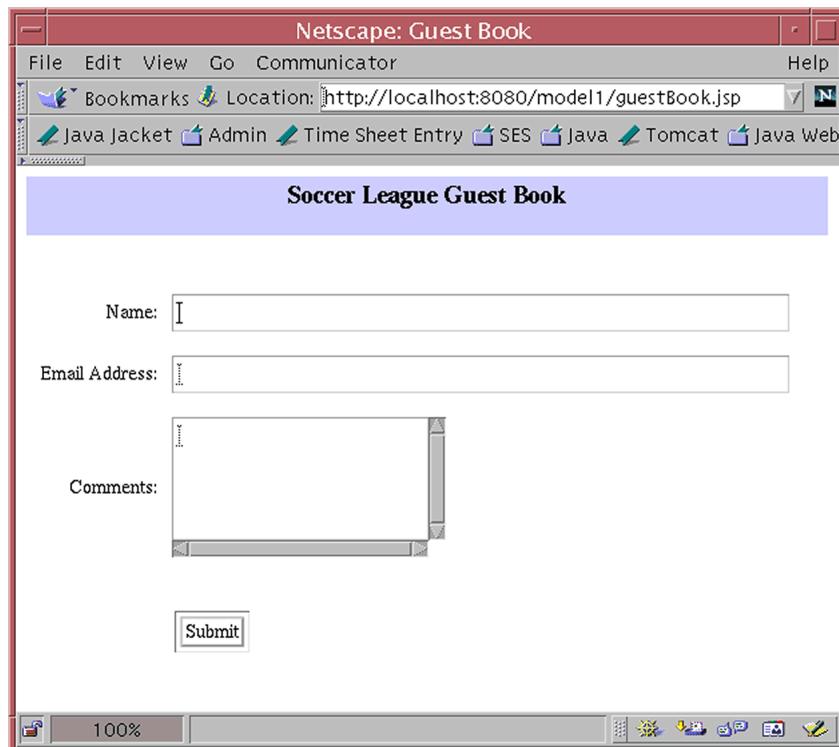
This module uses a “guest book” example, which uses a single JSP page (`guestBook.jsp`) as both the HTML form (View) and the form processor (Controller). The JSP page uses a JavaBeans component (`GuestBookService`) to perform the business and data access logic. It also handles the data verification. This architecture is illustrated in Figure 14-1.



**Figure 14-1** Guest Book Example Architecture

## Guest Book Form

The Guest Book form allows the user to enter his full name, email address, and a comment. The Web application must verify that the name field and email address field are entered. The email address field is also passed through a verification check to ensure that the format of the address matches an expected structure:  
*name@domain.top-level-domain*. A screen shot of the Guest Book form is shown in Figure 14-2.



**Figure 14-2** Screenshot of the Guest Book Form

## Guest Book Components

The Guest Book Web application is constructed from several components. The index page allows the user to access the Guest Book through a hypertext link. The guestBook.jsp page uses the GuestBookService bean to handle the business logic and verification logic. Based on the success of the form processing, the guestBook.jsp page selects either the guestBookThankYou.jsp page or the guestBookError.jsp page. The GuestBookService bean uses a few helper classes: Status object for exception handling, GuestBook to store the form data, and GuestBookDAO to handle data access logic. The relationships of these components is illustrated in Figure 14-3.

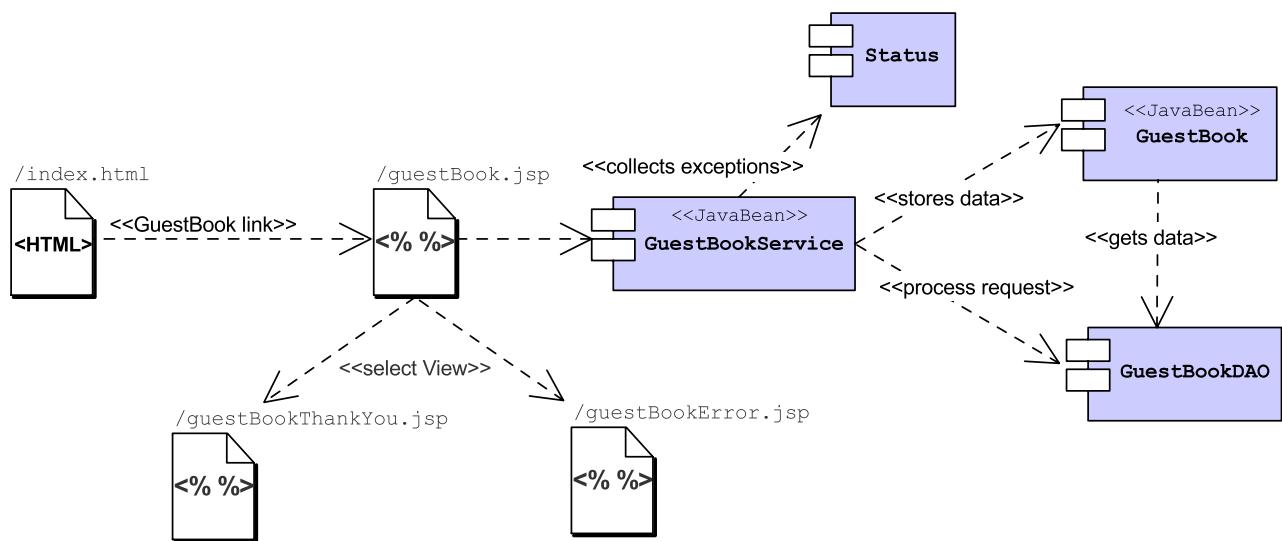
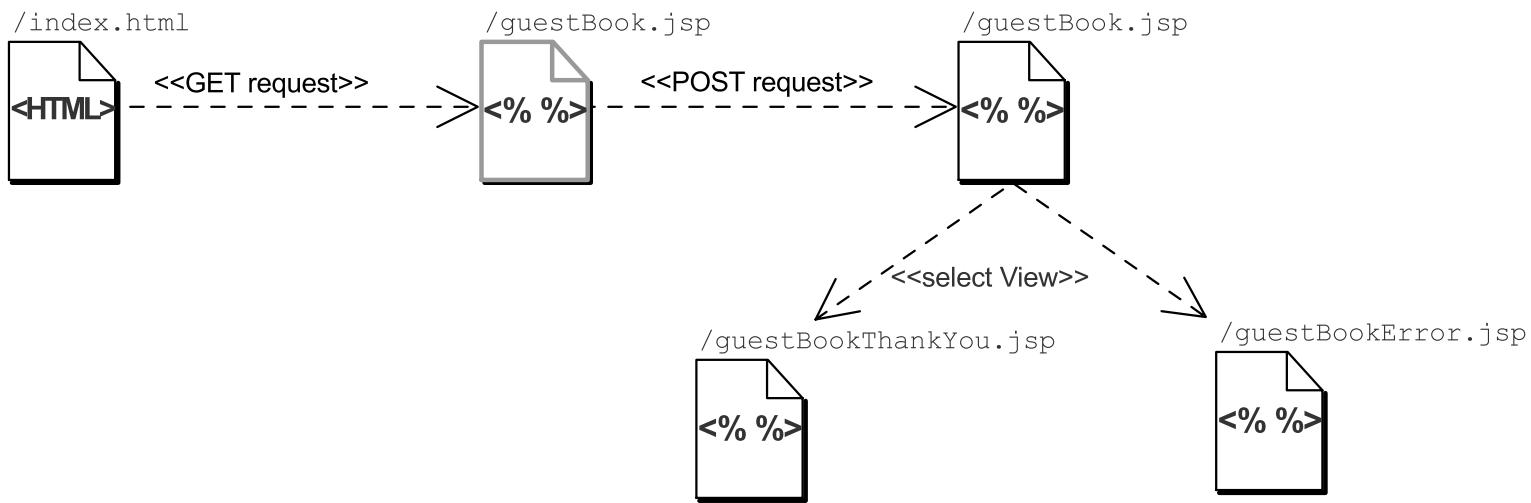


Figure 14-3 Guest Book Component Diagram

## Guest Book Page Flow

The page flow of this Web application is a little unusual because the `guestBook.jsp` page is used twice: the first time to provide the Guest Book form to fill in and the second time to process the form. The control logic in the `guestBook.jsp` page must distinguish between these two scenarios. This is accomplished by using the GET HTTP method on the first request and the POST HTTP method on the second request. During the processing of the guest book entry, the `guestBook.jsp` page will forward to either an Error page (`guestBookError.jsp`) or the Thank You page (`guestBookThankYou.jsp`). This is illustrated in Figure 14-4.



**Figure 14-4** Guest Book Page Flow

## What Is a JavaBeans Component?

A JavaBeans component is Java technology class with at least the following features:

- Properties defined with accessors and mutators (get and set methods); for example, a read-write property, `firstName`, would have `getFirstName` and `setFirstName` methods.
- A no-argument constructor.
- No public instance variables.



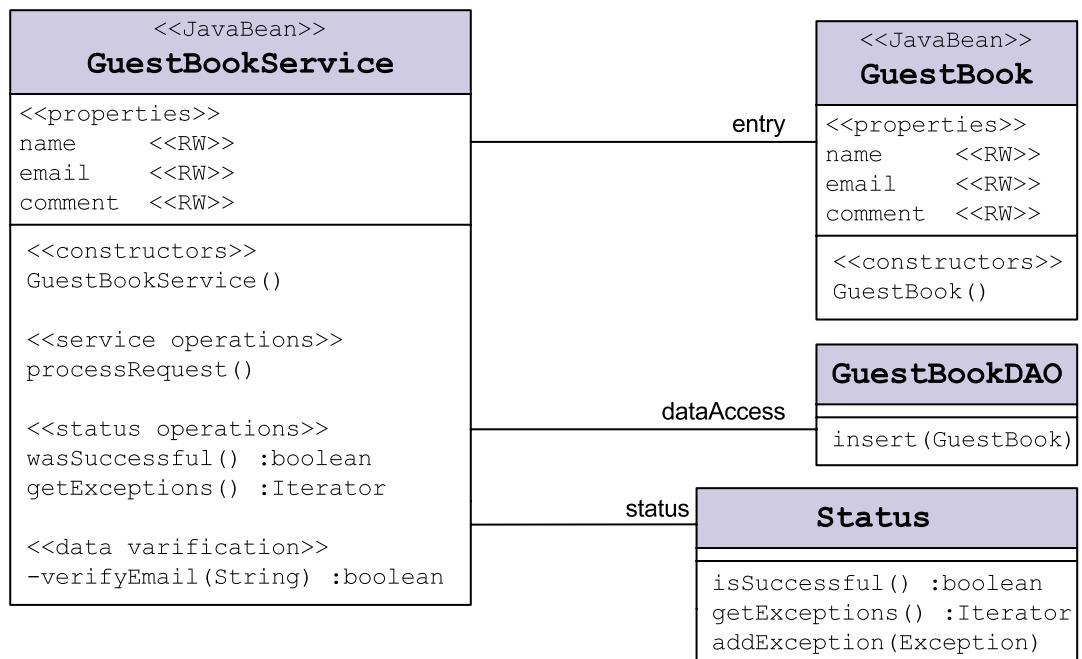
---

**Note** – JavaBeans components *are not* Enterprise JavaBeans (EJB) components. They are both based on a component-container model, but enterprise beans are used to create business logic components that use an EJB container to provide services, such as persistence, transaction management, and security control.

---

## The GuestBookService JavaBeans Component

In Model 1 architecture, the business logic is often partitioned into one or more Java technology classes. These classes often conform to the JavaBeans specification. The GuestBookService class exists to give the JSP pages a single object to work with as opposed to using the three objects that the service bean encapsulates. The GuestBook class holds the form data. The GuestBookDAO class performs the data access logic. The Status class holds the verification and processing errors. The relationship of these classes is illustrated in Figure 14-5.



**Figure 14-5** Guest Book Class Diagram

# Developing With Model 1 Architecture

In this section, you will see how the Model 1 Guest Book design is turned into code. The focus is on the `guestBook.jsp` page because it is the center of the application.

## The Guest Book HTML Form

The `guestBook.jsp` page presents the HTML form as well as controls the processing of the form (in two sequential HTTP requests). The `FORM` tag's `ACTION` attribute activates the `guestBook.jsp` page. This is how the second processing request is made using the `POST` HTTP method. The Guest Book form is shown in Code 14-1.

**Code 14-1**      The `guestBook.jsp` Page HTML Form

```
48 <FORM ACTION='guestBook.jsp' METHOD='POST'>
49
50 <TABLE BORDER='0' CELLSPACING='0' CELLPADDING='5' WIDTH='600'>
51 <TR>
52   <TD ALIGN='right'>Name:</TD>
53   <TD><INPUT TYPE='text' NAME='name' SIZE='50'></TD>
54 </TR>
55 <TR>
56   <TD ALIGN='right'>Email Address:</TD>
57   <TD><INPUT TYPE='text' NAME='email' SIZE='50'></TD>
58 </TR>
59 <TR>
60   <TD ALIGN='right'>Comments:</TD>
61   <TD><TEXTAREA NAME='comment' ROWS='5' COLUMNS='70'></TEXTAREA></TD>
62 </TR>
63 <TR HEIGHT='10'><TD HEIGHT='10' COLSPAN='2'>!-- vertical space --></TD>
64 <TR>
65   <TD></TD>
66   <TD><INPUT TYPE='submit' VALUE='Submit'></TD>
67 </TR>
68 </TABLE>
69
70 </FORM>
```

## JSP Standard Actions

*JSP standard actions* are XML-like tags used in JSP pages to perform actions at runtime. These tags are meant to reduce scripting code in JSP pages. This syntax of standard actions is identical to XML tag syntax. An empty standard action looks like:

```
<jsp:action [attr="value"]* />
```

Here are a few examples:

```
<jsp:useBean id="beanName" class="BeanClass" />
<jsp:setProperty name="beanName" property="prop1" value="val" />
<jsp:getProperty name="beanName" property="prop1" />
```

All JSP standard actions begin with the `jsp` prefix.

## Creating a JavaBeans Component in a JSP Page

The data from the form is stored in the `GuestBookService` bean. JavaBean components and regular Java technology classes can be instantiated using the `jsp:useBean` standard action. For example:

```
<jsp:useBean id="guestBookSvc" class="GuestBookService" />
```

This creates an instance of the `GuestBookService` class and stores that instance in the `guestBookSvc` attribute. This attribute is now accessible to the rest of the JSP page. You can call methods on it using other scripting tags. This is roughly equivalent to:

```
<%
    GuestBookService guestBookSvc = new GuestBookService();
%>
```

This example creates a JavaBeans component using the no-argument constructor. You can include scriptlet code to perform some initialization of the bean. For example, the guest book bean should be populated from the HTML form data. This is shown in Code 14-2.

**Code 14-2** Using `jsp:useBean` to Create a JavaBeans Component

```
3  <jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService"
scope="request">
4  <%
5      guestBookSvc.setName(request.getParameter("name"));
6      guestBookSvc.setEmail(request.getParameter("email"));
7      guestBookSvc.setComment(request.getParameter("comment"));
8  %>
9  </jsp:useBean>
```

This is roughly equivalent to:

```
<%
sl314.domain.GuestBookService guestBookSvc = new GuestBookService();
guestBookSvc.setName(request.getParameter("name"));
guestBookSvc.setEmail(request.getParameter("email"));
guestBookSvc.setComment(request.getParameter("comment"));
%>
```

## Initializing the JavaBean Component

One of the goals of using standard actions is to reduce the amount of scripting code in JSP pages. The `jsp:setProperty` action can be used to perform bean initialization. Another version of the `guestBookSvc` bean initialization is shown in Code 14-3.

**Code 14-3** Using `jsp:setProperty` to Initialize the JavaBeans Component

```
3  <jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService"
scope="request">
4      <jsp:setProperty name="guestBookSvc" property="name" />
5      <jsp:setProperty name="guestBookSvc" property="email" />
6      <jsp:setProperty name="guestBookSvc" property="comment" />
7  </jsp:useBean>
```

The name attribute of the `jsp:setProperty` action identifies the attribute holding the bean, and the `property` attribute identifies the name of the property being set. The value stored in the bean's property is determined from the HTML form parameter of the same name.



---

**Note** – The `jsp:setProperty` action also has another attribute, `param`, that allows you to specify the HTML form parameter name to be used.

---

Because some forms may be quite large, the JSP specification for the `jsp:setProperty` action provides a shortcut for setting all bean properties at once. By using an asterisk (\*) in the value of the `property` attribute, you command the JSP runtime engine to scan every HTML form parameter in the request object and set every matching property in the bean. This is shown in Code 14-4.

### Code 14-4 Using the `property="*"` Shortcut

```
3   <jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService"
4     scope="request">
5     <jsp:setProperty name="guestBookSvc" property="*" />
6   </jsp:useBean>
```



---

**Note** – The `jsp:setProperty` action can be used anywhere within the JSP page. It does not have to be used only in the body of a `jsp:useBean` action. However, if the `jsp:setProperty` action is used outside of a `jsp:useBean` action, then the processing of the former action will not be conditional.

---

## Control Logic in the Guest Book JSP Page

When the guestBookSvc bean has been created and initialized, it can be used to process the request. However, recall that the guestBook.jsp page is used in two modes: a GET request that represents the HTML form and a POST request that is used to process the form. Therefore, the guestBook.jsp page uses scriptlet code to make the processing (Line 12) conditional. The guestBookSvc bean is used to process the request (Line 13). If the processing is successful, then the guestBook.jsp page forwards the request to the Thank You View (Line 18); otherwise, it forwards the request to the Error View (Line 23). This is shown in Code 14-5.

**Code 14-5** Guest Book Control Logic

```

11  <%
12  if ( request.getMethod().equals("POST") ) {
13      guestBookSvc.processRequest();
14
15      if ( guestBookSvc.wasSuccessful() ) {
16          %>
17          <%-- Proceed to the Thank You page. --%>
18          <jsp:forward page="guestBookThankYou.jsp" />
19      <%
20      } else {
21          %>
22          <%-- There was a failure so print the error messages. --%>
23          <jsp:forward page="guestBookError.jsp" />
24      <%
25      } // end of IF guestBookSvc.wasSuccessful()
26      %>
27      <%
28  } // end of IF HTTP Method was POST
29  %>
```

The `<jsp:forward>` tag is another JSP standard action. It is equivalent to using the `forward` method on a `RequestDispatcher` object. For example:

```
RequestDispatcher view = request.getRequestDispatcher(page);
view.forward(request, response);
return;
```

## Accessing a JavaBeans Component in a JSP Page

Using the `jsp:useBean` action, a JSP page can create a bean and store it in a specific Web application scope. For example, the `guestBookSvc` bean was created in the request scope. Therefore, when the `guestBook.jsp` page forwards to another JSP page during the same request, that page may also access the `guestBookSvc` bean. You do this by using the `jsp:useBean` action again, but in this situation you will not include any initialization code because you expect the bean to already exist. This is shown on line 19 in Code 14-6.

**Code 14-6** Using `jsp:useBean` to Access a JavaBeans Component

```
18 <%-- Retrieve the GuestBookService bean from the request scope. --%>
19 <jsp:useBean id="guestBookSvc" class="sl314.domain.GuestBookService"
scope="request" />
20
21 <BR>
22 Thank you, <jsp:getProperty name="guestBookSvc" property="name"/>, for
23 signing our guest book.
```

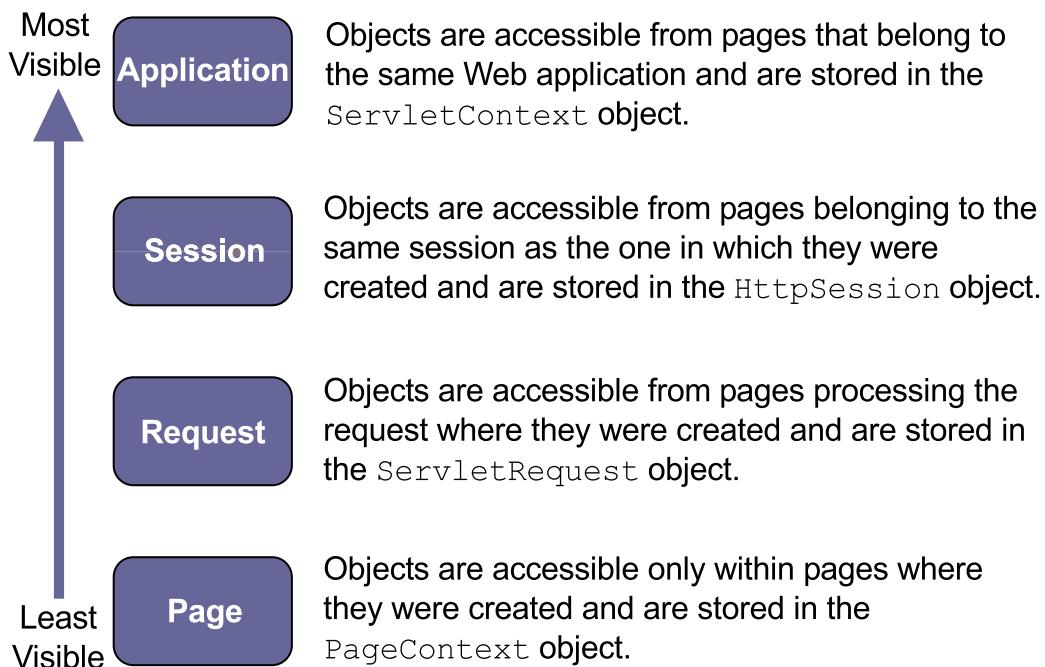
You can also access a bean's properties by using the `jsp:getProperty` action. In the example, a `jsp:getProperty` action is used to access the user's name (Line 22). Whereas most standard actions do not generate any text for the HTML response, the `jsp:getProperty` action does. The code on Line 22 is roughly equivalent to the following expression tag:

```
<%= guestBookSvc.getName() %>
```

As you can see, the expression tag is significantly shorter than the `jsp:getProperty` action tag. To reiterate, you use JSP standard actions to reduce the use of scripting elements (like expression tags), which require knowledge of Java language syntax. This makes it easier for HTML developers to create JSP pages.

## Beans and Scope

Using the `jsp:useBean` action, a JavaBeans component can be created and stored in (or retrieved from) one of four Web application scopes: page, request, session, or application. These scopes are illustrated in Figure 14-6.



**Figure 14-6** Beans and Scoping Rules

## Review of the `jsp:useBean` Action

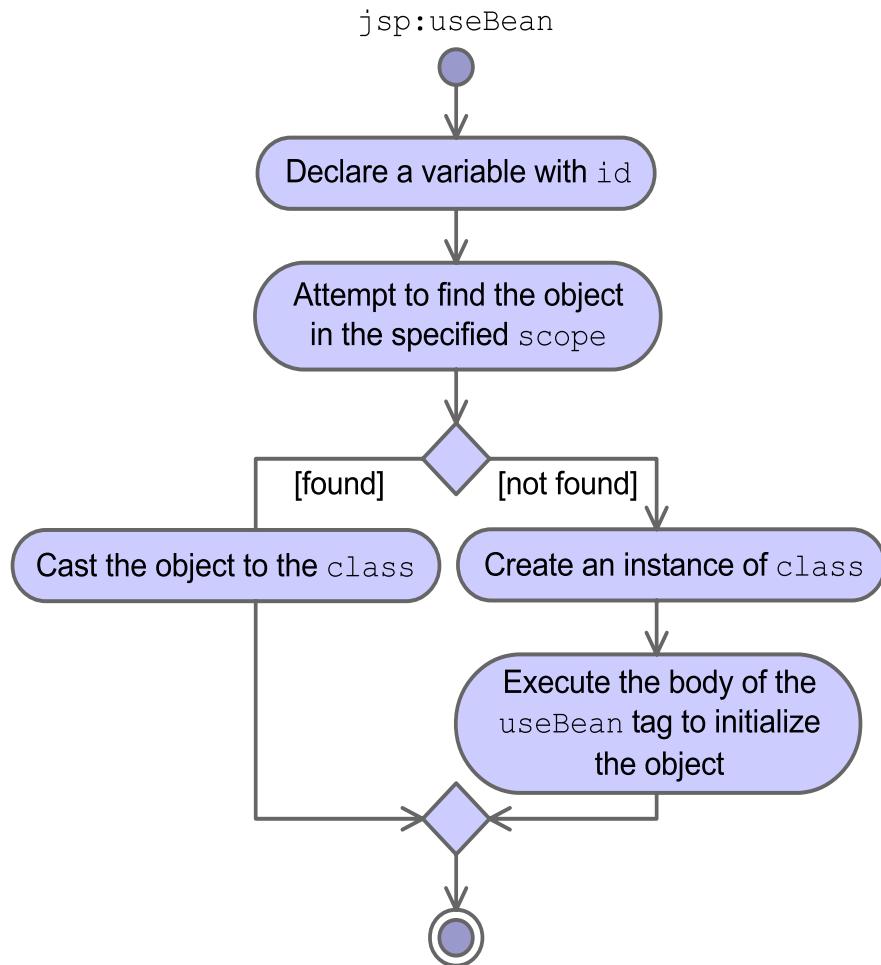
To give you a picture of what the JSP engine is doing behind the scenes, look at the following `jsp:useBean` action:

```
<jsp:useBean id="guestBookSvc" class="GuestBookService" scope="request" />
```

This is roughly equivalent to the following scriptlet code:

```
<%
// First, try to retrieve the bean from the scope
GuestBookService guestBookSvc
= (GuestBookService) request.getAttribute("guestBookSvc");
// If it doesn't exist, then create the bean and store it in the scope
if ( guestBookSvc == null ) {
    guestBookSvc = new GuestBookService();
    request.setAttribute("guestBookSvc", guestBookSvc);
}
%>
```

The activity of the `jsp:useBean` action is illustrated in Figure 14-7.



**Figure 14-7** Activity Diagram of the `jsp:useBean` Action

## Summary

The Model 1 architecture uses a JSP page to handle both Control and View aspects. The Model is handled by a service JavaBeans component. Use the following JSP standard actions to reduce the scripting code in the JSP pages:

- Create a JavaBeans component in a JSP page using the `jsp:useBean` standard action
- Set properties in the bean using either scriptlet code or the `jsp:setProperty` standard action
- Forward from one JSP page to another JSP page using the `jsp:forward` standard action
- Access a bean property using the `jsp:getProperty` standard action

# Certification Exam Notes

This module presented all of the objectives for Section 10, “Designing and Developing JSPs Using JavaBeans,” of the Sun Certification Web Component Developer certification exam:

- 10.1 For any of the following tag functions, match the correctly constructed tag, with attributes and values as appropriate, with the corresponding description of the tag’s functionality:
  - Declare the use of a JavaBeans component within the page
  - Specify, for `jsp:useBean` or `jsp:getProperty` tags, the name of an attribute
  - Specify, for a `jsp:useBean` tag, the class of the attribute
  - Specify, for a `jsp:useBean` tag, the scope of the attribute
  - Access or mutate a property from a declared JavaBeans component
  - Specify, for a `jsp:getProperty` tag, the property of the attribute
  - Specify, for a `jsp:setProperty` tag, the property of the attribute to mutate, and the new value
- 10.2 Given JSP page attribute scopes: request, session, application, identify the equivalent servlet code.
- 10.3 Identify techniques that access a declared JavaBeans component.

# Developing Web Applications Using the Model 2 Architecture

---

## Objectives

Upon completion of this module, you should be able to:

- Design a Web application using the Model 2 architecture
- Develop a Web application using the Model 2 architecture

## Relevance



**Discussion** – The following questions are relevant to understanding what the Model 2 architecture is all about:

- What was the main problem with the Web-MVC pattern, which was introduced in Module 7, “Developing Web Applications Using the MVC Pattern?”
  
- What is the main problem with the Model 1 architecture?

# Designing With Model 2 Architecture

In the first half of this module, you will learn how to design a Web application using the Model 2 architecture. In the second half of this module, you will see the implementation details.

The Model 2 architecture clearly separates the design and implementation of the Model, View, and Controller elements of the Web application. The Model is built from standard Java technology classes or JavaBeans components. The Views are built from JSP pages. The Controller is a servlet. All requests are sent through the servlet Controller, which is responsible for verifying HTML form parameters, updating the Model, and selecting the next View. The selected View generates the dynamic response by using data from the Model. If the Model interacts with a database, then the Model may include Data Access Objects that are hidden from the Web-tier components. This is illustrated in Figure 15-1.

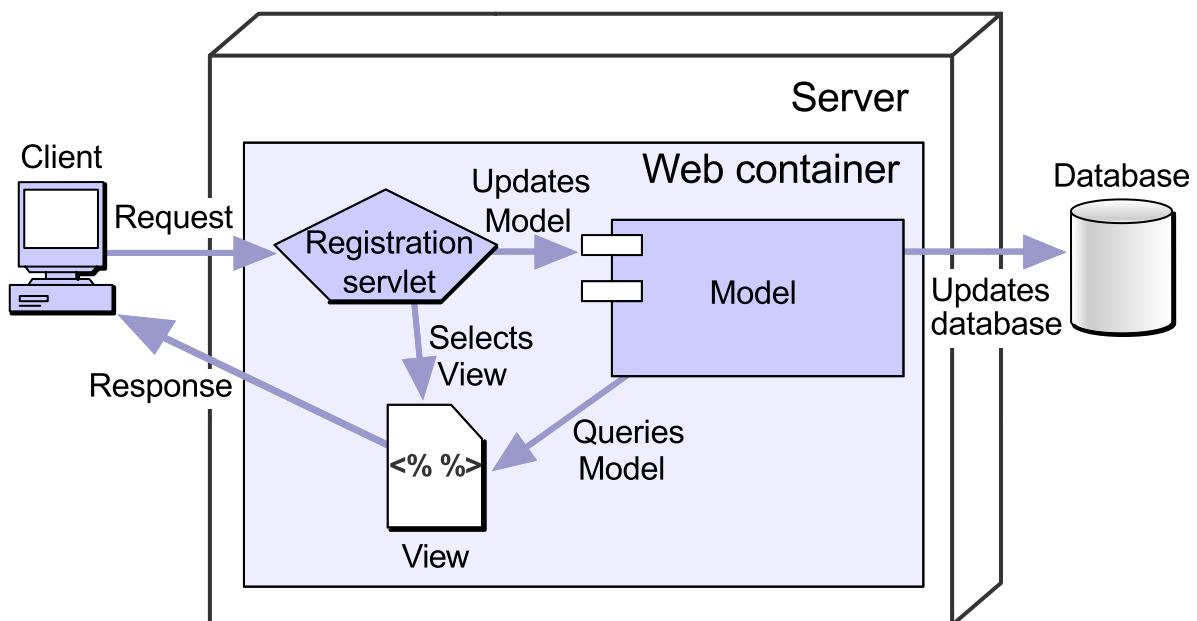
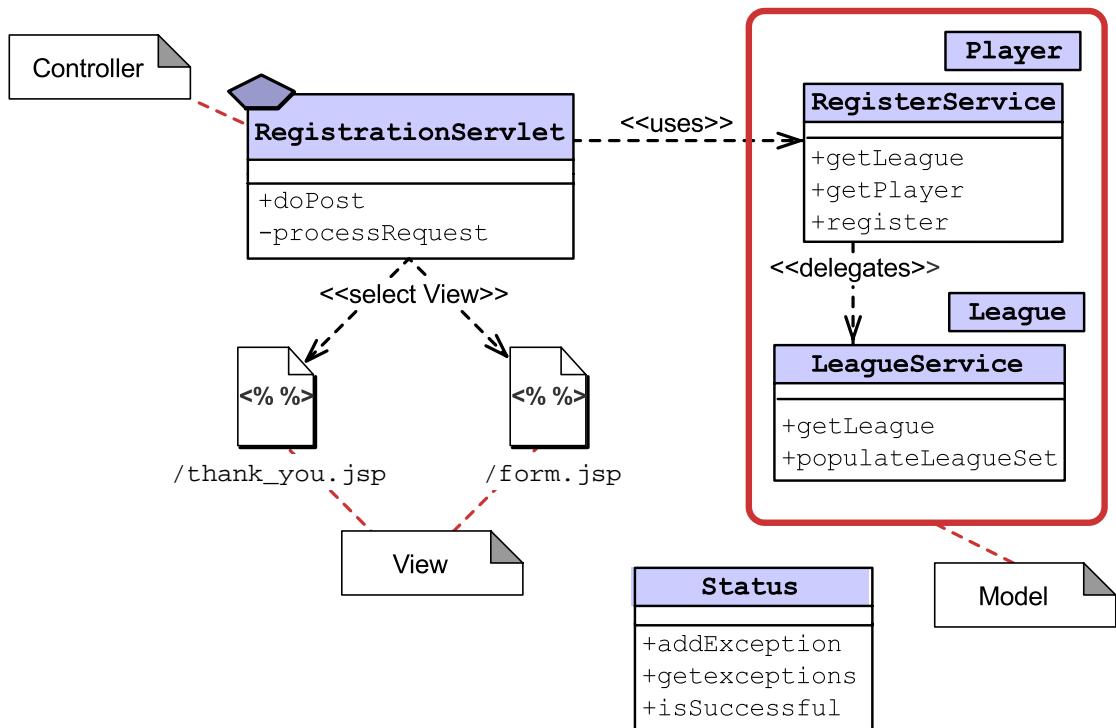


Figure 15-1 A Model 2 Web Application Architecture

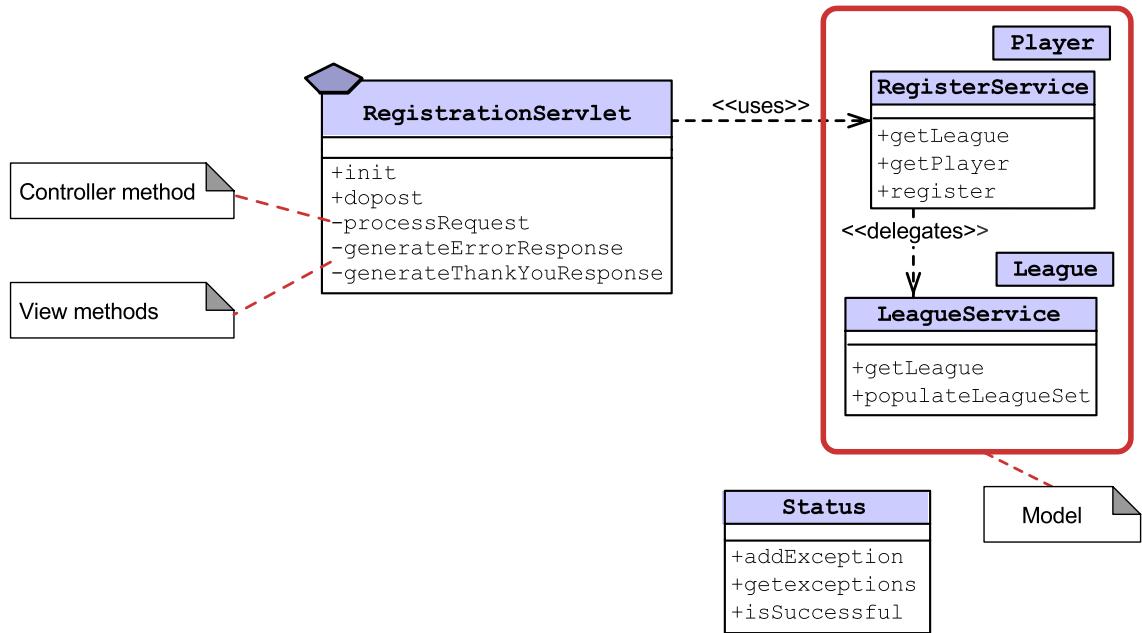
## The Soccer League Example Using Model 2 Architecture

In the Soccer League example, the RegistrationServlet acts as the Controller and the `thank_you.jsp` and `form.jsp` pages are the Views. The Model elements have not changed. These component relationships are illustrated in Figure 15-2.



**Figure 15-2** The New Web-MVC Component Diagram

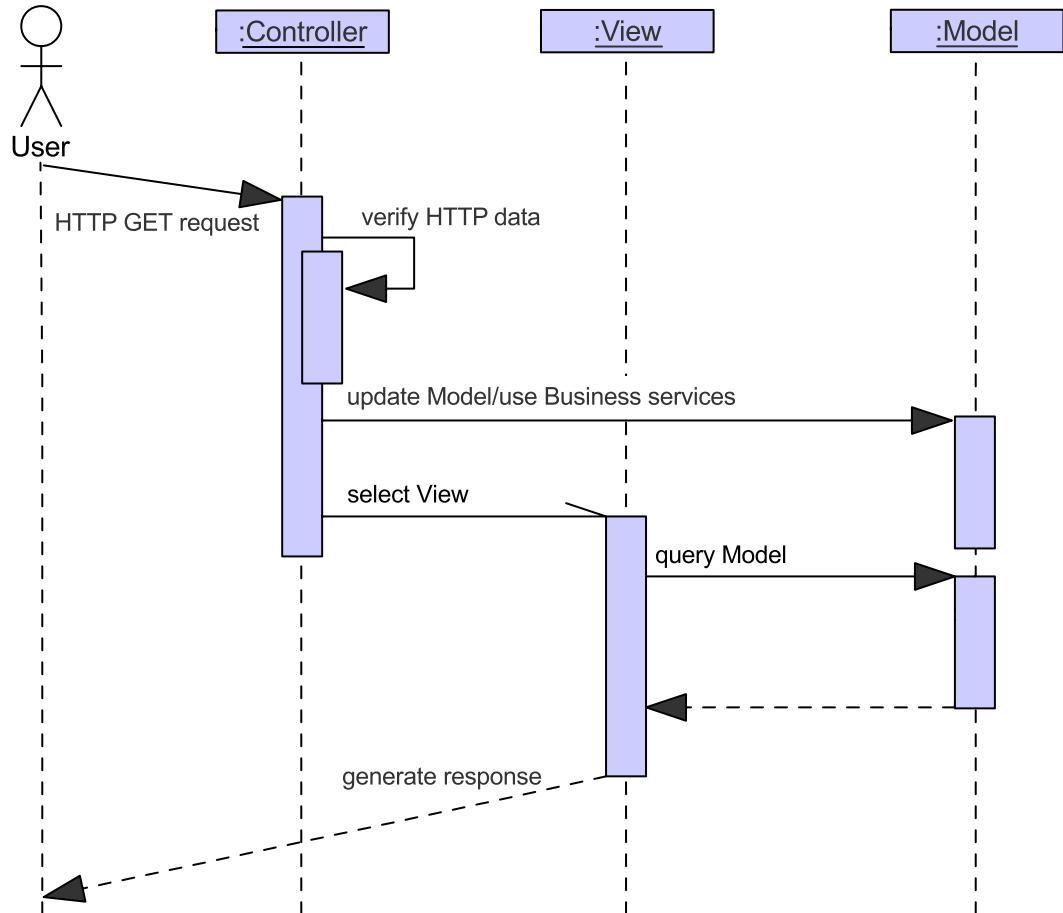
Compare that diagram with the old Web-MVC design from Module 7 in which the Views were methods within the servlet class. This is illustrated in Figure 15-3.



**Figure 15-3** The Old Web-MVC Component Diagram

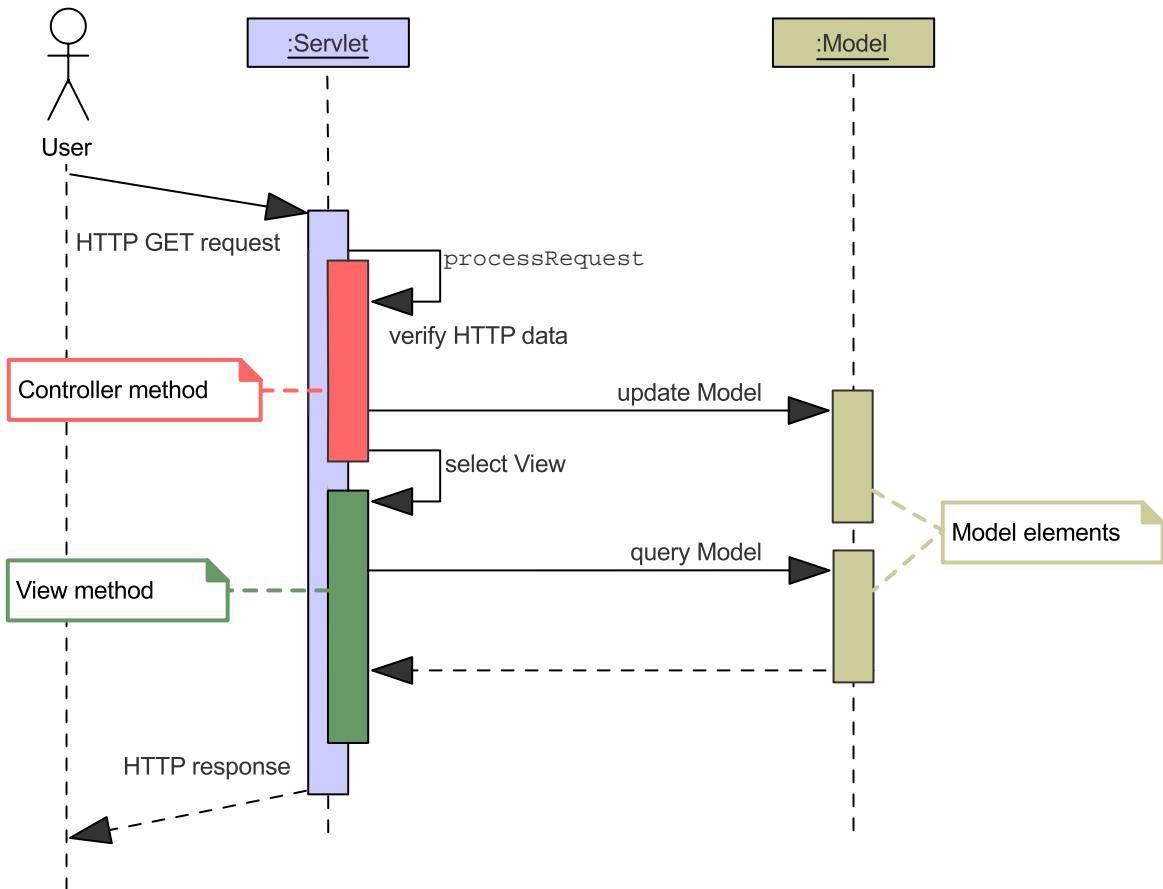
## Sequence Diagram of Model 2 Architecture

The interaction between the Web user, the servlet, the Model, and the View is illustrated with a sequence diagram in Figure 15-4.



**Figure 15-4** The New Web-MVC Sequence Diagram

Compare Figure 15-4 on page 15-6 with the old Web-MVC design from Module 7 in which servlet methods acted as Views. This is illustrated in Figure 15-5.



**Figure 15-5** The Old Web-MVC Sequence Diagram

## Developing With Model 2 Architecture

The servlet Controller must:

- Verify the HTML form data
- Call the business services in the Model
- Store domain objects in the request (or session) scope
- Select the next user View

The JSP page Views must:

- Render the user interface (in HTML)
- Access the domain objects to generate dynamic content

## Controller Details

By eliminating the old View methods, the RegistrationServlet class has been reduced to two methods. The doPost service method dispatches the request to the processRequest method. This is shown in Code 15-1.

**Code 15-1** The doPost Method

```
1 package sl314.web;
2
3 // Business Logic Component imports
4 import sl314.domain.League;
5 import sl314.domain.Player;
6 import sl314.domain.RegisterService;
7 // Servlet imports
8 import javax.servlet.http.HttpServlet;
9 import javax.servlet.http.HttpServletRequest;
10 import javax.servlet.http.HttpServletResponse;
11 import javax.servlet.ServletContext;
12 import javax.servlet.RequestDispatcher;
13 import javax.servlet.ServletException;
14 // Support classes
15 import sl314.util.Status;
16 import java.io.IOException;
17
18
19 public class RegistrationServlet extends HttpServlet {
20
21     public void doPost(HttpServletRequest request,
22                         HttpServletResponse response)
23         throws IOException, ServletException {
24
25         // Dispatch the request
26         processRequest(request, response);
27     }
28 }
```

The processRequest method contains the Controller logic. The initialization of this method includes:

- Declaring the view local variable that will hold a request dispatcher
- Creating the business logic object
- Creating the Status object and storing it in the request scope
- Retrieving the HTML form parameters

The initialization code is shown in Code 15-2.

**Code 15-2** Initialization of the Controller Logic

```
29  /**
30  * This method performs the Control aspects of the application.
31  */
32 public void processRequest(HttpServletRequest request,
33                     HttpServletResponse response)
34         throws IOException, ServletException {
35     // Declare the dispatcher for the View
36     RequestDispatcher view = null;
37
38     // Create business logic objects
39     RegisterService regService = new RegisterService();
40
41     // Create the status object and store it in the request for use
42     // by the 'Registration Form' View (if necessary)
43     Status status = new Status();
44     request.setAttribute("status", status);
45
46     // Extract HTML form parameters
47     String season = request.getParameter("season");
48     String year = request.getParameter("year");
49     String name = request.getParameter("name");
50     String address = request.getParameter("address");
51     String city = request.getParameter("city");
52     String province = request.getParameter("province");
53     String postalCode = request.getParameter("postalCode");
54     String division = request.getParameter("division");
55
```

The next step in the Controller logic is to perform data verification. Every field in the form must be completed, and an `Exception` object must be created to flag missing data. These “error messages” are presented to the user in a red bullet list. If any of the verification steps flag an error, then the servlet forwards the request back to the `form.jsp` page View (Lines 84 and 85). The `form.jsp` page presents the user with the registration form again and displays the list of verification errors. It also fills in the default values of the form field from the parameters passed in from the last request. This verification code is shown in Code 15-3.

### Code 15-3 Registration Form Verification Code

```

56 // Verify 'Select League' form fields
57 if ( season.equals("UNKNOWN") ) {
58     status.addException(new Exception("Please select a league season."))
59 }
60 if ( year.equals("UNKNOWN") ) {
61     status.addException(new Exception("Please select a league year."));
62 }
63
64 // Verify 'Enter Player Information' form fields
65 if ( (name == null) || (name.length() == 0) ) {
66     status.addException(new Exception("Please enter your name."));
67 }
68 if ( (address == null) || (address.length() == 0)
69     || (city == null) || (city.length() == 0)
70     || (province == null) || (province.length() == 0)
71     || (postalCode == null) || (postalCode.length() == 0) ) {
72     status.addException(new Exception("Please enter your full address."))
73 }
74
75 // Verify 'Select Division' form fields
76 if ( division.equals("UNKNOWN") ) {
77     status.addException(new Exception("Please select a division."));
78 }
79
80 // If any of the above verification failed, then return the
81 // 'Registration Form' View and return without proceeding with the
82 // rest of the business logic
83 if ( ! status.isSuccessful() ) {
84     view = request.getRequestDispatcher("form.jsp");
85     view.forward(request, response);
86     return;
87 }
88

```

If no verification errors are found, then the Controller uses the business services to perform the registration in three steps:

1. Retrieve the requested league and verify that it exists; throw an exception if it does not exist.
2. Retrieve a player object and populate it with the address information.
3. Call the register method to perform the actual registration.

This code is shown in Code 15-4.

### Code 15-4 Updating the Business Model

```
89     try {  
90         // Retrieve the league object, and verify that it exists  
91         League league = regService.getLeague(year, season);  
92         if ( league == null ) {  
93             throw  
94                 new Exception("The league you selected does not yet exist;"  
95                             + " please select another.");  
96         }  
97  
98         // Create and populate the player object  
99         Player player = regService.getPlayer(name);  
100        player.setAddress(address);  
101        player.setCity(city);  
102        player.setProvince(province);  
103        player.setPostalCode(postalCode);  
104  
105        // Now delegate the real work to the RegisterService object  
106        regService.register(league, player, division);  
107        request.setAttribute("league", league);  
108        request.setAttribute("player", player);  
109
```

The last step in the Controller logic is to select the next View for the user. You use a request dispatcher to forward the current request to a JSP page which encodes the View that is returned to the user. In this servlet, if the processing of the registration form was successful, then the next View is the `thank_you.jsp` page. Otherwise, if any exceptions are caught, then the next View goes back to the `registration_form.jsp` page. This code is shown in Code 15-5.

**Code 15-5** Selecting the Next View

```
110 // The registration process was successful,  
111 // forward to the 'Thank You' View  
112 view = request.getRequestDispatcher("thank_you.jsp");  
113 view.forward(request, response);  
114  
115 // Catch any business logic errors.  
116 // Forward back to the 'Registration Form' View to report  
117 // to the user what errors occurred.  
118 } catch (Exception e) {  
119     status.addException(e);  
120     view = request.getRequestDispatcher("form.jsp");  
121     view.forward(request, response);  
122 }  
123 }
```

## Request Dispatchers

Request dispatchers allow servlets to forward the user's original request to some other dynamic Web resource such as another servlet or a JSP page.

A request dispatcher to a named servlet can be retrieved from the context object. For example:

```
ServletContext context = getServletContext();
RequestDispatcher servlet = context.getNamedDispatcher("MyServlet");
servlet.forward(request, response);
```

A request dispatcher can also be retrieved from the request object. For example:

```
RequestDispatcher view = request.getRequestDispatcher("tools/nails.jsp");
view.forward(request, response);
```

The string parameter to this method takes a URL that is relative to the current directory of the user's original request URL. So, if that code ran from a servlet that is mapped to the URL /store/hardwareShop, then the effective URL would be /store/tools/nails.jsp.

## View Details

The responsibility of the View is simple: generate an HTML response that uses dynamic information from the Model. The Controller passes Model information to the View using request (or session) scope attributes. The View uses the `jsp:useBean` standard action to retrieve these objects from the request scope (Lines 19 and 20). Then the View uses the `jsp:getProperty` standard action to retrieve particular properties from the Model beans (Lines 23 and 24). This code is shown in Code 15-6.

**Code 15-6** The thank\_you.jsp Page

```

1  <%@ page session="false" %>
2  <HTML>
3
4  <HEAD>
5  <TITLE>Registration: Thank You</TITLE>
6  </HEAD>
7
8  <BODY BGCOLOR='white'>
9
10 <TABLE BORDER='0' WIDTH='600'>
11 <TR>
12   <TD COLSPAN='2' BGCOLOR='#CCCCFF' ALIGN='center'>
13     <H3>Registration: Thank You</H3>
14   </TD>
15 </TR>
16 </TABLE>
17
18 <%-- Retrieve the LEAGUE and PLAYER beans from the REQUEST scope. --%>
19 <jsp:useBean id="league" scope="request" class="sl314.domain.League"/>
20 <jsp:useBean id="player" scope="request" class="sl314.domain.Player"/>
21
22 <BR>
23 Thank you, <jsp:getProperty name="player" property="name"/>, for registering
24 in the <B><jsp:getProperty name="league" property="title"/></B> league.
25
26 </BODY>
27
28 </HTML>
```

## Summary

Model 2 architecture uses the Web-MVC pattern. The MVC pattern clearly distinguishes the role of each technology:

- A servlet is used as the Controller.
- JSP pages are used as the Views.
- Java technology classes (or JavaBeans components) are used as the Model.

The servlet Controller passes the domain objects to the View through request (or session) attributes; it selects the next View by using the forward method on a RequestDispatcher object.

The JSP page View accesses the domain objects using the `jsp:useBean` action.

## Certification Exam Notes

This module presented some of the objectives for Section 13, "Web Tier Design Patterns," of the Sun Certification Web Component Developer certification exam:

- 13.1 Given a scenario description with a list of issues, select the design pattern (Value Objects, MVC, Data Access Object or Business Delegate) that would best solve those issues.
- 13.2 Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: Value Objects, MVC, Data Access Object, Business Delegate.

The Data Access Object pattern is presented in Module 12, "Integrating Web Applications With Databases," and the Business Delegate pattern is presented in Module 20, "Integrating Web Applications With Enterprise JavaBeans Components."



# Building Reusable Web Presentation Components

---

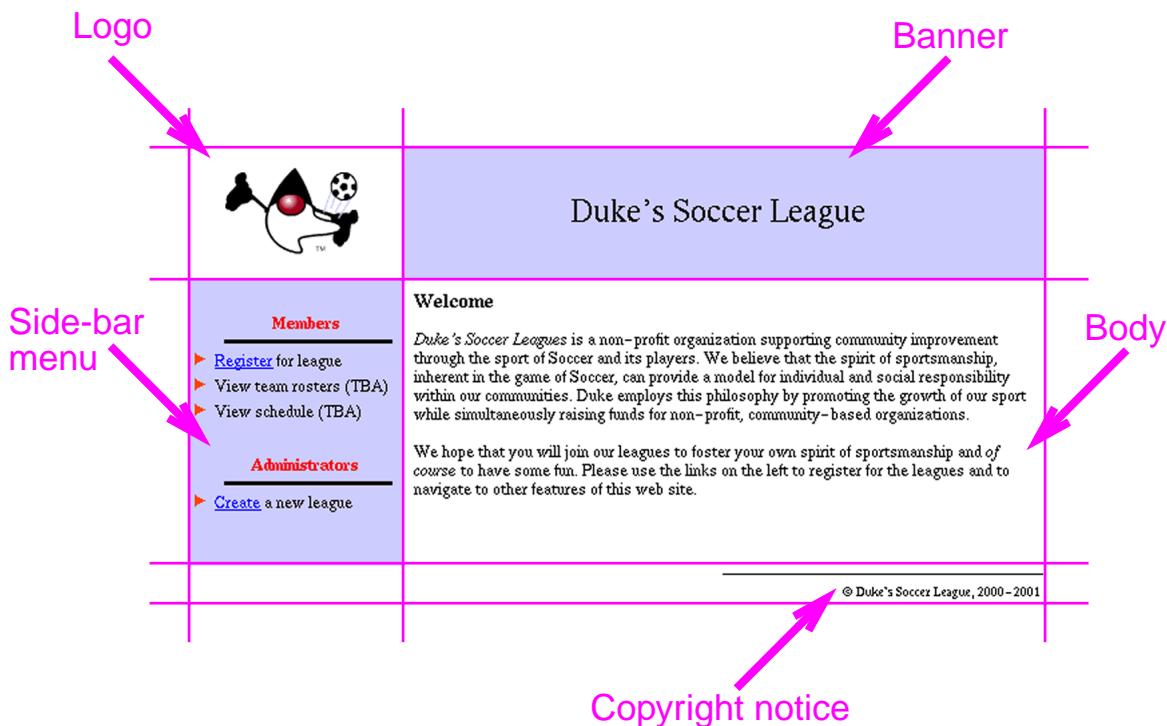
## Objectives

Upon completion of this module, you should be able to:

- Describe how to build Web page layouts from reusable presentation components
- Write JSP technology code using the `include` directive
- Write JSP technology code using the `jsp:include` standard action

# Complex Page Layouts

Many Web applications consist of a deep hierarchy of content and services. Navigational aids are needed for complex Web sites. These aids must be shown in every page to give a consistent look and feel to the site. Corporate logos and banners also add to the visual appeal of the Web pages. It is standard practice in Web page design to place these visual components (or presentation components) into a grid-based layout. An example layout for the Soccer League Web application is illustrated in Figure 16-1.



**Figure 16-1** Soccer League Page Layout

The most common technique for constructing a grid layout in an HTML page is to use one or more hidden tables. Each presentation fragment populates a single cell in an HTML table. The table is defined with the BORDER attribute equal to zero in the TABLE start tag; this makes the table hidden. Likewise, set the CELLPADDING and CELLSPACING attributes to zero. A simplified, skeletal layout table structure is shown in Code 16-1 on page 16-3.

**Code 16-1**     HTML Structure of a Table Layout

```

<BODY>
<TABLE BORDER='0' CELLPADDING='0' CELLSPACING='0' WIDTH='640'>
<TR>
  <TD WIDTH='160'> <!-- logo here --> </TD>
  <TD WIDTH='480'> <!-- banner here --> </TD>
</TR>
<TR>
  <TD WIDTH='160'> <!-- side-bar menu here --> </TD>
  <TD WIDTH='480'> <!-- main content here --> </TD>
</TR>
<TR>
  <TD WIDTH='160'> <!-- nothing here --> </TD>
  <TD WIDTH='480'> <!-- copyright notice here --> </TD>
</TR>
</TABLE>
</BODY>

```

## What Does a Fragment Look Like?

A presentation fragment can be a chunk of static HTML or of dynamic JSP technology code. It can be a small or large chunk of content. An important restriction is that fragments should not contain HTML, HEAD, or BODY tags. The intent of fragments is that they are embedded into other JSP pages and presumably these pages include the overall HTML and BODY tag structure for a complete HTTP response. For example, you could construct a fragment that generates a copyright notice that is used in all of your pages. Such a fragment is shown in Code 16-2.

**Code 16-2**     Example Presentation Fragment

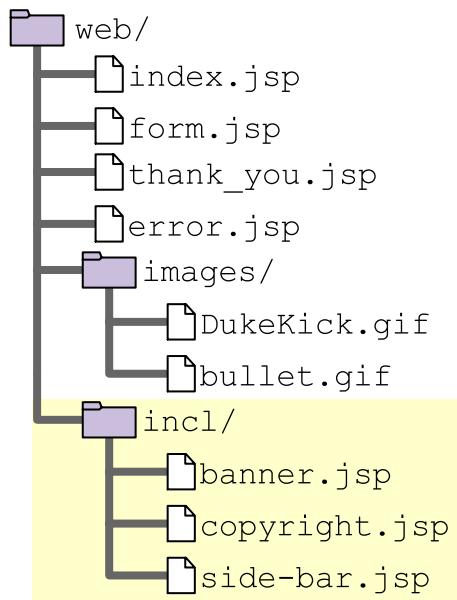
```

1  <%@ page import="java.util.Calendar" %>
2
3  <%-- get today's year --%>
4  <%
5      Calendar today = Calendar.getInstance();
6      int year = today.get(Calendar.YEAR);
7  %>
8
9      <SPACER HEIGHT='15'>
10     <HR WIDTH='50%' SIZE='1' NOSHADE COLOR='blue'>
11     <FONT SIZE='2' FACE='Helvetica, san-serif'>
12         &copy; Duke's Soccer League, 2000-<%= year %>
13     </FONT>

```

## Organizing Your Presentation Fragments

Presentation fragments exist as HTML or JSP technology files within your Web application. Collect your presentation fragments into a central location (directory) within your Web application file hierarchy. In the Soccer League example, the presentation fragment files are located in a directory called incl (short for include). This is illustrated in Figure 16-2.



**Figure 16-2** Presentation Fragment Directory

# Including JSP Page Fragments

There are two approaches to including presentation fragments in your main JSP pages:

- The `include` directive
- The `jsp:include` standard action

## Using the `include` Directive

The `include` directive allows you to include a fragment in the text of the main JSP page at translation time. The syntax of this JSP technology directive is:

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

An example use of the `include` directive is shown in Code 16-3.

**Code 16-3** Example Use of the `include` Directive

```
30 <!-- START of side-bar -->
31 <TD BGCOLOR='#CCCCFF' WIDTH='160' ALIGN='left'>
32   <%@ include file="/incl/side-bar.jsp" %>
33 </TD>
34 <!-- END of side-bar -->
```

The content of the fragment is embedded into the text of the main JSP page while the main page is being translated into a servlet class. If the included fragment also includes other fragments, then all of the fragments are embedded recursively until all of the `include` directives are eliminated.

The URL in the `file` attribute may be a path that is relative to the directory position of the original request URL or it may be an absolute path (denoted by a leading "/" character in the URL) that is relative to the Web application's context root. Because presentation fragment files are often stored in a central directory at the top-level of the Web application, it is a good practice to use the absolute URL notation.

## Using the `jsp:include` Standard Action

The `jsp:include` standard action allows you to include a fragment in the text of the main JSP page at runtime. The syntax of this JSP technology action tag is:

```
<jsp:include page="fragmentURL" />
```

An example use of the `jsp:include` action is shown in Code 16-4.

**Code 16-4** Example Use of the `jsp:include` Standard Action

```
20    <!-- START of banner -->
21    <jsp:include page="/incl/banner.jsp" />
22    <!-- END of banner -->
```

The content of the fragment is embedded into the HTTP response of the main JSP page at runtime. If the included fragment also includes other fragments, then all of the fragments are embedded recursively until all of the `jsp:include` actions have been processed. All of this happens at runtime. The text of the fragment *is not* placed in the main JSP page at translation time.



**Note** – The `jsp:include` action is equivalent to using the `include` method on a `RequestDispatcher` object. For example:

```
RequestDispatcher fragment = request.getRequestDispatcher(page);
fragment.include(request, response);
```

---

Semantically, the `include` directive and the `jsp:include` standard action achieve the same result: They both include an HTML or JSP page fragment in another JSP page. However, there are subtle differences. A fragment that uses the `include` directive will not be automatically updated when the fragment file is updated in the Web container. A fragment that uses the `jsp:include` standard action incurs a runtime performance penalty.

## Using the `jsp:param` Standard Action

Sometimes, you might want to alter a presentation fragment at runtime. For example, the Soccer League banner changes based on the user's movement through the Web application. At the end of the registration process the user is presented with the Thank You page. This message is added to the banner. This is illustrated in Figure 16-3.



**Figure 16-3** The Thank You Page Uses a Dynamic Parameter

The `jsp:include` action allows you to pass additional parameters to the presentation fragment at runtime using the `jsp:param` standard action. The syntax of this JSP technology action tag and the relationship to the `jsp:include` action is:

```
<jsp:include page="fragmentURL">
  <jsp:param name="paramName" value="paramValue" />
</jsp:include>
```

The `jsp:param` action is contained in the `jsp:include` action. The `jsp:include` action tag can contain any number of `jsp:param` actions. An example use of the `jsp:param` action is shown in Code 16-5.

**Code 16-5** Example Use of the include Directive

```
22  <!-- START of banner -->
23  <jsp:include page="/incl/banner.jsp">
24    <jsp:param name="subTitle" value="Thank You!" />
25  </jsp:include>
26  <!-- END of banner -->
```

The name-value pair specified in the `jsp:param` action is stored in the request object passed to the fragment's JSP page. These name-value pairs are stored as if they were HTML form parameters; therefore, the fragment page must use the `getParameter` method to retrieve these values (Line 6). This is shown in Code 16-6.

**Code 16-6** The banner.jsp Fragment

```
1  <%@ page import="sl314.domain.League" %>
2
3  <%-- Determine the page title and sub-title. --%>
4  <%
5      String bannerTitle = "Duke's Soccer League";
6      String subTitle = request.getParameter("subTitle");
7
8      // Use the title of the selected league (if known)
9      League league = (League) request.getAttribute("league");
10     if ( league != null ) {
11         bannerTitle = league.getTitle();
12     }
13 %>
14
15     <FONT SIZE='5' FACE='Helvetica, san-serif'>
16     <%= bannerTitle %>
17     </FONT>
18
19     <% if ( subTitle != null ) { %>
20     <BR><BR>
21     <FONT SIZE='4' FACE='Helvetica, san-serif'>
22     <%= subTitle %>
23     </FONT>
24     <% } %>
```

## Summary

Complex layouts reuse fragments of presentation code throughout a large Web application. For maintainability, isolate these fragments into separate files. The JSP specification provides two mechanisms for including presentation fragments in main JSP pages:

- The `include` directive is used to include fragments at translation time.
- The `jsp:include` action is used to include fragments at runtime.

## Certification Exam Notes

This module presented all of the objectives for Section 9, "Designing and Developing Reusable Web Components," of the Sun Certification Web Component Developer certification exam:

- 9.1 Given a description of required functionality, identify the JSP directive or standard tag in the correct format with the correct attributes required to specify the inclusion of a Web component into the JSP page.

# Developing JSP Pages Using Custom Tags

---

## Objectives

Upon completion of this module, you should be able to:

- Describe the problem with JSP technology scripting tags
- Given an existing custom tag library, develop a JSP page using this tag library

## Relevance



**Discussion** – The following questions are relevant to understanding why custom tag libraries are a good idea:

- Who in your organization will be creating JSP pages?
  
- Suppose you start with a small number of JSP pages in a Web application and have a significant amount of scripting code in these pages. What problems can you foresee as the Web application grows?

## Additional Resources



**Additional resources** – The following reference provides additional information on the topics described in this module:

- Hall, Marty. *Core Servlets and JavaServer Pages*. Upper Saddle River: Prentice Hall PTR, 2000.

## Job Roles Revisited

Job roles for developing a large Web application might include:

- *Content Creators* – Responsible for creating the Views of the application, which are primarily composed of HTML pages
- *Web Component Developers* – Responsible for creating the Control elements of the application, which are almost exclusively Java technology code
- *Business Component Developers* – Responsible for creating the Model elements of the application, which might reside on the Web server or on a remote server (such as an EJB technology server)

Using custom tags, you can completely develop a JSP page without the use of Java technology code (scripting elements). This permits Content Creators to create JSP pages without knowing the Java programming language.

# Introducing Custom Tag Libraries

This section introduces the concept of a custom tag and custom tag libraries.

## Contrasting Custom Tags and Scriptlet Code

A *custom tag* is an XML tag used in a JSP page to represent some dynamic action or the generation of content within the page during runtime. Custom tags are used to eliminate scripting elements in a JSP page. In the Soccer League registration form example, the fields in the form can be populated from the request parameters, if the form is being “returned” to the user because of some errors in processing the form. For example, the address form field can be populated from the address HTML form parameter using the following scripting code shown in Code 17-1.

### Code 17-1 Form Population Using Scripting Code

```
151 <TD ALIGN='right'>Address:</TD>
152 <TD>
153 <% String addressValue = request.getParameter("address");
154     if ( addressValue == null ) addressValue = ""; %>
155 <INPUT TYPE='text' NAME='address'
156     VALUE='<%= addressValue %>' SIZE='50'>
157 </TD>
```

Instead, you can replace these lines of code with a single, empty custom tag reference in the JSP page. This is shown in Code 17-2.

### Code 17-2 Form Population Using Custom Tags

```
88 <TD ALIGN='right'>Address:</TD>
89 <TD>
90 <INPUT TYPE='text' NAME='address'
91     VALUE='<soccer:getReqParam name="address"/>' SIZE='50'>
92 </TD>
```

Advantages of custom tags compared with scriptlet code:

- Removes Java technology code from a JSP page

Java technology code in a JSP page is hard to read and maintain. Java technology code in a JSP page is difficult for some HTML editors to handle properly.

- Custom tags are reusable components

Scripting code is not reusable. To reuse a chunk of scripting code requires you to cut-and-paste from one page to another. Custom tags encapsulate Java technology code in the tag handler. To reuse a custom tag only requires you to include the custom tag in the new JSP page.

- Support of standard job roles

Content Creators can use custom tags instead of scripting elements in JSP pages to facilitate dynamic behavior without having to know the Java programming language.

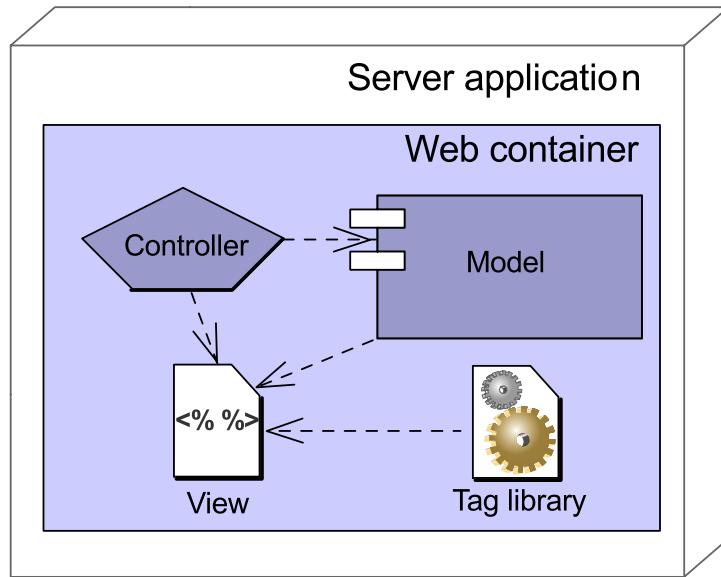
## Developing JSP Pages Using Custom Tags

In this module, you will see how to replace scripting code with custom tags using a custom tag library. Use the followings guidelines when creating JSP pages using an existing tag library:

- Use a custom tag library description
- Understand that custom tags follow the XML tag rules
- Declare the tag library in the JSP page and in the Web application deployment descriptor
- Use an empty custom tag in a JSP page
- Use a custom tag in a JSP page to make a section of the HTML response conditional
- Use a custom tag in a JSP page to iterate over a section of the HTML response

## What Is a Custom Tag Library?

A *custom tag library* is a collection of custom tag handlers and the tag library descriptor file. A custom tag library is a Web component that is part of the Web application. This is illustrated in Figure 17-1.



**Figure 17-1** A Tag Library as a Web Component

The tag library can access objects in any part of the application that are accessible to the View.

## Custom Tag Syntax Rules

JSP technology custom tags use XML syntax. There are four fundamental XML rules that all custom tags must follow:

- Standard tag syntax must conform to the following structure:

```
<prefix:name {attribute={"value" | 'value'}}*>
  body
</prefix:name>
```

Example:

```
<soccer:heading alignment="center" color="blue">
  Soccer League Registration Form
</soccer:heading>
```

- Empty tag syntax must conform to the following structure:

```
<prefix:name {attribute={"value" | 'value'} }* />
```

Example:

```
<soccer:getReqParam name="address" />
```

- Tag names, attributes, and prefixes are case sensitive.

For example:

- <soccer:heading> is different from <soccer:Heading>
- <soccer:heading> is different from <Soccer:heading>
- <soccer:heading color="blue"> is different from <soccer:heading COLOR="blue">.

- Tags must follow nesting rules:

```
<tag1>
    <tag2>
        </tag2>
    </tag1>
```

Valid example:

```
<soccer:checkStatus>
    <soccer:iterateOverErrors>
        <%-- JSP code showing a single error message --%>
    </soccer:iterateOverErrors>
</soccer:checkStatus>
```

Invalid example:

```
<soccer:checkStatus>
    <soccer:iterateOverErrors>
        <%-- JSP code showing a single error message --%>
    </soccer:checkStatus>
    </soccer:iterateOverErrors>
```

For more information about XML syntax, read Appendix E, “Quick Reference for XML.”

## Example Tag Library: Soccer League

The following sections present a description of tags in the Soccer League tag library. Each tag description includes the purpose and behavior of the tag, the type of body content (or empty), each attribute and a description of what the attribute does, and a usage example.

This tag library description represents a document that the Web component development team might give to the Content Creators.

### The `getReqParam` Tag

The `getReqParam` tag inserts the value of the named request parameter into the output. If the parameter does not exist, then either the default is used (if provided) or the empty string is used.

- Body content: This is an empty tag.
- Attribute: `name`

This mandatory attribute is the name of the request parameter.

- Attribute: `default`

This optional attribute can provide a default if the parameter does not exist.

- Example:

```
<soccer:getReqParam name="countryCode" default="JP"/>
```

### The `heading` Tag

The `heading` tag generates a hidden HTML table that creates a colorful, and well-formatted page heading.

- Body content: This tag contains JSP technology code.
- Attribute: `alignment`

This mandatory attribute specifies the text alignment of the heading. The acceptable values are: `left`, `center`, and `right`.

- Attribute: `color`

This mandatory attribute specifies the color of the box around the heading. The acceptable values include HTML color names or an RGB code.

- Example:

```
<soccer:heading alignment="center" color="#CCCCFF">
    Soccer League Registration Form
</soccer:heading>
```

## The checkStatus Tag

The checkStatus tag is used to make the body conditional based upon whether the “status” attribute has been set and if the Status object indicates “was unsuccessful.” This is used when a forms page is revisited (after a form validation error occurred).

- Body content: This tag contains JSP technology code.
- Example:

```
<soccer:checkStatus>
    <%-- JSP code to show error messages --%>
</soccer:checkStatus>
```

## The iterateOverErrors Tag

The iterateOverErrors tag iterates over all of the exception objects in the Status object. This tag *must* be used within the checkStatus tag.

- Body content: This tag contains JSP technology code.
- Example:

```
<soccer:checkStatus>
    <soccer:iterateOverErrors>
        <%-- JSP code showing a single error message --%>
    </soccer:iterateOverErrors>
</soccer:checkStatus>
```

## The getCurrentMessage Tag

The getCurrentMessage tag is an empty tag that prints the current error message. This tag *must* be used within the iteratorOverErrors tag.

- Body content: This is an empty tag.
- Example:

```
<soccer:checkStatus>
    <soccer:iterateOverErrors>
        <soccer:getCurrentMessage/>
    </soccer:iterateOverErrors>
</soccer:checkStatus>
```

# Developing JSP Pages Using a Custom Tag Library

A custom tag library is made up of two parts: the JAR file of tag handler classes and the tag library descriptor (TLD). The TLD is an XML file that names and declares the structure of each custom tag in the library. To create a Web application that uses a tag library, you must declare the TLD in the Web application's deployment descriptor. An example is shown in Code 17-3.

**Code 17-3** Declaring the TLD in the Deployment Descriptor

```
61 <taglib>
62   <taglib-uri>http://www.soccer.org/taglib</taglib-uri>
63   <taglib-location>/WEB-INF/taglib.tld</taglib-location>
64 </taglib>
```

The TLD file is usually stored in the WEB-INF directory and the JAR file is stored in the WEB-INF/lib directory (Solaris Operating Environment) or WEB-INF\lib (Microsoft Windows). The tag library declaration in the deployment descriptor maps the physical TLD file location to an arbitrary Universal Resource Identifier (URI). This URI is used in the JSP pages that need to access that particular tag library.

A JSP page may use a tag library by directing the JSP technology translator to access the TLD. This is specified using the taglib JSP technology directive. This directive includes the TLD URI and a custom tag prefix. This is shown in Line 2 of Code 17-4.

**Code 17-4** Declaring the TLD in the JSP Page

```
1 <%@ page session="false" %>
2 <%@ taglib uri="http://www.soccer.org/taglib" prefix="soccer" %>
3
4 <HTML>
5
6 <HEAD>
7 <TITLE>Registration Form</TITLE>
8 </HEAD>
9
10 <BODY BGCOLOR='white'>
11
12 <soccer:heading alignment="center" color="#CCCCFF">
13   Soccer League Registration Form
14 </soccer:heading>
```

A JSP page might include several tag libraries. The prefix is used to distinguish custom tags from each of these libraries. This prefix is prepended to the custom tag name (Lines 12 and 14). Every custom tag library must be used in a JSP page with a prefix. Every custom tag from a given library used in the JSP page must use the prefix assigned to that library.

## Using an Empty Custom Tag

An empty tag is one that contains no body. Such a tag is often used to embed dynamic content from the domain objects in the Web application. For example, the `getReqParam` custom tag is used to retrieve the value from a request parameter. An example use of the `getReqParam` custom tag is shown in Code 17-5

**Code 17-5**      Using an Empty Custom Tag

```
88 <TD ALIGN='right'>Address:</TD>
89 <TD>
90   <INPUT TYPE='text' NAME='address'
91     VALUE='<soccer:getReqParam name="address" />' SIZE='50'>
92 </TD>
```

If you forget the end slash (/) character, then the JSP technology translator thinks that this is a start tag and it attempts to look for a corresponding end tag. This is a common mistake and you will see a translation error message in this situation.

## Using a Conditional Custom Tag

A custom tag might be used to perform a conditional check to decide whether to include some HTML content. For example, in the Soccer League Web application the registration Controller might send the registration form back to the user if there are any verification errors. The following scriptlet code might be used in the registration form to determine if the page should show the error messages:

```
<%
  if ( (status != null) && !status.isSuccessful() ) {
%
<%-- "error messages" JSP code --%>
<%
  } // end of IF status
%>
```

This scriptlet code can be eliminated by using a custom tag that surrounds the “error messages” JSP technology code:

```
<soccer:checkStatus>
<%-- "error messages" JSP code --%>
</soccer:checkStatus>
```

## Using an Iterative Custom Tag

A custom tag might also be used to perform an iteration over dynamic JSP technology content. Iterations in JSP pages are used to construct lists (like UL or OL lists) or rows in a table. In the error handling code of the registration form, the JSP page needs to construct a UL list of all of the error messages stored in the Status object. This might be accomplished using scriptlet code that iterates over a “list item” (LI) HTML element. This JSP technology code is shown in Code 17-6.

### Code 17-6 Iteration Using Scriptlet Code

```
26 There were problems processing your request:  
27 <UL>  
28 <%  
29     Iterator errors = status.getExceptions();  
30     while ( errors.hasNext() ) {  
31         Exception ex = (Exception) errors.next();  
32     %>  
33         <LI><%= ex.getMessage() %>  
34     <%  
35     } // end of WHILE loop  
36     %>  
37 </UL>
```

Iteration scriptlet code can be replaced using a custom tag. This JSP technology code is shown in Code 17-7.

### Code 17-7 Using an Iterative Custom Tag

```
19 There were problems processing your request:  
20 <UL>  
21     <soccer:iterateOverErrors>  
22         <LI><soccer:getErrorMessage/>  
23     </soccer:iterateOverErrors>  
24 </UL>
```

## Summary

This module presented the development of JSP pages using a custom tag library. Custom tag libraries provide a mechanism to completely remove scriptlet code from your JSP pages. Follow these guidelines to use a tag library in your Web application:

- Use the `taglib` element in the deployment descriptor
- Use the `taglib` directive in the JSP page to identify which tag library is being used and by which prefix
- Use XML syntax (plus the prefix) when using custom tags

# Certification Exam Notes

This module presented all of the objectives for Section 11, "Designing and Developing JSPs Using Custom Tags," of the Sun Certification Web Component Developer certification exam:

11.1 Identify properly formatted tag library declarations in the Web application deployment descriptor.

11.2 Identify properly formatted `taglib` directives in a JSP page.

11.3 Given a custom tag library, identify properly formatted custom tag usage in a JSP page. Uses include:

- An empty custom tag
- A custom tag with attributes
- A custom tag that surrounds other JSP code
- Nested custom tags



# Developing a Simple Custom Tag

---

## Objectives

Upon completion of this module, you should be able to:

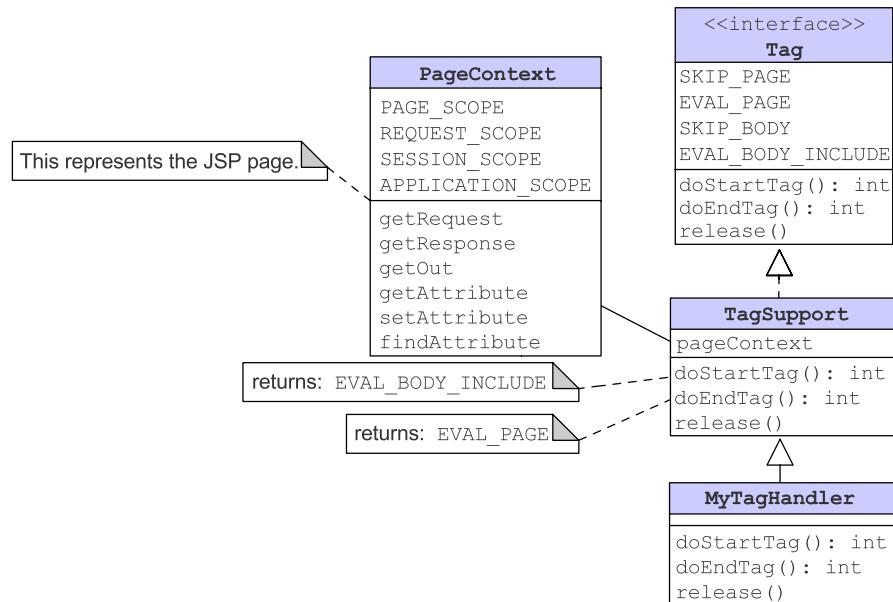
- Describe the structure and execution of a custom tag in a JSP page
- Develop the tag handler class for a simple, empty custom tag
- Write the tag library description for a simple, empty custom tag
- Develop a custom tag that includes its body in the content of the HTTP response

# Overview of Tag Handlers

This section provides an overview on how to build a custom tag handler class and how to declare the custom tag in the tag library descriptor (TLD) file.

## Fundamental Tag Handler API

All custom tag handler classes implement the Tag interface (in the `javax.servlet.jsp.tagext` package). This interface declares the methods that the JSP servlet executes when the start and end tags are processed at runtime. The `doStartTag` method is executed by the JSP servlet when the start tag is processed; likewise, the `doEndTag` method is executed when the end tag is processed. The `TagSupport` class is supplied by the JSP API as a default implementation of the Tag interface. This class provides a protected instance variable, `pageContext`, which gives the tag handler access to the data stored in the JSP page at runtime. The `PageContext` class provides access to the implicit variables by using accessor methods. For example, the `getRequest` method returns the HTTP request object. You should build your tag handler classes by extending the `TagSupport` class. You should override the `doStartTag` and `doEndTag` methods as appropriate. These API relationship are illustrated in Figure 18-1.



**Figure 18-1** Tag Handler API

## Tag Handler Life Cycle

There are three fundamental elements to a custom tag within a JSP page: the start tag, the body, and the end tag. To process the custom tag, the JSP page first creates the tag handler object, stores the tag attributes in the tag handler object, and then calls the `doStartTag` method on the tag handler object. Based on the return value from the `doStartTag` method, the JSP page might process the body of the tag. Finally, the end tag is processed by executing the `doEndTag` method on the tag handler object and releasing the tag handler object. This is shown in Figure 18-2.

```
<tag attr="value"> ← Start tag: create tag handler object, initialize attributes, doStartTag  
    body ← If EVAL_BODY_INCLUDE then process body  
</tag> ← End tag: doEndTag, release tag handler object
```

**Figure 18-2** Tag Handler Life Cycle Events

## Pseudo-Code of the Tag Handler Life Cycle

One view of the tag handler life cycle can be shown using pseudo-code. This pseudo-code is approximately what is generated inside of the translated JSP servlet class for a particular tag like the one shown in Figure 18-2 on page 18-3. This pseudo-code is shown in Code 18-1. The following are the detailed steps in the life cycle:

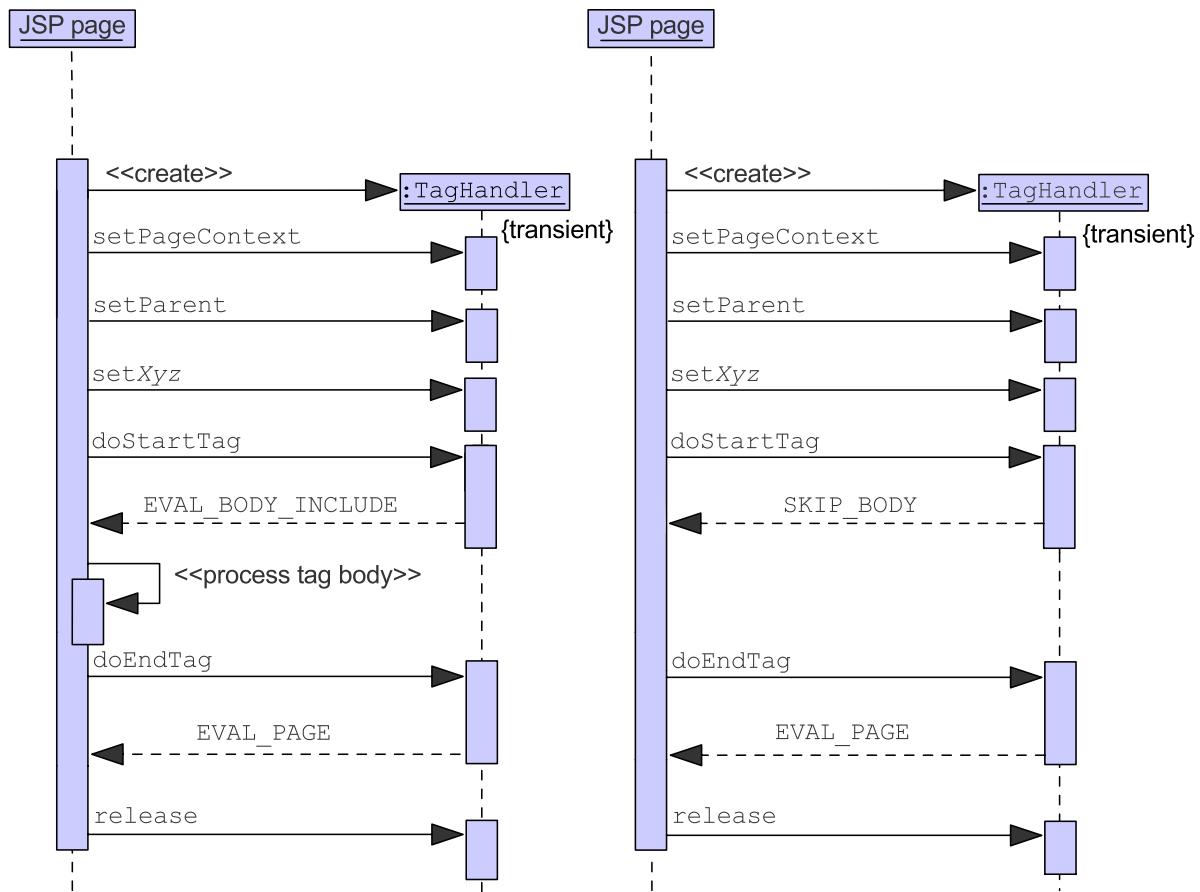
1. The tag handler is created (Line 1), and the page context is stored (Line 2).
2. The tag attributes are stored in the tag object (Line 3).
3. The `doStartTag` method is called (Line 5).
4. If the return value is not `SKIP_BODY`, then the JSP technology code of the body is executed (Line 7). This code is completely fictitious, but it demonstrates the processing of the JSP technology content within the body of the tag.
5. The `doEndTag` method is called (Line 9).
6. If the return value is `SKIP_PAGE`, then the JSP page halts. Usually, the return value will be `EVAL_PAGE`, which tells the JSP page to continue processing.
7. The tag handler is released (Line 13).

### Code 18-1 JSP Servlet Pseudo-Code

```
1 TagHandler tagObj = new TagHandler();
2 tagObj.setPageContext(pageContext);
3 tagObj.setAttr("value");
4 try {
5     int startTagResult = tagObj.doStartTag();
6     if ( startTagResult != Tag.SKIP_BODY ) {
7         out.write("body"); // process the BODY of the tag
8     }
9     if ( tagObj.doEndTag() == Tag.SKIP_PAGE ) {
10        return; // do not continue processing the JSP page
11    }
12 } finally {
13     tagObj.release();
14 }
```

## Sequence Diagram of the Tag Handler Life Cycle

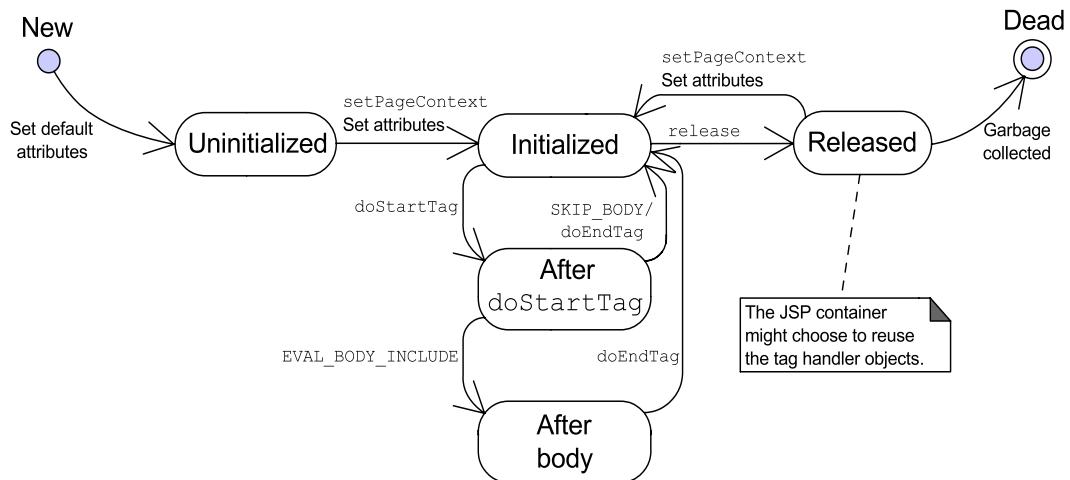
A Unified Modeling Language (UML) sequence diagram provides another perspective on the runtime life cycle of a tag handler object. Two sequence diagrams are shown in Figure 18-3. The diagram on the right shows the scenario in which the `doStartTag` method returns the `SKIP_BODY` value. In this scenario the body of the tag does not get processed. The diagram on the left shows the scenario in which the `doStartTag` method returns the `EVAL_BODY_INCLUDE` value. In this scenario the body of the tag is processed by the JSP page before proceeding to the `doEndTag` method.



**Figure 18-3** Tag Handler Life Cycle Sequence Diagrams

## State Diagram of the Tag Handler Life Cycle

A state diagram of the tag handler object provides another perspective. This diagram is illustrated in Figure 18-4.



**Figure 18-4** Tag Handler Object State Diagram

## Tag Library Relationships

In Module 17, you saw how to use a custom tag library in a JSP page. You use the `taglib` directive to declare the use of a tag library. Then you can use a custom tag using the prefix and tag name. The prefix is declared in the `taglib` directive. The tag library must be declared in the Web application deployment descriptor. The `taglib-location` element in the deployment descriptor specifies the file name of the tag library descriptor. The TLD is an XML file that contains the declarations for every custom tag in the tag library. For example, if the JSP page uses the `getReqParam` tag, the TLD file must contain a `tag` element that names this tag (case-sensitive) and identifies the tag handler class as well as the attributes of the custom tag. The final piece of the puzzle is the tag handler class itself. These relationships are illustrated in Figure 18-5 on page 18-7.

**JSP Page**

```
<%@ taglib prefix="soccer" uri="http://www.soccer.org/taglib" %>
<soccer:getReqParam name="countryCode" default="JP" />
```

**Deployment Descriptor**

```
<taglib>
  <taglib-uri>
    http://www.soccer.org/taglib
  </taglib-uri>
  <taglib-location>
    WEB-INF/taglib.tld
  </taglib-location>
</taglib>
```

**Tag Library Descriptor**

```
<tag>
  <name>getReqParam</name>
  <tag-class>sl314.web.taglib.GetRequestParamHandler</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>name</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>default</name>
    <required>false</required>
  </attribute>
</tag>
```

**Tag Handler Class**

<b>GetRequestParamHandler</b>
{from sl314.web.taglib}
setName(String)
setDefault(String)
doStartTag():int
release()

**Figure 18-5** Tag Handler Component Relationships

# Developing a Tag Handler Class

This section reviews the description of the `getReqParam` custom tag from the Soccer League example and then describes in detail the source code of the tag handler class. To develop a tag handler class, you must perform the following steps:

1. Extend the `TagSupport` class.
2. Provide private instance variables for each tag attribute. Provide an explicit, default value for all attributes that are not required.
3. Create mutator methods, `setXyz(String)`, for each tag attribute.
4. Override the `doStartTag` method to handle the start tag processing. It should return `SKIP_BODY` if the tag is an empty tag; otherwise, it should return `EVAL_BODY_INCLUDE` to tell the JSP page to process the body.
5. Override the `release` method to reset any instance variables, for tag attributes, which might not be set if reused. In other words, return the instance variable values to their default values.

## The `getReqParam` Tag

The `getReqParam` tag inserts the value of the named request parameter into the output. If the parameter does not exist, then either the default is used (if provided) or the empty string is used.

- Body content: This is an empty tag.
- Attribute: `name`

This mandatory attribute is the name of the request parameter.

- Attribute: `default`

This optional attribute can provide a default if the parameter does not exist.

- Example:

```
<prefix:getReqParam name="countryCode" default="JP"/>
```

## The getReqParam Tag Handler Class

The `getReqParam` custom tag is implemented by the `GetRequestParamHandler` class. This class extends the `TagSupport` class. The declaration of the tag handler class, the required import statements, and the package statement are shown in Code 18-2.

**Code 18-2** The `GetRequestParamHandler` Declaration

```

1  package sl314.web.taglib;
2
3  // Servlet imports
4  import javax.servlet.jsp.tagext.TagSupport;
5  import javax.servlet.jsp.JspWriter;
6  import javax.servlet.jsp.JspException;
7  import javax.servlet.ServletRequest;
8  import java.io.IOException;
9
10
11 /**
12  * This class handles the "get request parameter" tag.
13  * This is an empty tag.
14  */
15 public class GetRequestParamHandler extends TagSupport {

```

The `getReqParam` tag includes two tag attributes. The tag handler class must supply mutator methods for all tag attributes, which are implemented as JavaBeans properties. These tag attribute properties are shown in Code 18-3.

**Code 18-3** The Tag Attribute Properties

```

15 public class GetRequestParamHandler extends TagSupport {
16
17     private String name;
18     private String defaultValue = "";
19
20     public void setName(String name) {
21         this.name = name;
22     }
23
24     public void setDefault(String defaultValue) {
25         this.defaultValue = defaultValue;
26     }
27

```

Because the getReqParam tag is an empty tag, the tag handler class need only implement the doStartTag method. This method retrieves the value of the CGI parameter specified by the name tag attribute from the request object (Line 30), which is retrieved from the pageContext object (Line 29). If this value is null, then the default value (specified by the default tag attribute) is written to the HTTP response output stream (Line 35). Otherwise, the request parameter value is written to the response stream (Line 37). This method then returns the SKIP\_BODY value because this is an empty tag. The doStartTag method is shown in Code 18-4.

### Code 18-4 The doStartTag Method

```
28  public int doStartTag() throws JspException {
29      ServletRequest request = pageContext.getRequest();
30      String paramValue = request.getParameter(name);
31      JspWriter out = pageContext.getOut();
32
33      try {
34          if ( paramValue == null ) {
35              out.print(defaultValue);
36          } else {
37              out.print(paramValue);
38          }
39      } catch (IOException ioe) {
40          throw new JspException(ioe);
41      }
42
43      // This is an empty tag, skip any body
44      return SKIP_BODY;
45  }
```

Finally, the release method is implemented. In this example, the release method returns the default tag attribute property to its initial state, the empty string. This release method is shown in Code 18-5.

### Code 18-5 The release Method

```
47  public void release() {
48      defaultValue = "";
49  }
50 } // end of GetRequestParamHandler class
```

# Configuring the Tag Library Descriptor

The tag library descriptor file is an XML file. The structure of a TLD file is shown in Code 18-6. Lines 1–4 specify that the file is an XML file (Line 1) and declare the document type for XML validation (Lines 2–4). The root tag is taglib (Lines 6 and 20). The beginning of the taglib element includes some boilerplate information about the tag library. The most important element is the uri element which specifies the arbitrary URI for this tag library. This URI is used in the Web deployment descriptor taglib-uri element. After the introductory elements the TLD can declare any number of tag elements, which identify specific custom tags in the tag library.

**Code 18-6** Structure of the TLD File

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE taglib
3      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
4      "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_2.dtd">
5
6  <taglib>
7
8      <tlib-version>1.2</tlib-version>
9      <jsp-version>1.2</jsp-version>
10     <short-name>Sports League Web Taglib</short-name>
11     <uri>http://www.soccer.org/taglib</uri>
12     <description>
13         An example tag library for the Soccer League Web Application.
14     </description>
15
16     <tag>
17         <!-- tag declaration -->
18     </tag>
19
20 </taglib>
```

## Tag Declaration Element

The `tag` element in the TLD file declares a single custom tag. The `name` subelement specifies the case-sensitive name of the custom tag (Line 2). This is the name used in the JSP page. The `tag-class` subelement specifies the fully qualified Java technology class name for the tag handler. The `body-content` subelement specifies the type of content that is allowed between the start and end tags (Line 4). Because the `getReqParam` tag is an empty tag, the value of `empty` is declared in this subelement. This is followed by a description of the custom tag (Lines 5–9). Finally, the `attribute` subelement specifies a custom tag attribute. The `getReqParam` tag has two attributes: `name` and `default`. The declaration of the `getReqParam` tag is shown in Code 18-7.

**Code 18-7** The `getReqParam` Tag Declaration

```
1 <tag>
2   <name>getReqParam</name>
3   <tag-class>s1314.web.taglib.GetRequestParamHandler</tag-class>
4   <body-content>empty</body-content>
5   <description>
6     This tag inserts into the output the value of the named
7     request parameter. If the parameter does not exist, then
8     either the default is used (if provided) or the empty string.
9   </description>
10  <attribute>
11    <name>name</name>
12    <required>true</required>
13    <rtpvalue>false</rtpvalue>
14  </attribute>
15  <attribute>
16    <name>default</name>
17    <required>false</required>
18    <rtpvalue>false</rtpvalue>
19  </attribute>
20 </tag>
```

## Custom Tag Body Content

The body-content element can contain one of these three values:

- empty – This value tells the JSP technology engine that the tag does not accept any body content.
- JSP – This value tells the JSP technology engine that the tag can accept arbitrary JSP technology code in its body.
- tag-dependent – This value tells the JSP technology engine that the tag can accept arbitrary content in the tag body. The JSP technology engine will not process the body, but will pass it directly to the tag handler.

## Custom Tag Attributes

The attribute element contains three subelements:

- name – The name of the attribute (case-sensitive).
- required – Whether the attribute must be used in every tag use in a JSP page. The value of the required field must be either true or false.
- rtxprvalue – Whether the attribute value might be generated from JSP technology code at runtime. For example, if you had a tag that generated HTML headings, then you might want to include an attribute level, which takes a number. The value of the rtxprvalue field must be either true or false.

```
<prefix:heading level="<% currentHeading %>">  
    Act IV - Romeo Awakes  
</prefix:heading>
```

## Custom Tag That Includes the Body

This section describes how to develop a custom tag that includes the body between the start and end tags. The example is to create a simple page heading. An example of a page heading is shown in Figure 18-6.



**Figure 18-6** Example Page Heading in Netscape

The HTML that generates these page headings uses a hidden table with a single table cell. That cell is colored (in this case, blue) and includes the heading text aligned in the center. The HTML code to generate a page heading is shown in Code 18-8.

**Code 18-8** HTML Code to Generate a Page Heading

```
11  <TABLE BORDER='0' CELLPACING='0' CELLSPACING='0' WIDTH='600'>
12  <TR>
13  <TD BGCOLOR='#CCCCFF' ALIGN='center'>
14  <H3>Soccer League Registration Form</H3>
15  </TD>
16 </TR>
17 </TABLE>
```

The heading custom tag can be used to generate the same HTML code. This JSP technology code is shown in Code 18-9.

**Code 18-9** Custom Tag to Generate a Page Heading

```
12 <soccer:heading alignment="center" color="#CCCCFF">
13   Soccer League Registration Form
14 </soccer:heading>
```

## The heading Tag

The heading tag generates a hidden HTML table that creates a colorful, and well-formatted page heading.

- Body content: This tag contains JSP technology code.
- Attribute: `alignment`

This mandatory attribute specifies the text alignment of the heading. The acceptable values are: `left`, `center`, and `right`.

- Attribute: `color`

This optional attribute specifies the color of the box around the heading. The acceptable values include HTML color names or an RGB code.

- Example:

```
<soccer:heading alignment="center" color="#CCCCFF">
    Soccer League Registration Form
</soccer:heading>
```

## The heading Tag Handler Class

The heading custom tag is implemented by the HeadingHandler class. Because the heading tag includes two tag attributes, alignment and color, the HeadingHandler class must implement JavaBeans mutator methods for each tag attribute property (Lines 22 and 26). The color tag attribute is not required; therefore, the class supplies a default value (Line 20). This code is shown in Code 18-10.

**Code 18-10** The HeadingHandler Declaration

```
11  /**
12   * This class handles the "heading" tag.
13   * This tag takes a body, which is the text of the heading.
14  */
15 public class HeadingHandler extends TagSupport {
16
17     private static String DEFAULT_COLOR = "white";
18
19     private String alignment;
20     private String color = DEFAULT_COLOR;
21
22     public void setAlignment(String alignment) {
23         this.alignment = alignment;
24     }
25
26     public void setColor(String color) {
27         this.color = color;
28     }
29
```

The heading tag needs to generate HTML code that fits the template shown in Code 18-8 on page 18-14. The HTML tags from the TABLE start tag (Line 11) to the H3 start tag (Line 14) are generated when the start tag of the heading custom tag is encountered. This is handled by the doStartTag method (Lines 30–43) in Code 18-11 on page 18-17. The tag attributes are used to generate the dynamic elements of the table row HTML attributes (Lines 35 and 36).

Next, the content of the heading tag must be inserted between the H3 start and end tags. The heading tag handler tells the JSP page to include the body of the tag by returning the EVAL\_BODY\_INCLUDE constant from the doStartTag method (Line 42).

Finally, the `doEndTag` method must generate the HTML end tags that close the `H3` and `TABLE` tags (Lines 47–49).

**Code 18-11** The HeadingHandler Life Cycle Methods

```
30 public int doStartTag() throws JspException {
31     JspWriter out = pageContext.getOut();
32     try {
33         out.println("<TABLE BORDER='0' CELLSPACING='0' " +
34                     + "CELLPADDING='0' WIDTH='600'>");
35         out.println("<TR ALIGN='" + alignment +
36                     + "' BGCOLOR='" + color + "'>");
37         out.println("  <TD><H3>");
38     } catch (IOException ioe) {
39         throw new JspException(ioe);
40     }
41     // Tell the JSP page to include the tag body
42     return EVAL_BODY_INCLUDE;
43 }
44 public int doEndTag() throws JspException {
45     JspWriter out = pageContext.getOut();
46     try {
47         out.println("  </H3></TD>");
48         out.println("</TR>");
49         out.println("</TABLE>");
50     } catch (IOException ioe) {
51         throw new JspException(ioe);
52     }
53     // Continue processing the JSP page
54     return EVAL_PAGE;
55 }
```

## The heading Tag Descriptor

The heading tag must be declared in the TLD file. The important element of this descriptor is the body-content subelement. Because the heading tag includes its body in the HTML response, the body-content value must be JSP to indicate that the JSP servlet must process the content of the body of the heading tag (Line 40). The heading tag descriptor is shown in Code 18-12.

**Code 18-12** The heading Tag Descriptor

```
37 <tag>
38   <name>heading</name>
39   <tag-class>sl314.web.taglib.HeadingHandler</tag-class>
40   <body-content>JSP</body-content>
41   <description>
42     This tag creates a customizable page heading.
43   </description>
44   <attribute>
45     <name>alignment</name>
46     <required>true</required>
47     <rtpvalue>false</rtpvalue>
48   </attribute>
49   <attribute>
50     <name>color</name>
51     <required>false</required>
52     <rtpvalue>false</rtpvalue>
53   </attribute>
54 </tag>
```

# Summary

This module presented how to develop a simple custom tag library. A tag library requires the coordination of four components: the JSP page, the Web application deployment descriptor, the tag library descriptor, and the tag handler classes.

The tag handler class:

- Should extend the TagSupport class
- Must contain mutator methods, `setXyz(String)`, for every tag attribute
- Must override the `doStartTag` method
- Should override the `release` method to reset tag attributes to their default values

The tag library descriptor:

- Must include the URI of the tag library
- Must include an entry for each tag in the library
- Each tag entry must include:
  - The `name` element
  - The `tag-class` element (the fully qualified tag handler class name)
  - The `body-content` element (usually either `empty` or `JSP`)
  - An `attribute` element for every tag attribute.  
Every attribute element must include these subelements: `name`, `required` (either `true` or `false`), and `rteprvalue` (usually `false`).

## Certification Exam Notes

This module presented most of the following objectives for Section 12, "Designing and Developing A Custom Tag Library," of the Sun Certification Web Component Developer certification exam:

12.1 Identify the tag library descriptor element names that declare the following:

- The name of the tag
- The class of the tag handler
- The type of content that the tag accepts
- Any attributes of the tag

12.2 Identify the tag library descriptor element names that declare the following:

- The name of a tag attribute
- Whether a tag attribute is required
- Whether the attribute's value can be dynamically specified

12.3 Given a custom tag, identify the necessary value for the body-content TLD element for any of the following tag types:

- An empty tag
- A custom tag that surrounds other JSP code
- A custom tag that surrounds content that is used only by the tag handler

12.4 Given a tag event method (`doStartTag`, `doAfterBody`, and `doEndTag`), identify the correct description of the methods trigger.

12.5 Identify valid return values for the following methods:

- The `doStartTag` method
- The `doAfterBody` method
- The `doEndTag` method
- The `PageContext.getOut` method

12.6 Given a “BODY” or “PAGE” constant, identify a correct description of the constant’s use in the following methods:

- The doStartTag method
- The doAfterBody method
- The doEndTag method

12.7 Identify the method in the custom tag handler that accesses:

- A given JSP implicit variable
- The JSP page’s attributes

12.8 Identify methods that return an outer tag handler from within an inner tag handler.

Accessing JSP page attributes (part of objective 12.7) and the objectives for the doAfterBody method are discussed in Module 19, “Developing Advanced Custom Tags.” Objective 12.8 is not presented in this course. Review the JSP specification for more details.



# Developing Advanced Custom Tags

---

## Objectives

Upon completion of this module, you should be able to:

- Develop a custom tag in which the body is conditionally included
- Develop a custom tag in which the body is iteratively included

## Writing a Conditional Custom Tag

In Module 18, you saw how to construct a tag handler class that includes the body of the custom tag within a JSP page. In this section, you will see how to construct a tag handler that conditionally includes the body.

A conditional tag tells the JSP page to include the body of the tag if a certain condition is met. The conditional aspect of the tag is implemented through the return value of the `doStartTag` method.

- If the condition is true, the `doStartTag` method should return the `EVAL_BODY_INCLUDE` value.
- If it is false, it should return the `SKIP_BODY` value.

### Example: The `checkStatus` Tag

The `checkStatus` tag is used to make the body conditional based upon whether the “status” attribute has been set and if the `Status` object indicates “was unsuccessful.” This is used when a forms page is revisited (after a form validation error occurred).

- Body content: This tag contains JSP technology code.
- Example:

```
<soccer:checkStatus>
    <%-- JSP code to show error messages --%>
</soccer:checkStatus>
```

The checkStatus tag is used in the registration form JSP page. If the user does not fill in the form completely, then several error messages are generated in the Status object. The checkStatus tag is used in the JSP page to determine if the “show error messages” JSP technology code should be evaluated. This is shown in Code 19-1.

#### **Code 19-1** Using the checkStatus Tag

```

16 <%-- BEGIN: error status presentation --%>
17 <soccer:checkStatus>
18 <FONT COLOR='red'>
19 There were problems processing your request:
20 <UL>
21   <soccer:iterateOverErrors>
22     <LI><soccer:getErrorMessage/>
23   </soccer:iterateOverErrors>
24 </UL>
25 </FONT>
26 </soccer:checkStatus>
27 <%-- END: error status presentation --%>
```

## The checkStatus Tag Handler

The checkStatus tag is implemented by the CheckStatusHandler class. The doStartTag method is used to determine if the body of the tag should be evaluated. This is determined by whether the Status object indicates that the form processing was not successful. If that condition is true, then the constant EVAL\_BODY\_INCLUDE is returned. This is shown in Code 19-2.

#### **Code 19-2** The checkStatus Tag Handler

```

18 public class CheckStatusHandler extends TagSupport {
19
20   public int doStartTag() {
21     Status status
22       = (Status) pageContext.getAttribute("status",
23                                     PageContext.REQUEST_SCOPE);
24
25     if ( (status != null) && !status.isSuccessful() ) {
26       return EVAL_BODY_INCLUDE;
27     } else {
28       return SKIP_BODY;
29     }
30   }
31 }
```

## The checkStatus Tag Life Cycle

The life cycle of the checkStatus tag is very similar to the generic life cycles that were introduced in Module 18, “Developing a Simple Custom Tag.” The JSP servlet pseudo-code for the checkStatus tag is shown in Code 19-3. The important element is the conditional test on the return value of the doStartTag method (Lines 5–7). If the return value is not SKIP\_BODY (for example, if it is EVAL\_BODY\_INCLUDE), then the JSP servlet is instructed to evaluate the body within the checkStatus start and end tags.

**Code 19-3**      The checkStatus Tag Life Cycle

```
1  CheckStatusHandler tagObj = new CheckStatusHandler();
2  tagObj.setPageContext(pageContext);
3  try {
4      int startTagResult = tagObj.doStartTag();
5      if ( startTagResult != SKIP_BODY ) {
6          // process the BODY of the tag
7      }
8      if ( tagObj.doEndTag() == SKIP_PAGE ) {
9          return; // do not continue processing the JSP page
10     }
11 } finally {
12     tagObj.release();
13 }
```

## Writing an Iterator Custom Tag

Iteration is a common idiom in many programming situations. In an HTML page, iteration is used to generate an ordered list (OL tag), an unordered list (UL tag), or a collection of rows in a table. An iterator custom tag instructs the JSP page to process the body of the tag multiple times, until the iteration is complete.

An iteration (in general) requires five fundamental steps:

```
1 Iterator elements = getIterator(); // INITIALIZATION
2 while ( elements.hasNext() ) {      // ITERATION TEST
3     Object obj = elements.next();    // GET NEXT OBJECT
4     process(obj);                  // PROCESS THE OBJECT
5 }
```

// LOOP BACK UP

The doStartTag method is used to initialize the iteration, perform the first test, and (if the test passes) get the first object in the iteration. The JSP technology code of the body performs the “body processing.” A new method, doAfterBody, is used to perform subsequent iteration tests.

## Iteration Tag API

The tag handler API illustrated in “Fundamental Tag Handler API” on page 18-2 in Module 18, “Developing a Simple Custom Tag,” skipped the `IterationTag` interface for simplicity. The `IterationTag` interface extends the `Tag` interface and introduces the `doAfterBody` method and the `EVAL_BODY AGAIN` constant. The `TagSupport` class implements the `IterationTag` interface. The default implementation of the `doAfterBody` method returns the `SKIP_BODY` value. Therefore, by default, the JSP code of the tag body will only be processed once. To create an iteration tag handler, you will need to override the `doAfterBody` method. The `doAfterBody` method must return the `EVAL_BODY AGAIN` value to instruct the JSP page to iterate over the JSP body again. This extended tag handler API is illustrated in Figure 19-1.

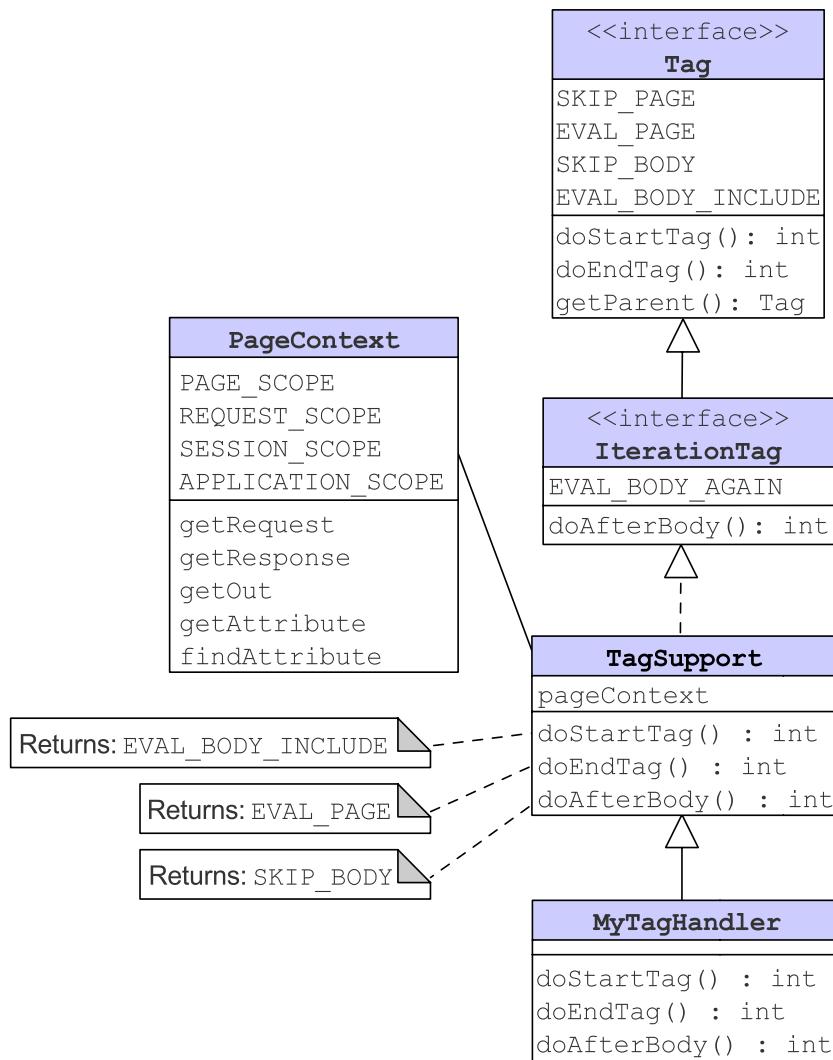


Figure 19-1 Iteration Tag API

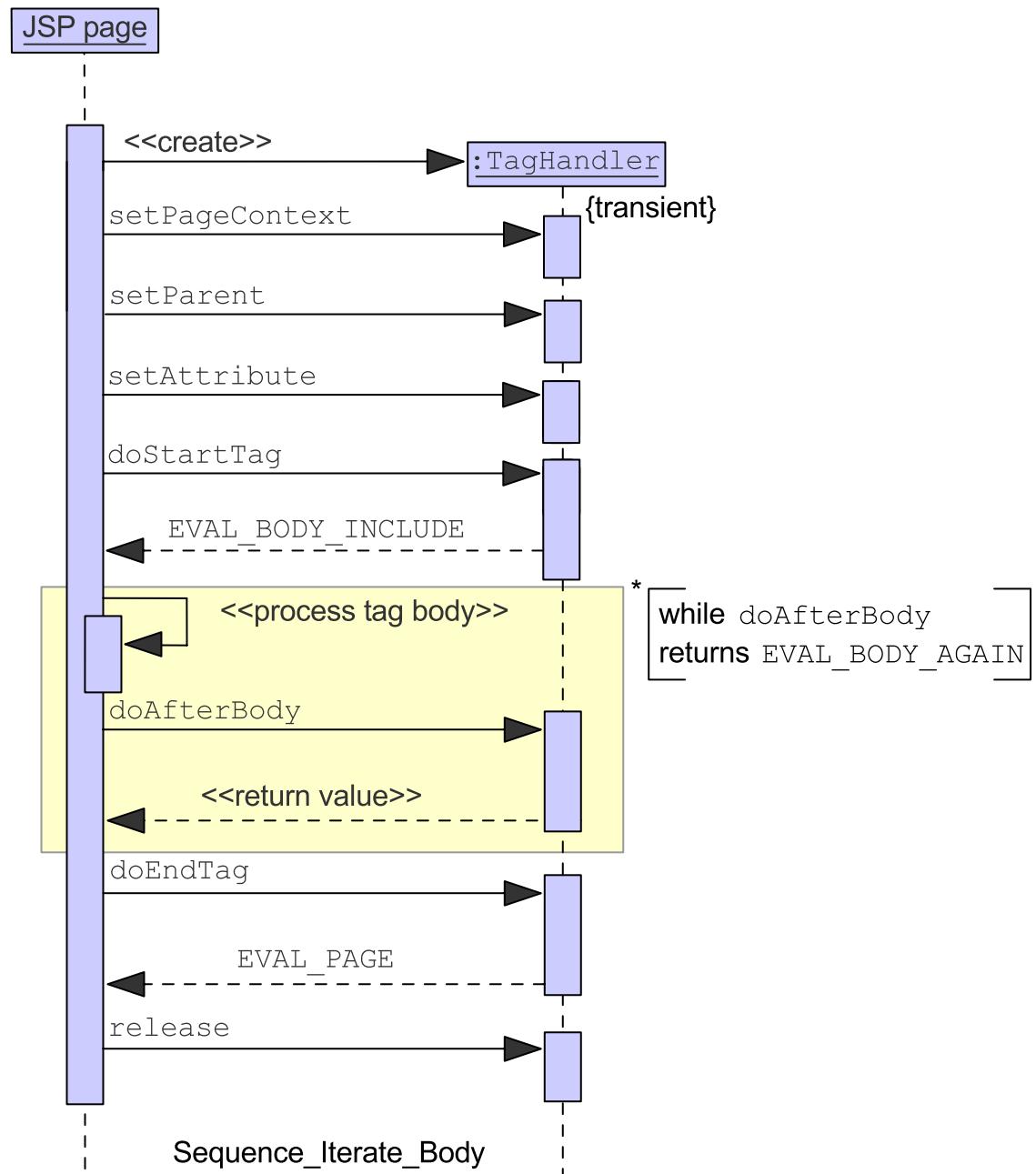
## Iteration Tag Life Cycle

To see how the `doAfterBody` method is used, look at the pseudo-code of a JSP page that uses a hypothetical iteration tag handler. If the `doStartTag` method returns `EVAL_BODY_INCLUDE`, then the JSP page enters a do-while loop that processes the tag body iteratively (but at least once) while the `doAfterBody` method returns the `EVAL_BODY_AGAIN` constant. This is shown in Lines 7–8 in Code 19-4.

**Code 19-4** Iteration Tag Life Cycle Pseudo-Code

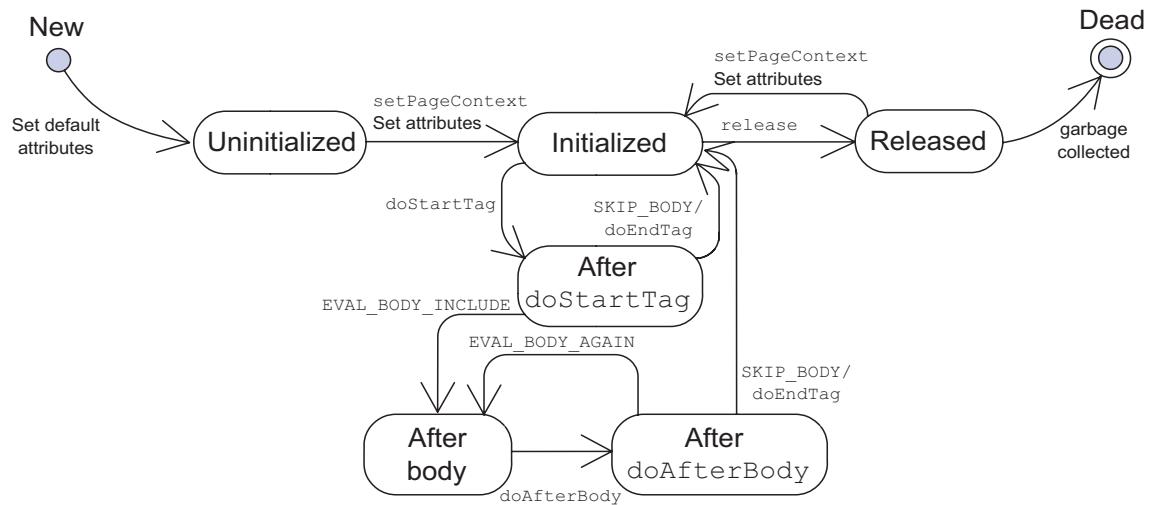
```
1 TagHandler tagObj = new TagHandler();
2 tagObj.setPageContext(pageContext);
3 tagObj.setAttr("value");
4 try {
5     int startTagResult = tagObj.doStartTag();
6     if ( startTagResult != SKIP_BODY ) {
7         do {
8             out.write("body"); // process the BODY of the tag
9         } while ( tagObj.doAfterBody() == EVAL_BODY_AGAIN );
10    }
11    if ( tagObj.doEndTag() == SKIP_PAGE ) {
12        return; // do not continue processing the JSP page
13    }
14 } finally {
15     tagObj.release();
16 }
```

A sequence diagram provides another perspective on the runtime interaction between the JSP page servlet and the iteration tag handler. This is illustrated in Figure 19-2.



**Figure 19-2** Sequence Diagram of the Iteration Tag Life Cycle

A state diagram provides another perspective on the runtime behavior of an iteration tag handler object. This is illustrated in Figure 19-3.



**Figure 19-3** State Diagram of an Iterator Tag Handler

## Example: The `iterateOverErrors` Tag

An example iteration custom tag is the `iterateOverErrors` tag in the Soccer League Web application. Here is the documentation for the `iterateOverErrors` custom tag:

This tag iterates over all of the exception objects in the `Status` object. This tag *must* be used within the `checkStatus` tag.

- Body content: This tag contains JSP technology code.
- Example:

```

<soccer:checkStatus>
  <soccer:iterateOverErrors>
    <%-- JSP code showing a single error message --%>
  </soccer:iterateOverErrors>
</soccer:checkStatus>
  
```

The `iterateOverErrors` tag is used within the `checkStatus` tag. The registration form used the `iterateOverErrors` tag to iterate over each error in the `Status` object and to generate an unordered list. This is shown in Code 19-5.

### Code 19-5 Using the `iterateOverErrors` Tag

```
16 <%-- BEGIN: error status presentation --%>
17 <soccer:checkStatus>
18 <FONT COLOR='red'>
19 There were problems processing your request:
20 <UL>
21   <soccer:iterateOverErrors>
22     <LI><soccer:getErrorMessage/>
23   </soccer:iterateOverErrors>
24 </UL>
25 </FONT>
26 </soccer:checkStatus>
27 <%-- END: error status presentation --%>
```

## The `iterateOverErrors` Tag Handler

The `IterateOverMessagesHandler` class implements the `iterateOverErrors` tag. It extends the `TagSupport` class that implements the `IterationTag` interface. The `IterateOverMessagesHandler` class must keep a copy of an `Iterator` object of the error set from the `Status` object (Line 22). This code is shown in Code 19-6.

### Code 19-6 The `IterateOverMessagesHandler` Class

```
11 /**
12  * This class handles the "iterate over all status exceptions" tag.
13  * This tag handler assumes that it is the child of the "check status"
14  * tag handler. This assumption allows this tag handler to assume
15  * that there is at least one exception in the set of errors in the
16  * Status object. On each iteration, the "current exception object"
17  * is stored within the PAGE scope for use by the "get current error
18  * message" tag.
19 */
20 public class IterateOverMessagesHandler extends TagSupport {
21
22     private Iterator errors;
23 }
```

The `doStartTag` method is responsible for initializing the iteration. The `doStartTag` method code is shown in Code 19-7. The `Status` object is retrieved from the request scope using the `getAttribute` method on the `pageContext` object (Lines 25–27). The `errors` instance variable (which holds the `Iterator` object) is initialized from the `Status` object (Line 30). Next, the first element in the iteration (retrieved by the `next` method) is stored as an attribute in the page scope (Lines 33–35). Finally, the `doStartTag` method returns the `EVAL_BODY_INCLUDE` value to tell the JSP page to process the tag body at least once (Line 37).

**Code 19-7**      The `doStartTag` Method

```
24 public int doStartTag() {
25     Status status
26     = (Status) pageContext.getAttribute("status",
27             PageContext.REQUEST_SCOPE);
28
29     // Initialize the iterator
30     errors = status.getExceptions();
31
32     // Store the first error in the PAGE scope
33     pageContext.setAttribute("iterateOverErrors.currentError",
34             errors.next(),
35             PageContext.PAGE_SCOPE);
36
37     return EVAL_BODY_INCLUDE;
38 }
39
```

The `doAfterBody` method is responsible for performing the iteration test. The `doAfterBody` method is shown in Code 19-8. The `hasNext` method is used on the `Iterator` object held in the `errors` instance variable (Line 43). This method returns false if there are no more elements in the iteration. When that occurs the `doAfterBody` method returns the `SKIP_BODY` value. This value ends the iteration. Otherwise, the `doAfterBody` method must store the next element in the iteration (Lines 46–48) and then return the `EVAL_BODY AGAIN` value. This value tells the JSP servlet to process the JSP technology code in the tag body again. The `doAfterBody` method is called after each evaluation of the tag body, until the iteration has come to an end.

### Code 19-8 The `doAfterBody` Method

```
40 public int doAfterBody() {
41
42     // Is there another element to iterate over?
43     if ( errors.hasNext() ) {
44
45         // If yes, then store the next error in the PAGE scope,
46         pageContext.setAttribute("iterateOverErrors.currentError",
47                             errors.next(),
48                             PageContext.PAGE_SCOPE);
49
50         // and tell the JSP interpreter to evaluate the body again
51         return EVAL_BODY AGAIN;
52
53     } else {
54         // If no, tell the JSP run-time that we are done iterating
55         return SKIP_BODY;
56     }
57 }
```

## Using the Page Scope to Communicate

The `iterateOverErrors` tag and the `getErrorMessage` tag must communicate. The `iterateOverErrors` tag must store the current `Exception` object in the iteration. The `getErrorMessage` tag uses that object to retrieve the text message of the error that will be reported to the user. The `iterateOverErrors` tag handler stores the `Exception` object in a page-scoped attribute called `iterateOverErrors.currentError`. Therefore, the `getErrorMessage` tag can retrieve that attribute and print the error message to the HTTP response stream. This code is shown in Code 19-9.

**Code 19-9**      The `GetCurrentErrorMsgHandler` Class

```
21 public class GetCurrentErrorMsgHandler extends TagSupport {  
22  
23     public int doStartTag() throws JspException {  
24         JspWriter out = pageContext.getOut();  
25  
26         // Retrieve the current exception object from the PAGE scope.  
27         Exception exc  
28         = (Exception)  
29             pageContext.getAttribute("iterateOverErrors.currentError",  
30                             PageContext.PAGE_SCOPE);  
31  
32         try {  
33             // Print out the message of the exception object.  
34             out.print(exc.getMessage());
```

## Notes

## Summary

This module presented conditional and iterative custom tag handlers.

You can create a conditional tag by making the decision to return either the SKIP\_BODY or the EVAL\_BODY\_INCLUDE values from the doStartTag method.

An iteration tag requires coordination between the doStartTag and doAfterBody methods:

- The doStartTag method initializes the iteration, retrieves the first element (if any), and determines whether to process the body if there is an element.
- The doAfterBody method determines whether to process the body again (returning the EVAL\_BODY AGAIN value) or skipping the body if the iteration is done.

Tag handlers can communicate using an attribute in the page scope.

## Certification Exam Notes

This module presented the following objectives for Section 12, “Designing and Developing A Custom Tag Library,” of the Sun Certification Web Component Developer certification exam:

12.4 Given a tag event method (`doStartTag`, `doAfterBody`, and `doEndTag`), identify the correct description of the methods trigger.

12.5 Identify valid return values for the following methods:

- The `doStartTag` method
- The `doAfterBody` method
- The `doEndTag` method
- The `PageConext.getOut` method

12.6 Given a “BODY” or “PAGE” constant, identify a correct description of the constant’s use in the following methods:

- The `doStartTag` method
- The `doAfterBody` method
- The `doEndTag` method

12.7 Identify the method in the custom tag handler that accesses:

- A given JSP implicit variable
- The JSP page’s attributes

# Integrating Web Applications With Enterprise JavaBeans Components

---

## Objectives

Upon completion of this module, you should be able to:

- Understand the Java 2 Platform, Enterprise Edition, (J2EE) at a high level
- Develop a Web application that integrates with an Enterprise JavaBeans (EJB) component using the Business Delegate pattern

## Relevance



**Discussion** – The following questions are relevant to understanding why you would choose a J2EE platform solution:

- What are the costs of having the business logic on the Web tier?
  
- What technologies can be used to distribute the business logic onto another physical tier?
  
- What benefits does an J2EE platform solution provide?

## Additional Resources

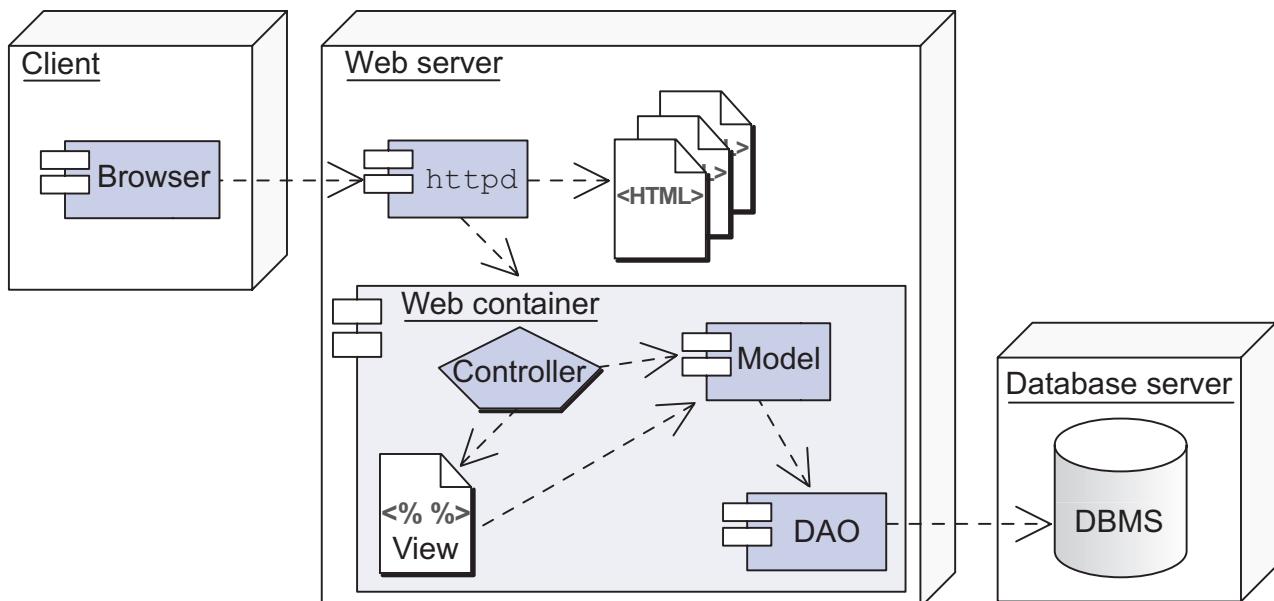


**Additional resources** – The following references provide additional information on the topics described in this module:

- *Enterprise JavaBeans Technology*. [Online]. Available: <http://java.sun.com/products/ejb/>
- *Java™ 2 Platform, Enterprise Edition*. [Online]. Available: <http://java.sun.com/j2ee/>
- *Java™ 2 Platform, Enterprise Edition Blueprints*. [Online]. Available: <http://java.sun.com/j2ee/blueprints/>
- Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns*. Upper Saddle River: Prentice Hall PTR, 2001.
- *Sun Java Center J2EE Patterns*. [Online]. Available: <http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/>

# Distributing the Business Logic

The Web applications that you have seen so far place all of the presentation logic, business logic, and data access logic on the physical Web server. This architecture is illustrated in Figure 20-1.



**Figure 20-1** A Physical 3-Tier Architecture

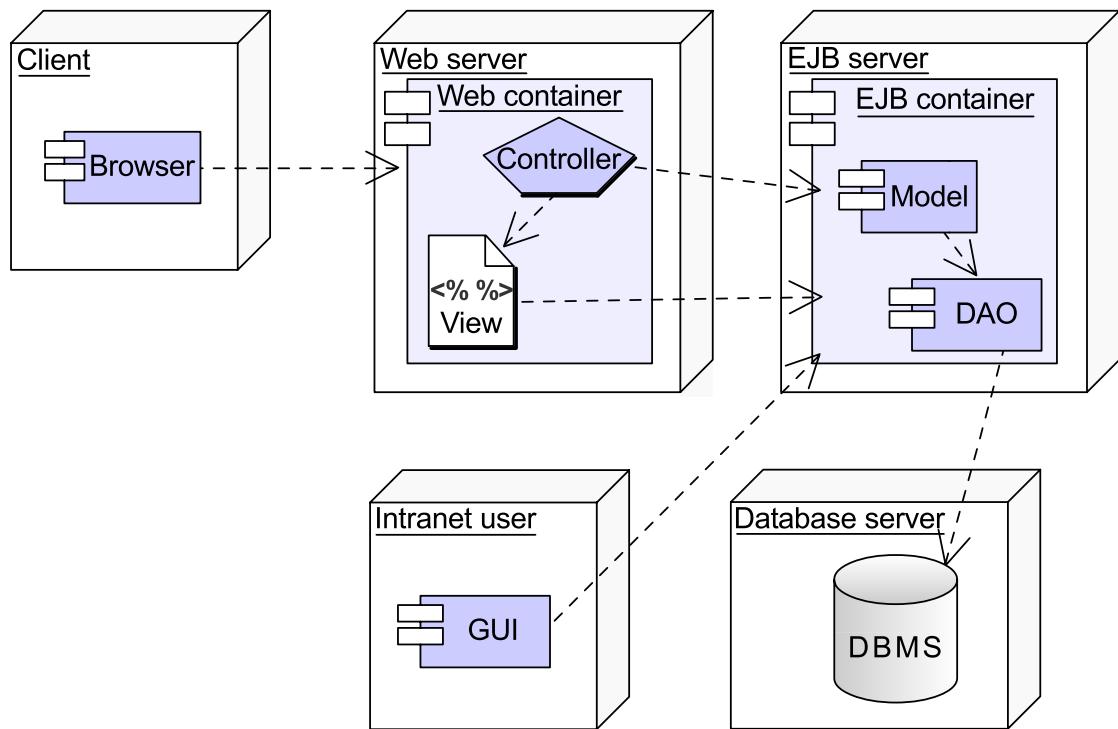
There are two main problems with this architecture. First, having the business logic and presentation logic on the same physical host increases the performance demands on that server. A related issue is that the hardware for a Web server might need to focus on I/O bandwidth, whereas the hardware for business logic is more CPU bound.

Second, if your project requires a standalone GUI application, you will have to duplicate the Model and DAO elements in that application. This might lead to design complications relative to transaction management.

One solution to both of these problems is to distribute the business and data access logical tiers onto a physical tier separated from the Web tier and any other standalone client tiers. This provides scalability enhancements as well as centralized transaction control.

## Java 2 Platform, Enterprise Edition

The Java 2 Platform, Enterprise Edition, includes technologies to facilitate the distribution of the business logic of an enterprise application. The main technology for constructing business logic components uses Enterprise JavaBeans (EJB) components. Using an EJB technology server, you can distribute your business logic onto a separate physical tier. This 4-tier architecture is illustrated in Figure 20-2.



**Figure 20-2** A Physical 4-Tier Architecture

## Enterprise JavaBeans Technology

Enterprise JavaBeans technology provides two main types of business logic components: session and entity beans. Session beans are used to encapsulate business services. Entity beans are used to encapsulate business domain objects, such as entities that exist in a data store (for example, rows in a relational database).



**Note** – Enterprise JavaBeans components are not related to JavaBeans components. JavaBeans components are basically Java technology classes that follow certain coding conventions (such as the “property naming” conventions of the accessor and mutator methods, making all instance variables private, and having a no-argument constructor). Enterprise JavaBeans components exist in an EJB container which provides services to the EJB components. EJB components are used to model business objects and services.

---

A detailed explanation of EJB technology is beyond the scope of this course.

# Integrating the Web Tier With the EJB Tier

This module does not describe how EJB components are created. It does describe how to integrate the Web tier to the EJB tier. There are three high-level steps in this process:

1. Business Component Developers publish the EJB interfaces to the application Model.

The job role of Business Component Developers is new for the J2EE platform. Business Component Developers are responsible for creating the business EJB components in an enterprise application. A product of this work is the Java technology interfaces that hide the bean implementation details. The client code that integrates to an EJB tier may only use the methods in these interfaces.

2. At deployment, names are given to the enterprise beans and the Java Naming and Directory Interface™ (JNDI) host is identified.

These names identify the EJB components. JNDI is used to provide a global naming service to look up and retrieve remote references to enterprise beans.

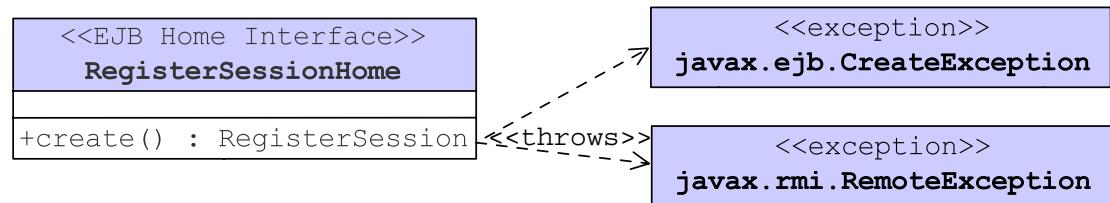
3. The Web Component Developers create servlets that access the enterprise beans by using a delegate to hide the complexity of integrating with the EJB tier.

The Business Delegate pattern is used to hide the implementation details of integrating the Web tier with the EJB tier. The servlet Controller interacts with the business delegate object, which delegates the method calls to the enterprise beans.

## Enterprise JavaBeans Interfaces

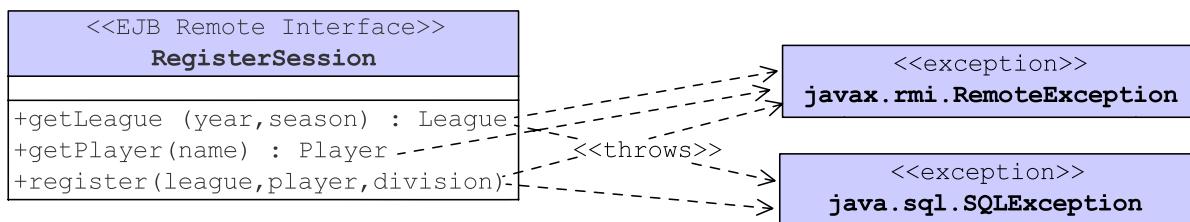
EJB session beans are used to encapsulate business services. For example, in the Soccer League application the RegisterService class can be migrated to a session bean, called RegisterSession. The EJB specification requires two interfaces: a Home interface and a Remote interface.

The EJB Home interface specifies methods for creating new or looking up existing enterprise beans. For example, the RegisterSessionHome interface provides the create method, which returns a remote reference to the actual session bean. The RegisterSessionHome interface is illustrated in Figure 20-3.



**Figure 20-3** An EJB Home Interface

The EJB Remote interface specifies the business logic methods of the remote service. For example, the RegisterSession interface provides the getLeague, getPlayer, and register methods that had previously existed in the RegisterService class. The RegisterSession interface is illustrated in Figure 20-4.



**Figure 20-4** An EJB Remote Interface

## Java Naming and Directory Service

JNDI is a global naming service. The EJB container publishes the names of the enterprise beans to the JNDI server. This is called binding. More specifically, the objects that implement the EJB Home interfaces are bound to the names. The Business Delegate object uses JNDI to retrieve the enterprise bean Home interfaces. This is called a lookup. The object returned by JNDI communicates with the EJB server to create (or find) the required enterprise bean. The bind and lookup processes are illustrated in Figure 20-5.

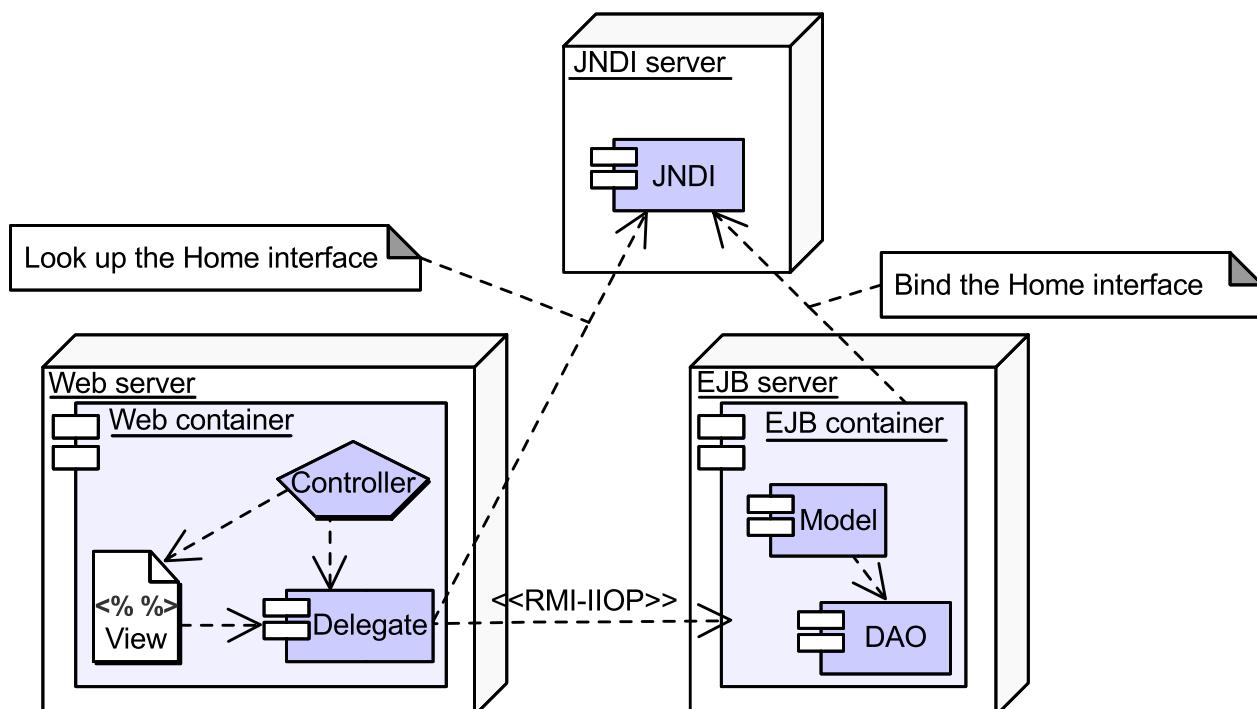


Figure 20-5 JNDI as a Naming Service for the EJB Tier

## Creating the Business Delegate

When you create a delegate to the EJB tier, the servlets interact with the delegate very much like they did with the business service objects. The delegate is used to hide the complexity of integrating with the EJB tier from the servlet.

The delegate must:

- Use JNDI to look up the Home interface
- Use the Home interface to create the enterprise bean object
- Call business methods on the enterprise bean object
- Remove the enterprise bean when the business method has completed its work

## Declaring the Business Delegate Class

In the Soccer League application, a Business Delegate can be constructed to hide the integration details of using an enterprise bean. For example the RegisterDelegate class hides the integration with the RegisterSession bean. The declaration of the RegisterDelegate class is shown in Code 20-1.

**Code 20-1** An Example Business Delegate Class

```

1 package sl314.web;
2
3 // Domain imports
4 import sl314.ejb.League;
5 import sl314.ejb.Player;
6 import sl314.ejb.RegisterSession;
7 import sl314.ejb.RegisterSessionHome;
8 // Remote service imports
9 import javax.naming.InitialContext;
10 import javax.naming.Context;
11 import javax.rmi.PortableRemoteObject;
12 import java.rmi.RemoteException;
13 import javax.ejb.CreateException;
14 import java.sql.SQLException;
15
16 /**
17 * This object performs a variety of league registration services.
18 * It acts a Delegate to the RegisterSession Enterprise bean.
19 */
20 public class RegisterDelegate {

```

This class requires the services of JNDI, EJB, and Remote Method Invocation (RMI); you need to import several classes (Lines 9–13). You also need to import the domain object classes (Lines 4 and 5) and the enterprise bean Home and Remote interfaces (Lines 6 and 7).



**Note** – The domain objects that are transmitted to and from the enterprise beans are called Value Objects. For example, the League class is a Value Object that holds the data about a soccer league (year, season, and title). A Value Object that has mutator methods is often called an Updateable Value Object. The Player class is an Updateable Value Object.

## Finding the Home interface Using JNDI

The Business Delegate should store the enterprise bean Home interface for use in delegating the business method calls to the enterprise bean. For example, the RegisterDelegate class has an instance variable to store the RegisterSession bean's Home interface (Line 25). The constructor for the RegisterDelegate class uses JNDI to look up the RegisterSession bean's Home interface by the name registerService (Line 33). The lookup method returns an Object which must be converted to the Home interface object using the narrow method (Lines 34–36). This code is shown in Code 20-2.

**Code 20-2** Use JNDI to Look Up the Home Interface

```
20 public class RegisterDelegate {  
21  
22     /**  
23      * The EJB remote object for the RegisterSession service.  
24      */  
25     private RegisterSessionHome registerSvcHome;  
26  
27     /**  
28      * This constructor creates a Registration Delegate object.  
29      */  
30     public RegisterDelegate() {  
31         try {  
32             Context c = new InitialContext();  
33             Object result = c.lookup("ejb/registerService");  
34             registerSvcHome  
35                 = (RegisterSessionHome)  
36                     PortableRemoteObject.narrow(result, RegisterSessionHome.class);  
37         } catch (Exception e) {  
38             e.printStackTrace(System.err);  
39         }  
40     }  
41 }
```

## Delegating Business Methods to the EJB Tier

The business methods in the Business Delegate object use the enterprise bean Home interface object to create the remote bean. The business method then delegates to the enterprise bean using a remote method call. For example, the getLeague method of the RegisterDelegate class uses the RegisterSession bean's Home interface to create a RegisterSession remote object (Line 53). The getLeague method is delegated to the remote object (the session bean on the EJB server, line 54). Finally, the remote object is destroyed by calling the remove method on the remote object (Line 58). The code for the getLeague method of the RegisterDelegate class is shown in Code 20-3.

**Code 20-3** Delegating Business Methods to the EJB Tier

```
46 public League getLeague(String year, String season)
47     throws CreateException, RemoteException, SQLException {
48     League league = null;
49     RegisterSession registerSvc = null;
50
51     // Delegate to the EJB session bean
52     try {
53         registerSvc = registerSvcHome.create();
54         league = registerSvc.getLeague(year, season);
55
56     } finally {
57         try {
58             if ( registerSvc != null ) registerSvc.remove();
59         } catch (Exception e) {
60             e.printStackTrace(System.err);
61         }
62     }
63
64     return league;
65 }
```

## Using the Business Delegate in a Servlet Controller

This section describes how to develop a servlet that uses a Business Delegate object to handle the business logic. The servlet class must:

- Create the delegate object
- Execute business methods on the delegate
- Handle any exceptions dealing with remote access to EJB components in the try-catch block

## Using a Delegate in a Servlet

In the Soccer League RegistrationServlet class, the processRequest method creates an instance of the RegisterDelegate class (Line 92) and then uses the delegate to call the business methods to process the registration form. First, the league object is retrieved (Line 95). Next, the player object is retrieved (Line 108). Finally, the register method is called to complete the registration process (Line 114). This code is shown in Code 20-4.

**Code 20-4** Using the Business Delegate

```

89 // Now delegate the real work to the RegisterDelegate object
90 try {
91     // Create a "registration delegate" object
92     registerDlg = new RegisterDelegate();
93
94     // Retrieve the league object
95     league = registerDlg.getLeague(year, season);
96     // and verify that it exists
97     if ( league == null ) {
98         // If not, then forward to the Error page View
99         status.addException(
100             new Exception("The league you selected does not yet exist;" +
101                         + " please select another."));
102         responsePage = request.getRequestDispatcher("/form.jsp");
103         responsePage.forward(request, response);
104         return;
105     }
106
107     // Create and populate the player object
108     Player player = registerDlg.getPlayer(name);
109     player.setAddress(address);
110     player.setCity(city);
111     player.setProvince(province);
112     player.setPostalCode(postalCode);
113
114     registerDlg.register(league, player, division);
115     request.setAttribute("league", league);
116     request.setAttribute("player", player);
117
118     // Forward to the next View: the "Thank You" page
119     responsePage = request.getRequestDispatcher("/thank_you.jsp");
120     responsePage.forward(request, response);
121     return;
122

```

As usual, all of the business method calls in the servlet are wrapped in a try-catch block (Lines 90–122). Using the Business Delegate pattern introduces a few more EJB and RMI related exceptions to handle. This code is shown in Code 20-5.

### Code 20-5      Exception Handling Code

```
123 // Handle any remote exceptions
124 } catch (CreateException ce) {
125     status.addException(
126         new Exception("Could not create the EJB Session object.<BR>" +
127             + ce.getMessage()));
128
129 // Handle any remote exceptions
130 } catch (RemoteException re) {
131     status.addException(
132         new Exception("An EJB error occurred.<BR>" +
133             + re.getMessage()));
134
135 // Handle any SQL exceptions
136 } catch (SQLException se) {
137     status.addException(
138         new Exception("A remote SQL exception occurred.<BR>" +
139             + se.getMessage()));
140
141 // Clean up
142 } finally {
143     // Nothing to do because GC will clean up the registerDlg object
144 }
145
146 // If we get there, then there was an error in the try-catch block
147 // Forward to the Error page View.
148 responsePage = request.getRequestDispatcher("/form.jsp");
149 responsePage.forward(request, response);
150 return;
151 }
```

# Summary

This module presented the integration of the Web tier with the EJB tier using the Business Delegate pattern. On the EJB tier, the Business Component Developer publishes the Home and Remote interfaces. On the Web tier, the Web Component Developer creates a Business Delegate to the EJB tier and modifies the servlets to interact with that delegate.

The Business Delegate object must:

- Use JNDI to retrieve the Home interface
- Use the Home interface to create the enterprise bean object
- Call business methods on the enterprise bean object
- Remove the enterprise bean when the business method has completed its work

## Certification Exam Notes

This module presented some of the objectives for Section 13, "Web Tier Design Patterns," of the Sun Certification Web Component Developer certification exam:

- 13.1 Given a scenario description with a list of issues, select the design pattern (*Value Objects*, MVC, Data Access Object, or *Business Delegate*) that would best solve those issues.
- 13.2 Match design patterns with statements describing potential benefits that accrue from the use of the pattern, for any of the following patterns: *Value Objects*, MVC, Data Access Object, *Business Delegate*.

The MVC pattern is presented in Module 15, "Developing Web Applications Using the Model 2 Architecture," and the Data Access Object pattern is presented in Module 12, "Integrating Web Applications With Databases."

# Quick Reference for HTML

---

## Objectives

Upon completion of this appendix, you should be able to:

- Define HTML and markup language
- Explain the difference between semantic and physical markup
- Create a simple HTML document that illustrates use of comments, lists, headings, and text formatting elements
- Explain how to create a link to a separate document or to somewhere within the same document
- Describe how to include an image or an applet in an HTML document
- Identify the four main elements used in creating HTML forms
- Create a basic HTML table
- Describe the main purpose for JavaScript, Cascading Style Sheets (CSS), and frames

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

- Graham, Ian S. *HTML 4.0 Sourcebook*. New York: John Wiley & Sons, Inc., 1998.
- JavaScript Tutorials. [Online]. Available at:  
<http://www.wsabstract.com/> and  
<http://www.javascript.com/>.
- Cascading Style Sheets. [Online]. Available at:  
<http://www.w3.org/Style/CSS/>.
- HTML 4.01 Specification. [Online]. Available at:  
<http://www.w3.org/TR/html4/>.
- Dave Raggett's Introduction to HTML. [Online]. Available at:  
<http://www.w3.org/MarkUp/Guide/>.

# HTML and Markup Languages

This section provides a brief overview of markup languages and the Hypertext Markup Language.

## Definition

HTML stands for Hypertext Markup Language. A markup language is a language with specialized markings (or tags) that you embed in the text of a document to provide instructions or to indicate what the text is supposed to look like.

HTML was developed for marking up documents for delivery over the Internet and is closely associated with Web browsers. Hypertext links within HTML documents enable you to move from one document to another.

## Types of Markup

Markup can be categorized as one of two types:

- Physical markup – The markup tags indicate how the text that is marked will look.
- Logical (or semantic) markup – The markup tags define the structural meaning of the text that is marked and do not specify how the text is to look.

## Simple Example

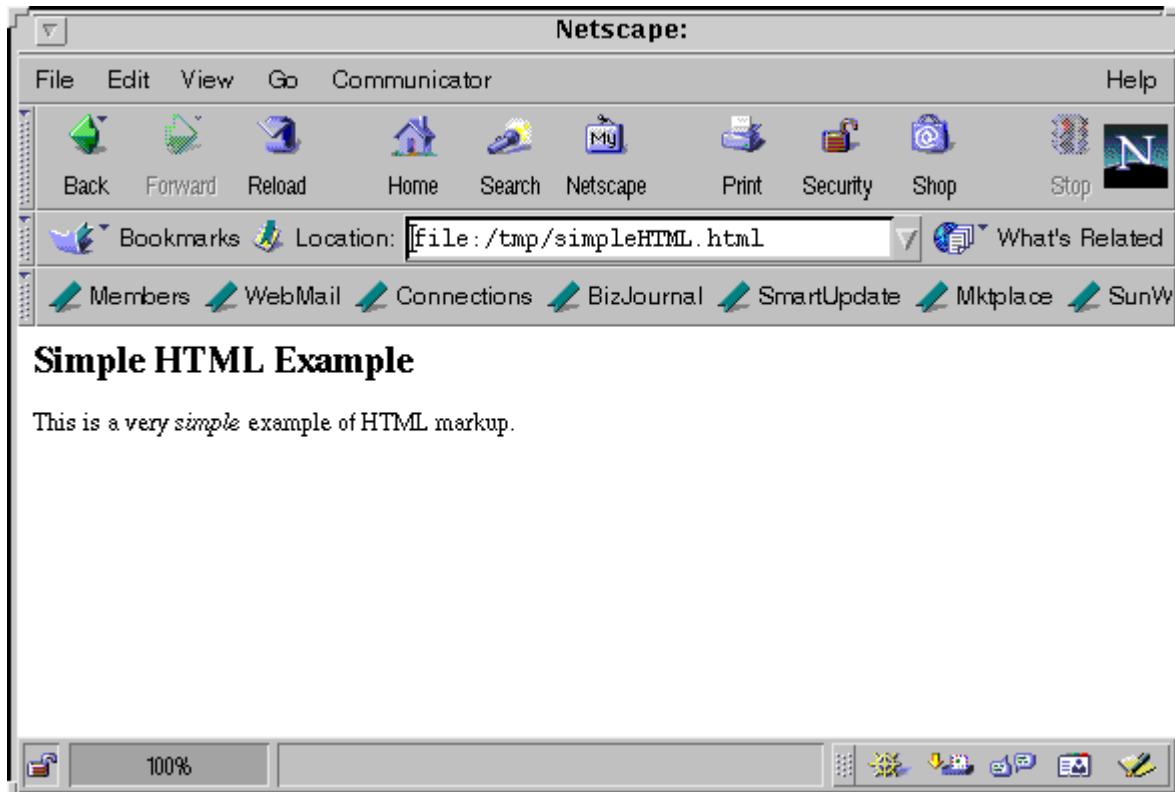
In HTML, the markup is specified within a pair of angle brackets (< >). These markings are referred to as *tags*. Figure A-1 shows a simple HTML file.

```
<HTML>
<BODY BGCOLOR="#ffffff">
<H1>Simple HTML Example</H1>
This is a very <I>simple</I> example of HTML
markup.
</BODY>
```

**Figure A-1** Simple HTML File

Most HTML documents contain the tag <HTML> to indicate the type of document you are looking at. The <BODY> tag is used around the entire document text that will be displayed in the browser window.

Figure A-2 displays this HTML file in a browser.



**Figure A-2** Browser Display of Simple HTML File

# Basic Structure of HTML

This section describes the basic syntactic structure of HTML documents.

## Tag Syntax

From the previous example, you can see that an HTML document consists of text that is interspersed with tags of the form `<NAME>` and `</NAME>`. The tag `<NAME>` is referred to as the *start tag*, and the tag `</NAME>` as the *end tag*.

### Elements

Tags are also called *elements* in HTML documents. So, for example, the HTML file displayed in Figure A-1 contains the following elements: HTML, BODY, H1, and I.

### Attributes

Some elements can be further described using attributes. An attribute is included within the pair of angle brackets for the element and uses the following syntax:

`attribute_name="attribute_value"`

In the HTML file displayed in Figure A-1, the element BODY has an attribute called BGCOLOR, which sets the color of the background in the browser to white when this HTML file is displayed.

---

**Note** – HTML element and attribute names are not case sensitive. All elements and attributes shown in this appendix will use capital letters.



## Comments

You can include comments in your HTML file using the syntax

`<!-- put your comment here -->`

The string `<!--` begins a comment and the string `-->` ends a comment.

## Spaces, Tabs, and Newlines Within Text

Browsers that display HTML documents ignore extra spaces, tabs, blank lines, and newlines in the HTML document. That is, the occurrence of any of these is treated as a single space by browsers. If you need to specify these, you must use an appropriate HTML tag (element) to accomplish what you want to do.

## Character and Entity References

HTML supports character sets for international use as specified by the International Standards Organization (ISO). For the World Wide Web (WWW), this set of characters is ISO Latin-1 (or ISO 8859-1).

You can represent any ISO Latin-1 character with a *character* reference (its decimal code). Some characters can also be represented using their *entity* reference. The entity reference is typically used in an HTML document for the characters '<', '>', '&', and '"' because these characters have special meaning in HTML syntax. Table A-1 shows the decimal code and entity references for some typical characters.

**Table A-1** Decimal Code and Entity References

Character	Decimal Code	Entity Reference
<	60	&lt;
>	62	&gt;
&	38	&amp;
"	34	&quot;
a space	160	&nbsp;

Thus, if you needed to have the characters <HTML> actually displayed in a browser, you should enter the following in your HTML document:

&lt;HTML&gt;

To refer to the character < in HTML using the decimal code, use the syntax &#60; that is, you place the decimal code between &# and a semi-colon (;).

# Links and Media Tags

The HTML specification provides elements that enable you to create hyperlinks to other documents or to another location of the same document, create inline images, and run external programs (applets).




---

**Note** – The `IMG` and `APPLET` elements are somewhat subsumed in HTML 4.0 by the new `OBJECT` element. A description of each of these is included here for completeness.

---

## The `HREF` Attribute and the `A` Element

The `A` element (referred to as the *anchor* element) and its `HREF` attribute are used to create a hypertext link. The syntax is:

```
<A HREF="reference_to_somewhere">the displayed link</A>
```

The *reference\_to\_somewhere* can be any of the following:

- URL, such as `http://w3.org/xxx.html`
- Path name relative to the location of the current file, such as `example.html` or `../samples/sample1.html`
- Fragment identifier, such as `#named_location`, where the string *named\_location* is the value of a `NAME` attribute specified previously in the current document. For example, if the following line occurred previously in an HTML document,

```
<A NAME="tagSyntax">
```

then you can create a link to this location using the following hyperlink:

```
<A HREF="#tagSyntax">tag syntax</A>
```

The text specified between the start tag `<A HREF="reference_to_somewhere">` and the end tag `</A>` is the link that the browser displays. Browsers display hyperlinks in HTML documents as underlined text. The previous example would display “tag syntax” as underlined to indicate that it is a link to somewhere else. Clicking the link takes you to the location referred to by the link.

## The `IMG` Element and the `SRC` Attribute

The `IMG` element (image element) is used in an HTML document to indicate that an image is to be included (displayed by the browser).

```
<IMG SRC="URL_of_image">
```

The value of the attribute `SRC` specifies the actual image to display. By default, the image occurs inline with the text. There are other attributes of the `IMG` element that can be used to control the display of the image and the text around it. These include:

- `ALIGN` – Specifies the alignment of the image with the text around it. Values include “top,” “bottom,” “middle,” “left,” and “right.”
- `HEIGHT` – Specifies the height of the image in integer units (pixels by default).
- `WIDTH` – Specifies the width of the image in integer units (pixels by default).

## The `APPLET` Element

The `APPLET` element is used in an HTML document to indicate that a Java technology applet is to be downloaded and run. The usual syntax is:

```
<APPLET  
    CODE="appletClass.class"  
    WIDTH="200"  
    HEIGHT="200"  
>  
</APPLET>
```

This syntax assumes that the applet `.class` file is in the same directory as the HTML document. If this is not the case, the `CODEBASE` attribute is used to specify the URL or directory containing the class file.

## The OBJECT Element

The OBJECT element is new as of HTML 4.0 and is used wherever you would use an SRC or APPLET element (and will be used for implementing future media types).

---

**Note** – The APPLET element is now considered deprecated.

---



According to the HTML 4.0 specification, the following OBJECT element can be used to replace the previous APPLET element:

```
<OBJECT
    CODETYPE="application/java"
    CLASSID="appletClass.class"
    WIDTH="200"
    HEIGHT="200"
>
</OBJECT>
```

However, certain browser implementations might not support this. Also, if you are using the Java technology plug-in there may be a conflict with the use of CLASSID. Then the following is recommended.

```
<OBJECT
    CODETYPE="application/java"
    WIDTH="200"
    HEIGHT="200"
>
<PARAM name="code" value="appletClass.class">
</OBJECT>
```

Common attributes used with the OBJECT element include:

- ARCHIVE – A space-separated list of URLs
- CODEBASE – Base URL for CLASSID, DATA, and ARCHIVE
- CODETYPE – Indicates the content type for code
- DATA – A URL to refer to the object's data
- TYPE – Indicates the content type for data (such as "application/java")
- HEIGHT – Overrides the height
- WIDTH – Overrides the width

# Text Structure and Highlighting

This section describes the HTML tags for structuring text and highlighting text.

## Text Structure Tags

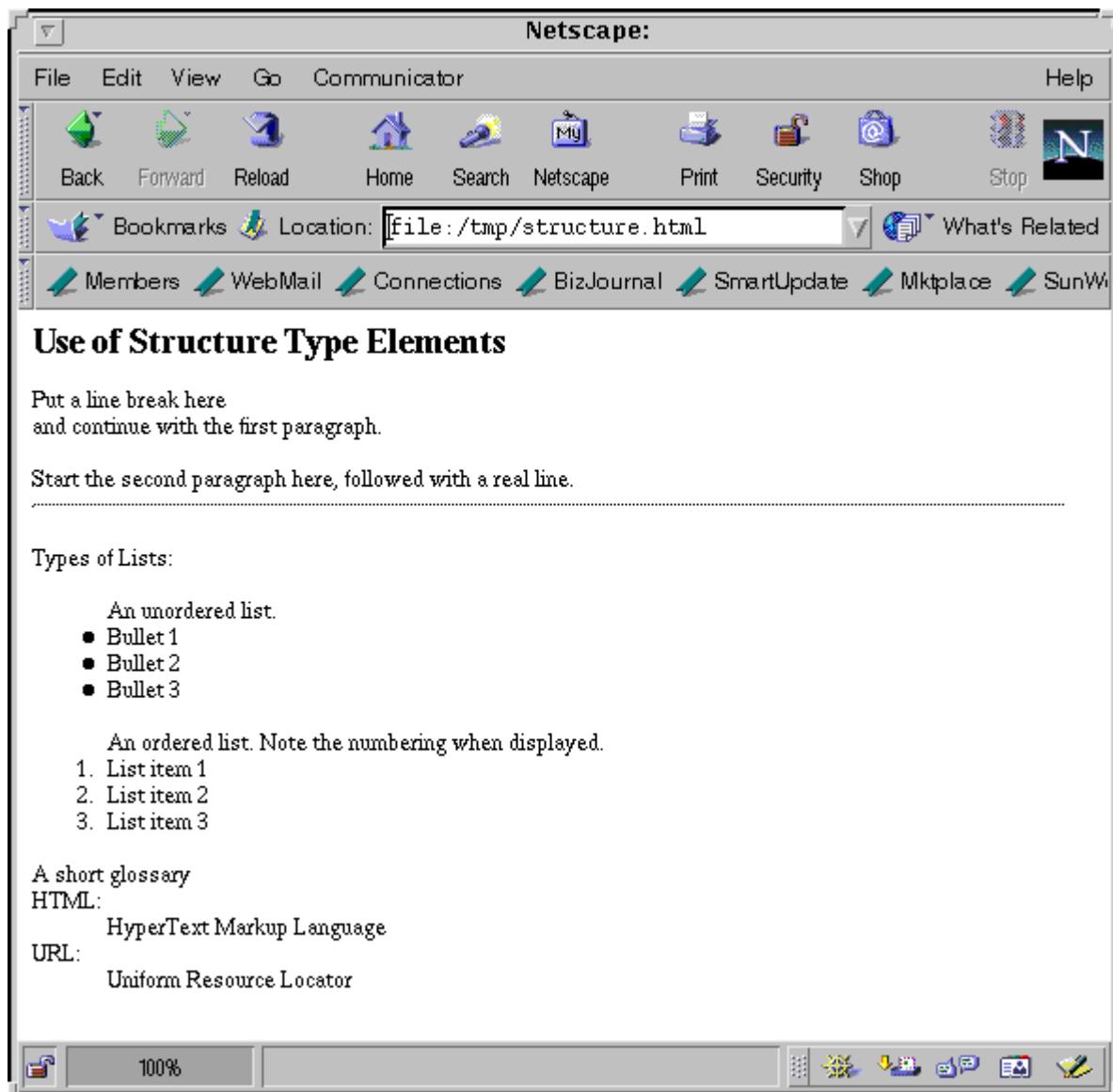
HTML provides the following elements to control the structure of your document:

- P – Indicates the beginning of a paragraph and is used to create logical blocks of text
- BR – Indicates a line break and is used to force a newline in the display by browsers
- HR – Indicates a horizontal rule, which displays as a horizontal line across the screen
- DIV – Indicates a block of text in a document that is to be treated as one logical group or division
- UL – Indicates an unordered list element, and usually contains the sub-element LI
- OL – Indicates an ordered list element. and usually contains the sub-element LI
- DL – Indicates a glossary list element, and usually contains sub-elements DT and DD

Figure A-3 shows the use of these elements in an HTML document. This HTML document is displayed in a browser in Figure A-4 on page A-12.

```
<HTML>
<BODY BGCOLOR="#ffffff">
<H1>Use of Structure Type Elements</H1>
Put a line break here <BR>
and continue with the first paragraph.
<P>
Start the second paragraph here, followed with a real line.
<HR>
<P>
<DIV>Types of Lists:
<UL>An unordered list.
    <LI>Bullet 1</LI>
    <LI>Bullet 2</LI>
    <LI>Bullet 3</LI>
</UL>
<P>
<OL>An ordered list. Note the numbering when displayed.
    <LI>List item 1</LI>
    <LI>List item 2</LI>
    <LI>List item 3</LI>
</OL>
<P>
<DL>A short glossary
    <DT>HTML:</DT>
        <DD>HyperText Markup Language</DD>
    <DT>URL:</DT>
        <DD>Uniform Resource Locator</DD>
</DL>
</DIV>
</BODY>
```

**Figure A-3** Structure Type Tags in an HTML Document



**Figure A-4** Display of Structure Type HTML Elements

## Text Highlighting

HTML provides elements that can be used to emphasize (special meaning) or to highlight text in some visual way. The elements provided can be categorized as *semantic* or *physical* elements. Semantic (or logical) text elements use the name of the element to indicate the type of text that it tags, such as a piece of computer code, a variable, or a citation or quote. Physical elements indicate that a specific physical format be used, such as boldface or italics.

### Semantic Elements

The following are typical semantic elements:

- CITE – Indicates a citation or quote; usually rendered in italics
- CODE – Indicates a piece of code; usually rendered in fixed-width font
- EM – Indicates text is to be emphasized; usually rendered in italics
- KBD – Indicates keyboard input; usually rendered in fixed-width font
- SAMP – Indicates a sequence of literal characters
- STRONG – Indicates strong emphasis; usually rendered in boldface
- VAR – Indicates a variable name; usually rendered in italics

### Physical Elements

The following are typical physical elements:

- B – Display as boldface.
- I – Display as italics.
- TT – Display as fixed-width typewriter font.
- U – Display as underlined. (Not recommended because it can be confused by the user as a hypertext link.)

# HTML Forms

This section describes the basic syntactic structure of HTML forms.

## The FORM Tag

In an HTML page, the FORM tag acts as a container for a specific set of GUI components. The GUI components are specified with input tags. There are several varieties of input tags, which are shown in the next few sections. An example HTML form is shown in Code A-1.

**Code A-1** Simple FORM Tag Example

```
9  <B>Say Hello Form</B>
10
11 <FORM ACTION='/servlet/sl314.web.FormBasedHello' METHOD='POST'>
12 Name: <INPUT TYPE='text' NAME='name'> <BR>
13 <BR>
14 <INPUT TYPE='submit'>
15 </FORM>
```

This HTML page creates a GUI for entering the user's name. This form is shown in Figure A-5.

The figure shows a rectangular window with a pink border. Inside, at the top, is the title "Say Hello Form" in bold black font. Below the title is a label "Name:" followed by a text input field containing the text "Bryar". At the bottom of the window is a single button labeled "Submit Query".

**Figure A-5** The "Say Hello" HTML Form



**Note** – An HTML page may contain any number of forms. Each FORM tag contains the input tags for that specific form. In general, GUI components cannot be shared between forms even within the same HTML page.

## HTML Form Components

Web browsers support several major GUI components. These are listed in Table A-2.

**Table A-2** HTML Form Components.

Form Element	Tag	Description
Textfield	<INPUT TYPE='text' ...>	Enter a single line of text.
Submit button	<INPUT TYPE='submit'>	The button to submit the form.
Reset button	<INPUT TYPE='reset'>	The button to reset the fields in the form.
Checkbox	<INPUT TYPE='checkbox' ...>	Choose one or more options.
Radio button	<INPUT TYPE='radio' ...>	Choose only one option.
Password	<INPUT TYPE='password' ...>	Enter a single line of text, but the text entered cannot be seen.
Hidden	<INPUT TYPE='hidden' ...>	A static data field. This does not show up in the HTML form in the browser window, but the data is sent to the server in the CGI.
Select drop-down list	<SELECT ...> <OPTION ...> ... </SELECT>	Select one or more options from a list box.
Textarea	<TEXTAREA ...> ... </TEXTAREA>	Enter a paragraph of text.

## Input Tags

There are several types of INPUT tags. They all have three tag attributes in common:

- **TYPE** – The type of the GUI component.

This mandatory attribute specifies the type of the GUI component. The valid values of this attribute are: `text`, `submit`, `reset`, `checkbox`, `radio`, `password`, and `hidden`.

- **NAME** – The name of the form parameter.

This mandatory attribute specifies the name of the parameter that is used in the form data in the HTTP request.

- **VALUE** – The default value of the GUI component.

This is the value that is set in the GUI component when the HTML page is initially rendered in the Web browser. This is an optional attribute.

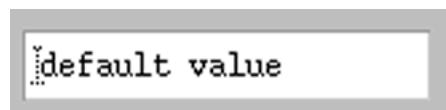
## Text Fields

An INPUT tag of type `text` creates a Textfield GUI component in the HTML form. An example Textfield component is shown in Code A-2.

**Code A-2** A Textfield HTML Tag

```
<INPUT TYPE='text' NAME='name' VALUE='default value' SIZE='20'>
```

A rendered Textfield as shown in Figure A-6.



**Figure A-6** The Textfield Component

A Textfield component allows the user to enter a single line of text. The data entered into this field by the user is included in the form data of the HTTP request when this form is submitted.



---

**Note** – A Textfield accepts an optional `SIZE` attribute, which allows the HTML developer to change the width of the field as it appears in the Web browser screen. There is also a `MAXSIZE` attribute, which determines the maximum number of characters typed into the field.

---

## Submit Buttons

An INPUT tag of type submit creates a Submit button GUI component in the HTML form. An example Submit button component is shown in Code A-3.

**Code A-3** Submit Button HTML Tags

```
<INPUT TYPE='submit'> <BR>
<INPUT TYPE='submit' VALUE='Register'> <BR>
<INPUT TYPE='submit' NAME='operation' VALUE='Send Mail'> <BR>
```

The example Submit buttons are rendered as shown in Figure A-7.



**Figure A-7** Submit Button Components

The Submit button triggers an HTTP request for this HTML form. If the Submit button tag *does not* include a VALUE attribute, then the phrase “Submit Query” is used as the text of the button. If the Submit button tag *does* include a VALUE attribute, then the value of that attribute is used as the text of the button.

If the Submit button tag *does not* include a NAME attribute, then form data is not sent for this GUI component. If the Submit button tag *does* include a NAME attribute, then that name (and the VALUE attribute) is used in the form data. This feature allows you to represent multiple actions that the form can process. For example, if the third Submit button tag in Code A-3 is clicked, then the name-value pair of “operation=Send+Mail” is included in the form data sent in the HTTP request.

## Reset Button

An INPUT tag of type reset creates a Reset button GUI component in the HTML form. An example Reset button component is shown in Code A-4.

**Code A-4** A Reset Button HTML Tag

```
<INPUT TYPE='reset'>
```

A rendered Reset button is shown in Figure A-8.



**Figure A-8** The Reset Button Component

The Reset button is special in the HTML form. It does not send any form data to the server. When selected, this button resets all of the GUI components in the HTML form to their default values.

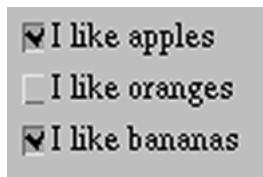
## Checkboxes

An INPUT tag of type checkbox creates a Checkbox GUI component in the HTML form. An example Checkbox component is shown in Code A-5.

**Code A-5** A Checkbox HTML Tag

```
<INPUT TYPE='checkbox' NAME='fruit' VALUE='apple'> I like apples <BR>
<INPUT TYPE='checkbox' NAME='fruit' VALUE='orange'> I like oranges <BR>
<INPUT TYPE='checkbox' NAME='fruit' VALUE='banana'> I like bananas <BR>
```

A rendered Checkbox is shown in Figure A-9.



**Figure A-9** The Checkbox Component

The Checkbox component allows the user to select multiple items from a set of values. For example, if “I like apples” and “I like bananas” are both checked, then two name-value pairs of “fruit=apple” and “fruit=banana” are included in the form data sent in the HTTP request. However, if no checkboxes are selected in the HTML form, then no name-value pairs are included in the form data sent in the request. The servlet developer must keep this feature in mind when extracting data from the HTTP request for checkboxes.

More specifically, the Checkbox component allows the user to toggle an item from checked to unchecked or from unchecked to checked. The checked position indicates that the VALUE of that Checkbox component will be added to the form data. If there are multiple Checkbox components, a user can toggle one independently of the others.

Checkbox components are grouped together by the NAME attribute. You can develop an HTML form with multiple, independent groups of checkboxes all with different names.

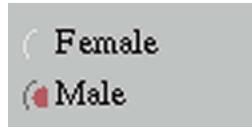
## Radio Buttons

An INPUT tag of type radio creates a Radio button GUI component in the HTML form. An example Radio button component is shown in Code A-6.

**Code A-6** A Radio Button HTML Tag

```
<INPUT TYPE='radio' NAME='gender' VALUE='F'> Female <BR>
<INPUT TYPE='radio' NAME='gender' VALUE='M'> Male <BR>
```

A rendered Radio button as shown in Figure A-10.



**Figure A-10** The Radio Button Component

The Radio button component allows the user to select one item from a mutually exclusive set of values.

Radio button components are grouped together by the NAME attribute. You can develop an HTML form with multiple, independent groups of radio buttons all with different names. It is the name of the radio button groups that determine the “mutually exclusive set of values” mentioned above.

## Password

An INPUT tag of type password creates a Password GUI component in the HTML form. An example Password component is shown in Code A-7.

**Code A-7** A Password HTML Tag

```
<INPUT TYPE='password' NAME='psword' VALUE='secret' MAXSIZE='16'>
```

A rendered Password component as shown in Figure A-11.



**Figure A-11** The Password Component

The Password component is similar to a Textfield, but the characters typed into the field are obscured. However, the value of the field is sent in the form data “in the open,” which means that the text entered in the field is not encrypted when it is sent in the HTTP request stream.

## Hidden Fields

An INPUT tag of type hidden creates a non-visual component in the HTML form. An example Hidden component is shown in Code A-8.

**Code A-8** A Hidden Field Tag

```
<INPUT TYPE='hidden' NAME='action' VALUE='SelectLeague'>
```

A Hidden component is not rendered in the GUI. The value of this type of field is sent directly in the form data of the HTTP request. Think of hidden fields as constant name-value pairs sent in the form data. For example, the Hidden field in Code A-8 will include “action=SelectLeague” in the form data of the HTTP request.

## The SELECT Tag

A SELECT tag creates a GUI component in the HTML form to select items from a list. There are two variations: single selection and multiple selection.

An example single selection component is shown in Code A-9.

**Code A-9** A Single Selection HTML Component

```
<SELECT NAME='favoriteArtist'>
  <OPTION VALUE='Genesis'> Genesis
  <OPTION VALUE='PinkFloyd' SELECTED> Pink Floyd
  <OPTION VALUE='KingCrimson'> King Crimson
</SELECT>
```

A rendered single selection component is shown in Figure A-12.



**Figure A-12** The Single Selection Component

The OPTION tag specifies the set of items (or options) that are selectable. The SELECTED attribute determines which option is the default selection when the HTML form is initially rendered or reset.

An example multiple selection component is shown in Code A-10.

**Code A-10** A Multiple Selection HTML Component

```
<SELECT NAME='sports' MULTIPLE>
  <OPTION VALUE='soccer' SELECTED> Soccer
  <OPTION VALUE='tennis'> Tennis
  <OPTION VALUE='ultimate' SELECTED> Ultimate Frisbee
</SELECT>
```

A rendered multiple selection component is shown in Figure A-13.



**Figure A-13** The Multiple Selection Component

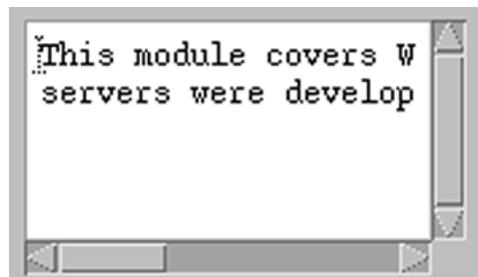
## The TEXTAREA Tag

A TEXTAREA tag creates a Textarea GUI component in the HTML form. An example Textarea component is shown in Code A-11.

**Code A-11** A Textarea HTML Tag

```
<TEXTAREA NAME='comment' ROWS='5' COLUMNS='70'>  
This module covers Web application basics: how browsers and Web  
servers were developed and what they do. It also...  
</TEXTAREA>
```

A rendered Textarea is shown in Figure A-14.



**Figure A-14** The Textarea Component

The Textarea component allows the user to enter an arbitrary amount of text. Multiline input is permitted in the Textarea component.

## Table Elements

Table A-3 shows the basic HTML elements used in table creation and their attributes.

**Table A-3** Table Elements and Their Attributes

Element	Attributes	Description
TABLE	BORDER	Indicates a table
CAPTION	ALIGN	Indicates a table caption
TR	ALIGN, VALIGN	Indicates a new row in the table
TH	ALIGN, VALIGN, COLSPAN, ROWSPAN, NOWRAP	Indicates a table heading
TD	ALIGN, VALIGN, COLSPAN, ROWSPAN, NOWRAP	Indicates a table cell or table data

Figure A-15 illustrates an HTML document that includes a table.

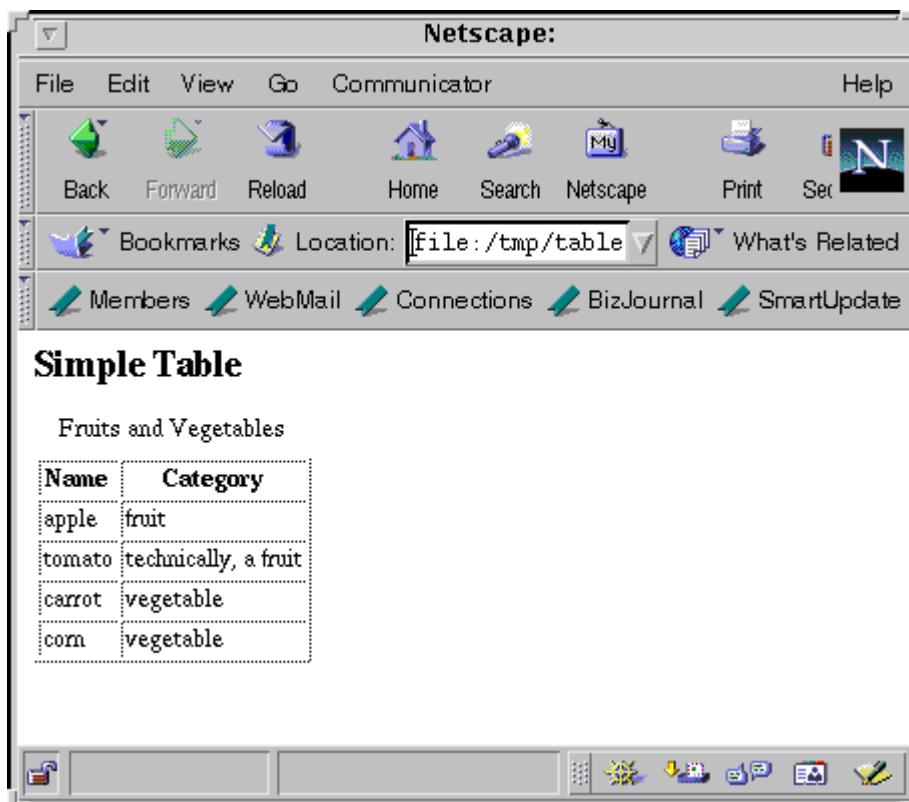
```
<HTML>
<BODY BGCOLOR="#fffffff">
<H1>Simple Table</H1>
<TABLE BORDER="1">
  <CAPTION>Fruits and Vegetables</CAPTION>
  <TR>
    <TH>Name</TH>
    <TH>Category</TH>
  </TR>
  <TR>
    <TD>apple</TD>
    <TD>fruit</TD>
  </TR>
  <TR>
    <TD>tomato</TD>
    <TD>technically, a fruit</TD>
  </TR>
  <TR>
    <TD>carrot</TD>
    <TD>vegetable</TD>
  </TR>
  <TR>
    <TD>corn</TD>
    <TD>vegetable</TD>
  </TABLE>
</BODY>
```

**Figure A-15** Table Specification in HTML Document

**Note** – If you did not use the BORDER="1" attribute, the table would not have lines separating the rows and columns.



Figure A-16 illustrates the previous HTML document in a browser.



**Figure A-16** HTML Table Displayed in Browser

# Advanced HTML

This section describes the basic function and syntactic structure of the JavaScript™ language.

## The JavaScript™ Language

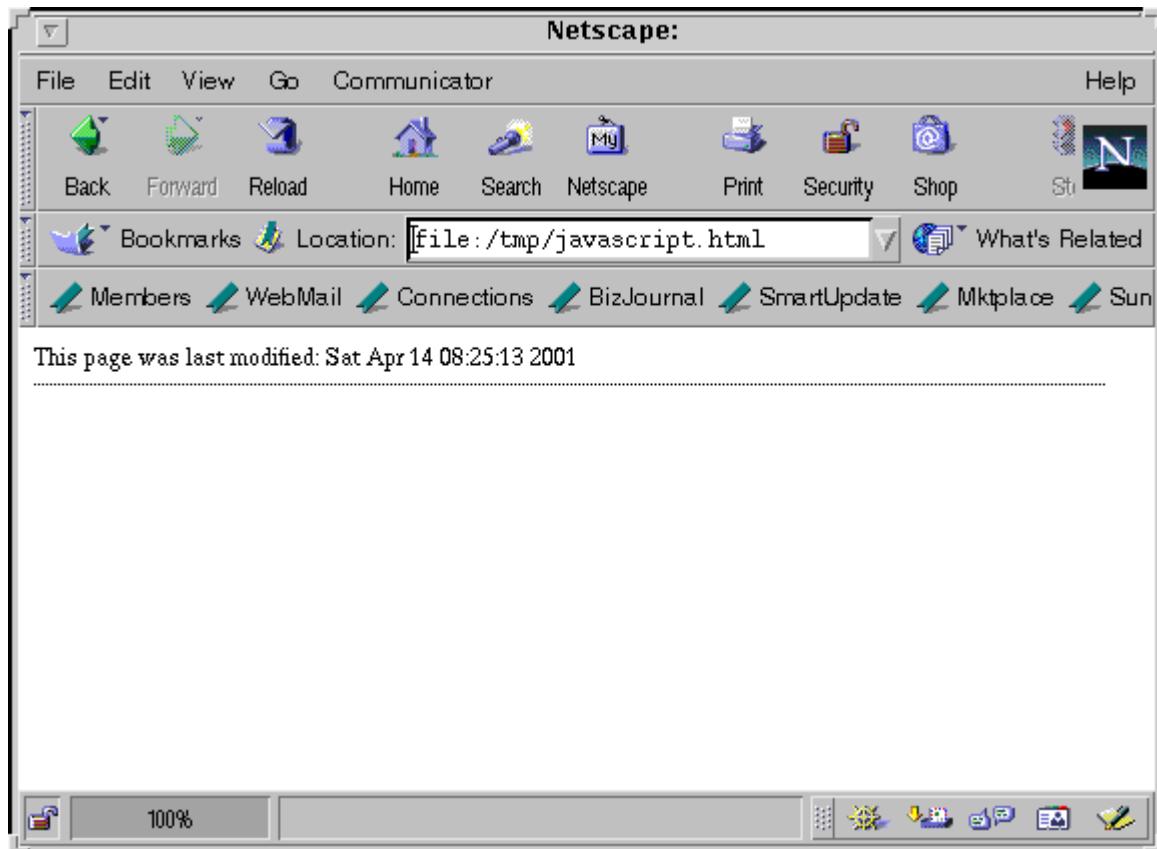
JavaScript is a scripting language (does not need to be compiled) that can be used to make your HTML more dynamic.

Use the element <SCRIPT LANGUAGE="JavaScript"> to indicate the beginning of your JavaScript code, ending your code with the end tag </SCRIPT>. The example in Code A-12 calls a function `setUp()` to set the document background color to white and display the date of last modification.

**Code A-12** Simple JavaScript Technology Example

```
<HTML>
<SCRIPT LANGUAGE="JavaScript">
 so that older browsers that do not understand the JavaScript programming language will ignore the code.
```

Code A-12 on page A-28 generates the HTML page illustrated in Figure A-17.



**Figure A-17** Display of HTML Document Containing the JavaScript Code

For tutorials and free JavaScripts, see <http://www.wsabstract.com/> or <http://www.javascript.com>.

**Note** – The JavaScript programming language has no relationship with the Java programming language.



## CSS

Cascading Style Sheets (CSS) is a way to add style, such as fonts, spacing, colors, and so on, to Web documents. For the latest on CSS, refer to <http://www.w3.org/Style/CSS/>.

Some Web page design applications support CSS. This enables you to create style sheets without worrying about the syntax of the language. However, there are times when you need to manually tweak a CSS file, so it is beneficial to understand the syntax.

### Style Sheets

A style sheet is a set of rules that apply to an HTML document, controlling how the HTML document is displayed. A rule has the following general syntax:

*selector {declaration}*

where declaration has the syntax:

*property : value*

A simple example is the following rule, which sets the color of H1 elements to blue:

```
H1 { color : blue }
```

All available properties are defined in <http://www.w3.org/TR/REC-CSS2>. If you had other style changes for the H1 element, you can enclose all of them within the curly braces “{ }”:

```
H1 {  
    color : blue;  
    font-style: italic;  
    text-transform : uppercase;  
}
```

## Attaching Style Sheets to an HTML Document

You can use the **STYLE** element within the **HEAD** element in an HTML document to specify all of your style sheet rules.

```
<HTML>
<HEAD>
<STYLE TYPE="text/css">
    <!-- to protect from non-CSS-supporting browsers
    -->
</STYLE>
</HEAD>
```

Indicate which style sheet language you are using with the **TYPE** attribute and enclose all of your rules between **<!--** and **-->**. Refer to the simple example in Code A-13.



**Note** – You can also refer to another style sheet in some other file by using the **LINK** element or by importing a style sheet that merges with the style sheet in the document. Refer to the CSS specification for details.

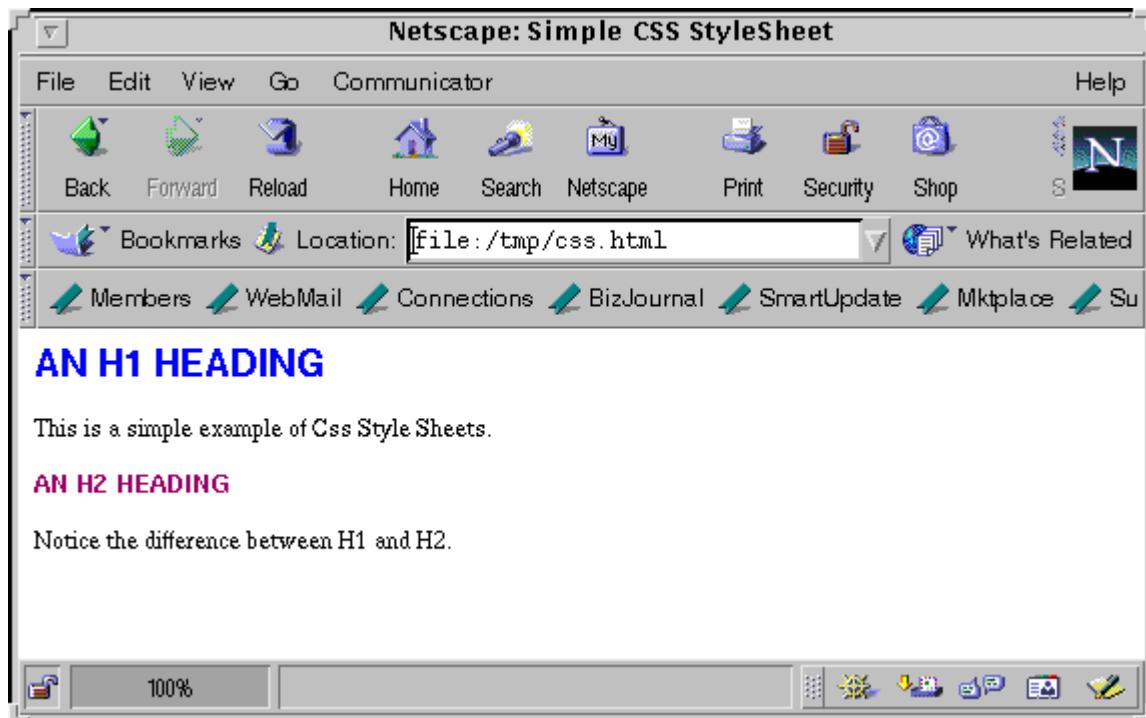
### Code A-13     CSS Example

```
<HTML>
<HEAD><TITLE>Simple CSS StyleSheet</TITLE>
<STYLE TYPE="text/css">
<!-- { protect from older browsers }
H1,H2 {
    font-family: helvetica;
    text-transform : uppercase;
    color : blue;
}
H2 {
    font-size : 12pt;
    color : purple;
}
</STYLE>
</HEAD>
<BODY BGCOLOR="#ffffff">
<H1>An H1 Heading</H1>
This is a simple example of Css Style Sheets.
<H2>An H2 Heading</H2>
Notice the difference between H1 and H2.
</BODY>
```



**Note** – The set of curly braces must be used in the line beginning with `<!--` or Netscape 4.7 ignores the first style sheet rule.

Code A-13 on page A-31 generates the HTML page illustrated in Figure A-18.



**Figure A-18** Display of CSS in a Browser

## Frames

Frames are used to provide several panes within the same browser window for document viewing. The basic elements and attributes for using frames in HTML documents are shown in Table A-4.

**Table A-4** Frame Elements and Their Attributes

Element	Attributes	Description
FRAMESET	COLS, ROWS	Replaces the BODY element in an HTML document and defines the layout of the frames
FRAME	MARGINWIDTH, MARGINHEIGHT, NAME, NORESIZE, SCROLLING, SRC	Defines the content and properties of a frame
NOFRAMES	none	Contains the normal BODY element and HTML markup for use in browsers that do not support frames

Use the ROWS or COLS attributes of the FRAMESET element to specify the number of frames within a frameset. If ROWS is used, the number of frames specified are laid out in rows; if COLS is used, the frames are laid out in columns. For example, if you want two frames laid out in columns, specify something similar to the following:

```
<FRAMESET ROWS="30%, 70%">
```

This indicates that the first frame will be 30 percent of the available height and the second frame will be the remainder (70 percent). You can specify actual numbers, interpreted as pixels, percentages as above, or relative size (using a number followed by an asterisk [\*]).

When using frames, you must create several HTML files: The first HTML file is sometimes referred to as the *master* file, because it specifies the layout of frames in the browser window. See Figure A-19 on page A-34. Each frame specified in the master HTML file uses the SRC attribute to indicate the initial HTML file to display in its frame. Therefore, you must create an HTML file for each SRC value specified in each FRAME element.

Figure A-19 shows an HTML document with frames.

Master file,  
controls  
frame layout

```
<HTML>
<FRAMESET COLS="20%, 80%">
    <FRAME SRC="menu.html" NAME="menu">
    <FRAME SRC="content.html" NAME="content">
</FRAMESET>
</HTML>
```

Content  
of left frame,  
menu.html

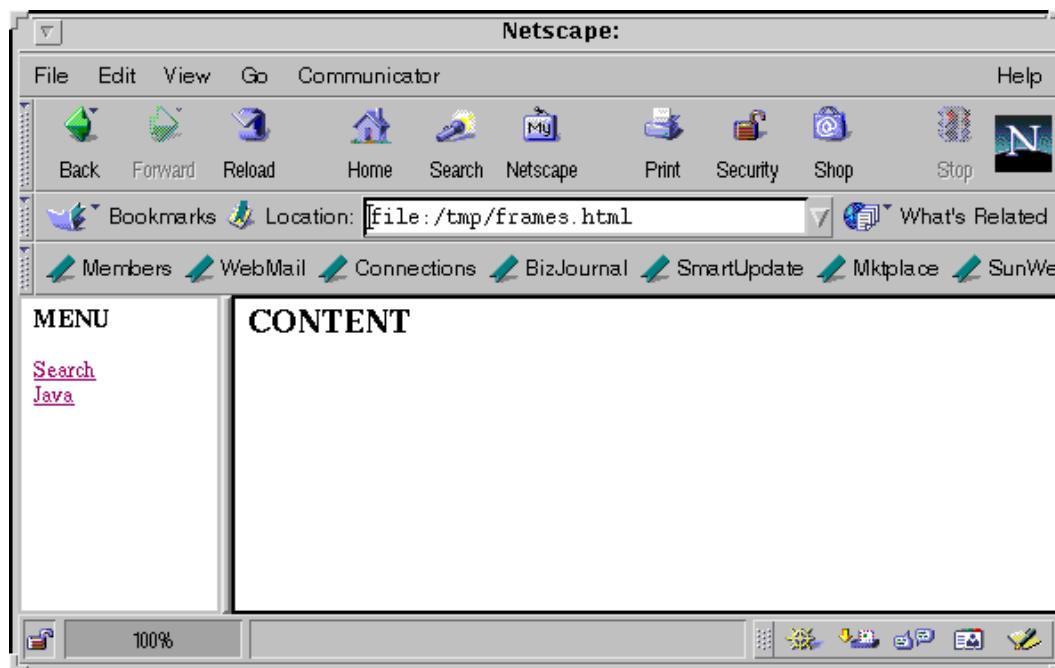
```
<HTML>
<BODY BGCOLOR="#ffffff">
<H3>MENU</H3>
<A HREF="http://yahoo.com" TARGET="content">Search</A>
<BR>
<A HREF="http://java.sun.com" TARGET="content">Java</A>
</BODY>
```

Initial  
content of  
right frame,  
content.html

```
<HTML>
<BODY BGCOLOR="#ffffff">
<H1>CONTENT</H1>
</BODY>
</HTML>
```

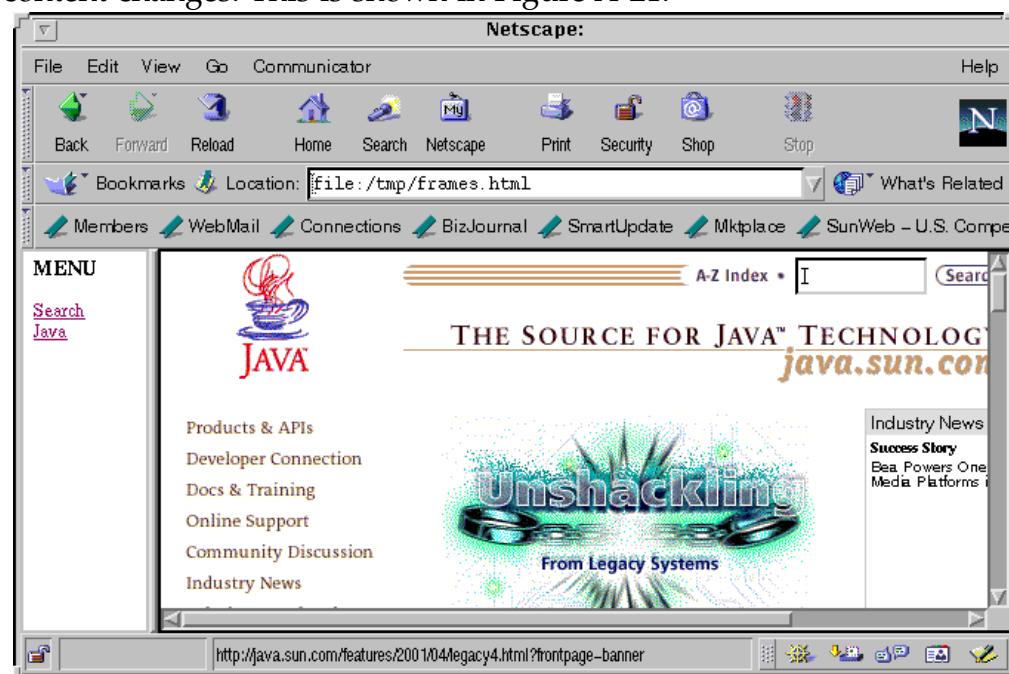
**Figure A-19** HTML Documents Using Frames

Figure A-20 shows the initial frames in a browser.



**Figure A-20** Browser Display of Frames

If the user clicks a link from the menu frame on the left, the right frame's content changes. This is shown in Figure A-21.



**Figure A-21** Frames After Clicking "Java" in the Menu Frame



# Quick Reference for HTTP

---

## Objectives

Upon completion of this appendix, you should be able to:

- Define and provide a short description of HTTP
- Describe the structure of HTTP requests and responses
- List the HTTP methods
- Explain the use of HTTP request header and response header fields
- Describe the five categories of HTTP status codes
- Define CGI and explain the relationship between some of the CGI environment variables and `HttpServletRequest` methods

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

- IETF – Hypertext Transfer Protocol (HTTP) Working Group. [Online]. Available at: <http://www.ics.uci.edu/pub/ietf/http/>.
- HTTP Specifications and Drafts. [Online]. Available at: <http://www.w3.org/Protocols/Specs.html#RFC>.
- The Common Gateway Interface. [Online]. Available at: <http://hoohoo.ncsa.uiuc.edu/cgi/>.

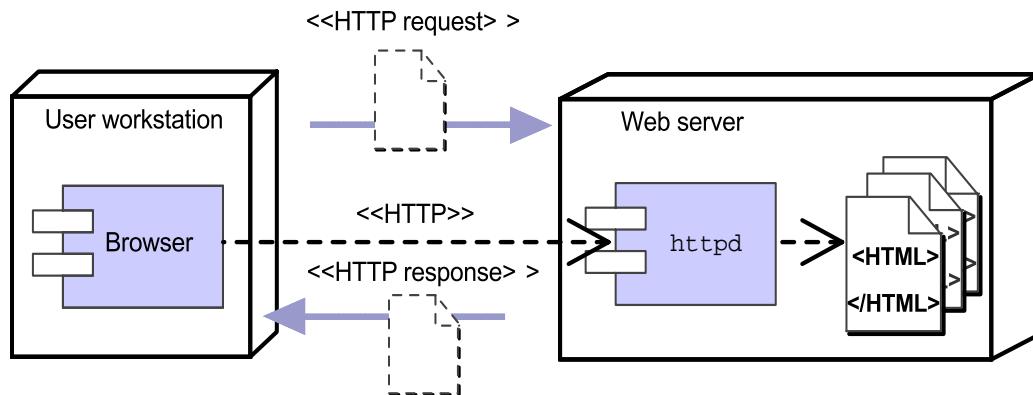
# Introduction to HTTP

This section defines the Hypertext Transfer Protocol.

## Definition

The Request for Comments (RFC) 2068 defines the Hypertext Transfer Protocol (HTTP) as an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol. HTTP can be used for many tasks, such as naming servers and distributed object management servers. These additional features can be achieved through extension of the HTTP request methods.

HTTP is a request and response driven protocol. That is, a client sends a request to a server and the server sends a response. HTTP usually takes place over a TCP/IP connection, but any other protocol that guarantees a reliable connection can be used. This process is illustrated in Figure B-1.



**Figure B-1** Simple HTTP Request and Response

## Structure of Requests

The request stream acts as an envelop to the request URL and message body of the HTTP client request. The request stream has the following format:

```
Request = Request-Line  
        ( Header-Line )*  
        CRLF  
        [ Message-Body ]
```

where

Request-Line = Method SP Request-URL SP HTTP-Version CRLF

Header-Line = header-name ":" header-value ("," header-value)\*

Message-Body is either empty or contains arbitrary text or binary data (usually empty).

An example request stream is show in Code B-1.

**Code B-1**      Example HTTP Request Header

```
1  GET /servlet/s1314.web.HelloServlet HTTP/1.0  
2  Connection: Keep-Alive  
3  User-Agent: Mozilla/4.76 [en] (X11; U; SunOS 5.8 sun4u)  
4  Host: localhost:8088  
5  Accept: image/gif, image/x-xbitmap, image/jpeg, image/pj  
6  Accept-Encoding: gzip  
7  Accept-Language: en  
8  Accept-Charset: ISO-8859-1,* ,utf-8  
9
```

## HTTP Methods

The method specified in the Request-Line indicates what operation should be performed by the server on the resource identified by Request-URL. All servers must support GET and HEAD methods.

HTTP 1.1 defines the methods shown in Table B-1:

**Table B-1** HTTP Methods and Descriptions

Method	Description
GET	Retrieves the information identified by the Request-URL.
HEAD	Retrieves only the meta-information contained in the HTTP headers; no message-body is returned in the response. The header information in a response to a HEAD method and a GET method should be identical.
POST	Requests that the server accept the entity identified in the request as part of the resource specified by the Request-URL in the Request-Line.
PUT	Requests that the entity included in the request be stored under the Request-URL.
DELETE	Requests that the server delete the resource specified by Request-URL.
OPTIONS	Requests information about the communication options available on the request and response chain identified by the Request-URL.
TRACE	Invokes a remote, application-layer loopback (trace) of the request message.

## Request Headers

The client can use the request header field to pass additional information about the request or itself to the server. Table B-2 on page B-6 lists the request header names that are defined by HTTP 1.1.

**Table B-2** Request Header Names

Name	Description
Accept	Specifies certain media types that are acceptable in the response.
Accept-Charset	Indicates the character sets that are acceptable in the response.
Accept-Encoding	Restricts the content-coding values that are acceptable in the response.
Accept-Language	Restricts the set of languages that are preferred in the response.
Authorization	Indicates a user agent that wishes to authenticate itself with a server; this field consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.
From	Contains an Internet email address for the human user who controls the requesting user agent.
Host	Specifies the Internet host and port number of the resource being requested.
If-Modified-Since	Makes a GET method conditional; do not return the requested information if it was not modified since the specified date.
If-Match	Makes a method conditional; allows for efficient update of cached information.
If-None-Match	Makes a method conditional; allows efficient update of cached information with a minimum amount of transaction overhead.
If-Range	Obtains an up-to-date copy of an entire entity based on a partial copy in the cache. Usually used with If-Unmodified-Since or If-Match (or both).
If-Unmodified-Since	Makes a method conditional; if the requested resource has not been modified since the time specified in this field, the server should perform the requested operation.

**Table B-2** Request Header Names (Continued)

Name	Description
Max-Forwards	Limits the number of proxies or gateways that can forward the request to the next inbound server. Used with the TRACE method.
Proxy-Authorization	Enables the client to identify itself (or its user) to a proxy which requires authentication.
Range	Requests one or more sub-ranges of the entity, instead of the entire entity.
Referer	Enables the client to specify, for the server's benefit, the address (URL) of the resource from which the Request-URL was obtained.
User-Agent	Contains information about the user agent originating the request.

## Structure of Responses

The response stream acts as an envelop to the message body of the HTTP server response. The response stream has the following format:

```
Response = Status-Line
          ( Header-Line )*
          CRLF
          [ Message-Body ]
```

where

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
Header-Line = header-name ":" header-value ("," header-value)*
Message-Body is either empty or contains arbitrary text or binary data.
```

An example is show in Code B-2.

**Code B-2**      Example HTTP Response Header

```
1  HTTP/1.0 200 OK
2  Content-Type: text/html
3  Date: Tue, 10 Apr 2001 23:36:58 GMT
4  Server: Apache Tomcat/4.0-b1 (HTTP/1.1 Connector)
5  Connection: close
6
7  <HTML>
8  <HEAD>
9  <TITLE>Hello Servlet</TITLE>
10 </HEAD>
11 <BODY BGCOLOR='white'>
12 <B>Hello, World</B>
13 </BODY>
14 </HTML>
15
```

## Response Headers

The server uses the response header field to pass additional information about the response that cannot be placed in the Status-Line. The additional information can be about the server or about further access to the resource identified by the Request-URL.

Table B-3 lists the response-header names that are defined in HTTP 1.1.

**Table B-3** Response Header Names

Name	Description
Accept-Ranges	Indicates the server's acceptance of range requests for a resource.
Age	Estimates the amount of time since the response (or its revalidation) was generated at the server.
Location	Redirects the recipient to a location other than the Request-URL for completion of the request or identification of a new resource.
Proxy-Authenticate	<i>Must</i> be included as part of a status code 407 (Proxy Authentication Required) response; requests the client or user agent to authenticate (which they can do with an Authorization request-header field). Applies only to the current connection, unlike WWW-Authenticate.
Public	Lists the set of methods supported by the server.
Retry-After	Indicates how long the service is expected to be unavailable to the requesting client. Used with a status code 503 (Service Unavailable) response.
Server	Contains information about the software used by the origin server to handle the request.
Vary	Signals that the response entity was selected from the available representations of the response using server-driven negotiation.
Warning	Carries additional information about the status of a response that may not be reflected by the response status code.
WWW-Authenticate	<i>Must</i> be included in 401 (Unauthorized) response messages; requests the client or user agent to authenticate (which they can do with an Authorization request-header field).

## Status Codes

Status codes are three-digit integers that are part of an HTTP response. These codes provide an indication of what happened in attempting to understand and fulfill a request.

Table B-4 defines the status codes for HTTP. Currently, there are five categories of status codes:

- 1xx codes – Indicates a provisional response
- 2xx codes – Indicates that the client's request was successfully received, understood, and accepted
- 3xx codes – Indicates that further action is needed to fulfill the request
- 4xx codes – Indicates that the client has erred
- 5xx codes – Indicates that the server is aware that it has erred or cannot perform the request

**Table B-4** HTTP Status Codes

Code	Reason-Phrases
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
205	Reset Content
206	Partial Content
300	Multiple Choices
301	Moved Permanently
302	Moved Temporarily
303	See Other
304	Not Modified

**Table B-4** HTTP Status Codes (Continued)

<b>Code</b>	<b>Reason-Phrases</b>
305	Use Proxy
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	URL Too Long
415	Unsupported Media Type
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

## CGI

CGI means Common Gateway Interface and is a standard for communication between server-side gateway programs and HTTP servers. The specification is maintained by the National Center for Supercomputing Applications (NCSA) at <http://hoohoo.ncsa.uiuc.edu/cgi/>.

The HTTP server passes data from the client to the gateway program. When the program has processed the data, it sends return information to the server, and the server then forwards the response to the client.

The CGI standard specifies how data is transferred between the server and gateway program.

## Set of Environment Variables

One way that a server can pass information to a gateway program is to put the information into environment variables before invoking the program. There are special environment variables to hold all of the information types the client sends to the server in the client's HTTP request. Table B-5 lists the various CGI variables and, where relevant, identifies the corresponding `HttpServletRequest` method that obtains the information represented by the CGI environment variable.

**Table B-5** CGI Environment Variables and Corresponding `HttpServletRequest` Methods

CGI Variable	Description	<code>HttpServletRequest</code> Method
GATEWAY_INTERFACE	The version of the Common Gateway Interface that the server uses.	no method
SERVER_NAME	The server's host name or IP address.	<code>getServerName</code>
SERVER_SOFTWARE	The name and version of the server software that is answering the client request.	no method
SERVER_PROTOCOL	The name and version of the information protocol the request came in with.	<code>getProtocol</code>

**Table B-5** CGI Environment Variables and Corresponding `HttpServletRequest` Methods (Continued)

CGI Variable	Description	<code>HttpServletRequest</code> Method
<code>SERVER_PORT</code>	The port number of the host on which the server is running.	<code>getServerPort</code>
<code>REQUEST_METHOD</code>	The method with which the information request was issued.	<code>getMethod</code>
<code>PATH_INFO</code>	Extra path information passed to a CGI program.	<code>getPathInfo</code>
<code>PATH_TRANSLATED</code>	The translated version of the path given by the variable <code>PATH_INFO</code> .	<code>getPathTranslated</code>
<code>SCRIPT_NAME</code>	The virtual path (for example, <code>/cgi-bin/script.pl</code> ) of the script being executed.	No method
<code>QUERY_STRING</code>	The query information passed to the program. It is appended to the URL with a "?".	<code>getQueryString</code>
<code>REMOTE_HOST</code>	The remote host name of the user making the request.	<code>getRemoteHost</code>
<code>REMOTE_ADDR</code>	The remote IP address of the user making the request.	<code>getRemoteAddr</code>
<code>AUTH_TYPE</code>	The authentication method used to validate a user.	<code>getAuthType</code>
<code>REMOTE_USER</code>	The authenticated name of the user.	<code>getRemoteUser</code>
<code>REMOTE_IDENT</code>	The remote user name making the request. This variable is only set if the <code>IdentityCheck</code> flag is enabled, and the client machine supports the RFC 931 identification scheme ( <code>identd</code> identification daemon).	No method
<code>CONTENT_TYPE</code>	The MIME type of the query data, such as "text/html."	<code>getContentType</code>
<code>CONTENT_LENGTH</code>	The length of the data (in bytes or the number of characters) passed to the CGI program through standard input.	<code>getContentLength</code>

**Table B-5** CGI Environment Variables and Corresponding HttpServletRequest Methods (Continued)

CGI Variable	Description	HttpServletRequest Method
HTTP_ACCEPT	A list of the MIME types that the client can accept.	no method
HTTP_USER_AGENT	The browser that the client is using to issue the request.	no method

## Data Formatting

CGI scripts are often used to process user information submitted in HTML forms. This information sent to a CGI script is encoded. The convention used is the same as for URL encoding:

- Each form's element name is paired with the value entered by the user to create a key-value pair. If no user input is provided for some element name, the value is left blank when passed ("element\_name=").
- Each key-value pair is separated by an ampersand (&).
- Special characters, such as forward slash (/), double quotes ("), and ampersand (&), are converted to their hexadecimal codes, %xx.
- Spaces are converted to a plus sign (+).

The CGI program parses this encoded information. The program uses the value of the environment variable REQUEST\_METHOD to determine if the request is a GET or a POST.

If the request is a GET, then the program parses the value of the QUERY\_STRING environment variable. The query string is appended to the URL following a question mark (?). For example, if searching for "cgi http" at yahoo.com, the URL becomes the following after clicking the Search button:

```
http://search.yahoo.com/bin/search?p=cgi+http
```

The program is search; the keywords entered by the user for the search are "cgi" and "http," and the query string is everything following the question mark (?).

If the request is a POST, the size of the data to read from standard input and parse is determined by the value of the environment variable CONTENT\_LENGTH.

When processing these data strings, a CGI program needs to decode or undo what was encoded:

- Separate key-value pairs at each ampersand (&)
- Change all plus signs (+) to spaces and hexadecimal codes to ASCII characters



# Quick Reference for the Tomcat Server

---

## Objectives

Upon completion of this appendix, you should be able to:

- Explain what the Tomcat server is and what organization is developing it
- Identify the major steps for installing the Tomcat server
- Describe how to start and stop the Tomcat server
- Describe some of the key elements used in configuring the Tomcat server and the relevance of the file `server.xml`
- Describe the logging process used in the Tomcat server

## Additional Resources



**Additional resources** – The following reference provides additional information on the topics described in this module:

- The Jakarta Project: Subprojects – Tomcat. [Online]. Available at: <http://jakarta.apache.org/tomcat/index.html>.

## Definition of the Tomcat Server

The Tomcat server is a reference implementation for the Java™ Servlet and JavaServer Pages (JSP) technologies. It is a subproject of the Jakarta Project, an endeavor of the Apache Software Foundation (<http://www.apache.org>) to create open server solutions based on the Java platform.

Tomcat can be further described as a container or shell that manages and executes servlets based on user requests. The container also has a JSP technology engine. You can run Tomcat as a standalone container (mainly used for development and debugging) or as an add-on to a Web server.

## Installation Instructions

Full installation details are provided with the releases of the Tomcat server or you can refer to the online documentation at:  
<http://jakarta.apache.org/tomcat/jakarta-tomcat/src/doc/index.html>.

The basic steps for installing the Tomcat server are:

- Download a ZIP, GZIP, or TAR file from  
<http://jakarta.apache.org/downloads/binindex.html>.
- Extract the Tomcat files into some predetermined directory using the zip, gunzip, or tar xf commands. This should create a new subdirectory named *jakarta-tomcat-version*, where *version* represents the version of the Tomcat server that you downloaded (such as 3.2.1 or 4.0-b3).
- Set the environment variable **TOMCAT\_HOME** to the location of your Tomcat server installation. For example, if you extracted the Tomcat files in the directory /usr/local, then the **TOMCAT\_HOME** variable is set to /usr/local/jakarta-tomcat-version.
- Add the  `${TOMCAT_HOME} /bin` directory to your **PATH** environment variable.
- Set the environment variable **JAVA\_HOME** to the location of your Java™ 2 SDK installation.
- Add the  `${JAVA_HOME} /bin` directory to your **PATH** environment variable.

You can now execute the Tomcat server as a standalone servlet container.

## Starting and Stopping the Tomcat Server Execution

This section describes how to start and shutdown the Tomcat server.

### Starting the Tomcat Server

When you have completed installation of the Tomcat server, execution is started using the startup script provided in the Tomcat server bin directory. For UNIX® environments, use the `startup.sh` script; for Microsoft Windows environments, use the `startup.bat` script.

The following is the output from starting the Tomcat server version 4.0 in a UNIX environment:

#### **startup.sh**

```
Guessing CATALINA_HOME from catalina.sh to /work/jakarta-tomcat-4.0-b3/bin/..
Setting CATALINA_HOME to /work/jakarta-tomcat-4.0-b3/bin/..
Using CLASSPATH: /work/jakarta-tomcat-4.0-b3/bin/../bin/bootstrap.jar
Using CATALINA_HOME: /work/jakarta-tomcat-4.0-b3/bin/..
```

---

**Note** – For the Tomcat server version 4.0, the  `${CATALINA_HOME}` value is the same as  `${TOMCAT_HOME}`.

---



The actual command that is responsible for starting the Tomcat server version 4.0 execution is:

```
$JAVA_HOME/bin/java $CATALINA_OPTS -classpath $CP \
-Dcatalina.home=$CATALINA_HOME \
org.apache.catalina.startup.Bootstrap "$@" start \
>> $CATALINA_HOME/logs/catalina.out 2>&1 &
```

---

**Note** – The output and the java command used to start and stop the Tomcat server varies slightly from one release of the Tomcat server to another.

---



## Stopping the Tomcat Server

Stopping execution of the Tomcat server is also accomplished using a script in the Tomcat bin directory. For UNIX environments, use the script shutdown.sh, and for Microsoft Windows environments, use the script shutdown.bat.

The following is the output from stopping execution of the Tomcat server version 4.0 in a UNIX environment:

**shutdown.sh**

```
Guessing CATALINA_HOME from catalina.sh to /work/jakarta-tomcat-4.0-b3/bin/..
Setting CATALINA_HOME to /work/jakarta-tomcat-4.0-b3/bin/..
Using CLASSPATH: /work/jakarta-tomcat-4.0-b3/bin/..../bin/bootstrap.jar
Using CATALINA_HOME: /work/jakarta-tomcat-4.0-b3/bin/..
```

The actual command responsible for stopping the Tomcat server version 4.0 execution is:

```
$JAVA_HOME/bin/java $CATALINA_OPTS -classpath $CP \
-Dcatalina.home=$CATALINA_HOME \
org.apache.catalina.startup.Bootstrap "$@" stop
```

# Configuration

The main configuration file for the Tomcat server is the `server.xml` file. The default configuration file used during the Tomcat server execution is  `${TOMCAT_HOME}/conf/server.xml`.



**Note** – The structure of the configuration file has changed slightly between Tomcat v3.2.1 and v4.0. Check the `conf/server.xml` file of your Tomcat server version for correct element usage if you plan to make changes to `server.xml`.

Because `server.xml` is an XML file, it is composed of various elements. The following are some important elements in the `server.xml` file. All examples and descriptions are based on the file  `${TOMCAT_HOME}/conf/server.xml` of the Tomcat server version 4.0.

- **Server** – The root element in the file `server.xml`. A `Server` element defines a single Tomcat server. A `Server` element contains one or more sub-elements called `Service`.

```
<Server port="8005" shutdown="SHUTDOWN" debug="0">
...
</Server>
```

- **Service** – A collection of one or more sub-elements called `Connector` that share a single container (usually the container is an `Engine` element) to process incoming requests.

```
<!-- Define the Tomcat Stand-Alone Service -->
<Service name="Tomcat-Standalone">
    <Connector ... />
    <Connector ... />
    <Engine ... />
</Service>
```

- **Connector** – A connection to the user, either through a Web server or directly to the user's browser (in a standalone configuration) and through which requests are received and responses are returned.

```
<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->
<Connector
    className="org.apache.catalina.connector.http.HttpConnector"
    port="8080" minProcessors="5" maxProcessors="75"
    acceptCount="10" debug="0"
    connectionTimeout="60000"
/>
```

- Engine – The entry point (in Tomcat 4.0) that processes every request, such as analyzing HTTP headers and passing them to the appropriate virtual host.

```
<!-- Define the top level container in our container hierarchy -->
<Engine name="Standalone" defaultHost="localhost" debug="0">
...
</Engine>
```

- Logger – A logger object; that is, a Java technology class that logs messages and errors, specified by the className attribute of the Logger element. The detail of messages written to the log file is controlled by the verbosity attribute.

Available loggers include the following:

- FileLogger – The file logger, which records logged messages to disk files in a specified directory
- SystemErrLogger – The standard error logger, which records logged messages to the standard error output stream
- SystemOutLogger – The standard output logger, which records all logged messages to the standard output stream

By default, the FileLogger logger writes messages to the \${CATALINA\_HOME}/logs directory, but you can use the directory attribute to indicate the directory where log messages should be written.

---

**Note** – If absolute paths are not used with the directory attribute, paths are relative to the \${CATALINA\_HOME} directory.



The following creates log files in the logs directory with names such as localhost\_log.2001-04-26.txt, where localhost\_log. is the prefix, 2001-04-26 is the timestamp, and .txt is the suffix. The timestamp value will change.

```
<Logger className="org.apache.catalina.logger.FileLogger"
       directory="logs"  prefix="localhost_log."  suffix=".txt"
       timestamp="true"
/>
```

The next Logger element creates log files with names similar to catalina\_log.2001-04-26.txt.

```
<!-- Global logger unless overridden at lower levels -->
<Logger className="org.apache.catalina.logger.FileLogger"
    prefix="catalina_log." suffix=".txt"
    timestamp="true"
/>
```

- Host – The default virtual host. The appBase attribute indicates the document root for your applications. Here, a relative path is specified, so this is relative to \${CATALINA\_HOME}.

```
<Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">
...
</Host>
```

- Context – Represents an individual Web application running in a particular host. This element is only necessary if you want to set non-default properties or specify document roots for a Web application in a location other than the virtual host's appBase directory.

```
<!-- Tomcat Examples Context -->
<Context path="/examples" docBase="examples" debug="0" reloadable="true">
...
</Context>
```

In the previous example, the path attribute represents the context path for your application, which is the prefix of a request URI. So any request URI beginning with /examples will be processed by this application. This attribute is required, and must start with a slash ('/') character.

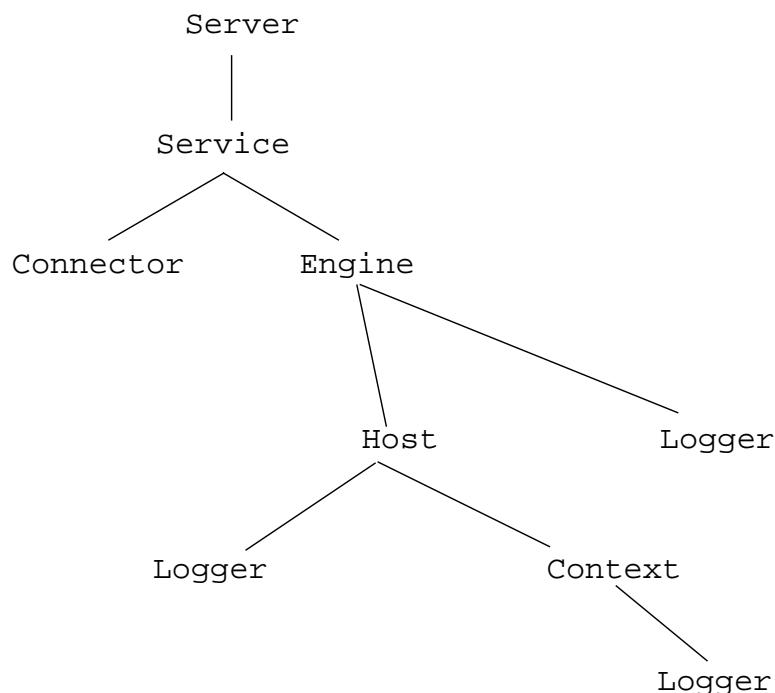
The docBase attribute represents the document root directory for this Web application. This can be a relative path or an absolute path to the directory containing your application.

---

**Note** – This is not a complete list of available XML elements in the server.xml file. Refer to the Tomcat documentation for all elements and descriptions of each.



Figure C-1 shows allowable nesting relationships for the previously mentioned XML elements.



**Figure C-1** Allowable Nesting for `server.xml` Elements

## Logging and Log Files

Tomcat supports a logging feature to track what happens during Tomcat execution with messages and exceptions from the `ServletContext` interface. You control the detail of messages written to the log files using the `verbosityLevel` attribute of the `Logger` element, configured in the `conf/server.xml` configuration file.

The `Logger` interface of the Tomcat software supports this logging feature and is configured using the `Logger` element in the `server.xml` file. The verbosity levels for message detail are:

- FATAL – Assigned integer value is `Integer.MIN_VALUE`.
- ERROR – Assigned integer value is 1.
- WARNING – Assigned integer value is 2.
- INFORMATION – Assigned integer value is 3.
- DEBUG – Assigned integer value is 4.

A message is written to the log file when the verbosity level is greater than or equal to the specified value for the message. Thus, the log file becomes more detailed as the verbosity level increases.

---

**Note** – The default verbosity level is 1.

---



# Quick Reference for the Ant Tool

---

## Objectives

Upon completion of this appendix, you should be able to:

- Describe Ant and provide a skeletal Ant build file
- Describe the basic structure of an Ant build file, including projects, targets, properties, and tasks
- Describe how file sets and filtering are used in Ant
- Provide a brief description of the following basic Ant built-in tasks: `copy`, `delete`, `mkdir`, `echo`, `javac`, `javadoc`, and `jar`
- Explain the syntax for executing Ant, including a description of the options `-buildfile` and `-D`
- Identify the major steps for installing Ant

## Additional Resources



**Additional resources** – The following reference provides additional information on the topics described in this module:

- The Jakarta Site – Ant. [Online]. Available at:  
<http://jakarta.apache.org/ant/>.

## Introduction to Ant

Ant is a build tool, similar to the make build tool, based on Java technology. Ant is not operating system dependent because it is not shell-based. Based on Java technology and extensible using Java classes, Ant is platform independent.

The build files for Ant are eXtensible Markup Language (XML) files that contain build rules for one project. Figure D-1 shows the skeletal XML syntax for an Ant build file.

```
<project name="MyProject" default="defaultName" basedir=".">
  <property name="propertyName1" value="PropertyValue1" />
  <property name="propertyName2" value="PropertyValue2" />

  <target name="targetName1" ... >
    <!-- task elements go here; use Ant task names -->
    <task1 ... />
    <task2 ... />
  </target>

  <target name="targetName2" ... >
    <!-- task elements go here; use Ant task names -->
    <task1 ... />
    <task2 ... />
    <task3 ... />
  </target>

</project>
```

**Figure D-1**    Skeletal Ant Build File

## Build File Structure

This section describes the basic structure of an Ant build file. If necessary, refer to the skeletal syntax shown in Figure D-1 on page D-3.

### Projects

As mentioned previously, each XML build file contains one project, specified by the project element. This is the root element (all other elements are defined between the start and end tag of the project element) of the XML file. The project element has three attributes:

- name – The name of the project; not required to be specified.
- default – The default target to use when no target is supplied; this attribute specification is required.
- basedir – The base directory from which all path calculations are made. This attribute can be overridden by setting the basedir property beforehand. When this is done, it must be omitted in the project tag. If neither the attribute nor the property have been set, the parent directory of the build file is used as the base directory.

The following example defines a project named `ServletsJSPs`, with default target `lab1`, and uses the current directory for all path calculations.

```
<project name="ServletsJSPs" default="lab1" basedir=". ">
```

### Targets

Targets in the build file are defined by the target element. The target element is composed of a set of tasks that need to be executed and has the following attributes:

- name – The target name; this is a required attribute.
- depends – A comma-separated list of target names that this target depends on. The tasks specified by these other targets must be completed before the tasks for this target are executed.
- if – The property name that *must* be set for this target to execute.

- unless – The property name that *must not* be set for this target to execute.
- description – A one-line description of the target's function; this description is printed if the -projecthelp option is used when executing Ant.

## Tasks

Tasks are specified between the start and end tags of the target element. Tasks are actually done for a specific target. The syntax for a task is:

```
<name attribute1="value1" attribute2="value2" ... />
```

Ant provides a set of task names, which are divided into two categories: built-in tasks and optional tasks. Refer to the Ant User Manual for a complete listing of each. Some basic task names are discussed on page D-8.

## Properties

One of the built-in task names in Ant is the property task, which sets properties in the project. There are several ways to define properties (all are discussed in the Ant User Manual). Two common ways are:

- Use the property tag with the name and value attributes.
- Use the property tag with the file or resource attribute to specify a file to read the properties from. This property file has the format as defined by the class java.util.Properties.

A property using the syntax \${*propertyName*} can be used elsewhere, such as in specifying task attributes. For example, the following defines a property named install\_dir that is used to define the value of the attribute dir in the mkdir task.

```
<property name="install_dir" value="/usr/local" />
<target name="createDir">
    <mkdir dir="${install_dir}/java" />
</target>
```

The attribute dir thus has the value /usr/local/java.

You can also access all Java system properties, such as os.name or user.name, using the syntax \${os.name} or \${user.name}.

## Ant Special Features

Special features of Ant that are often used with other tasks include: patterns, filesets, and filtering.

### Patterns

Patterns are used to define specific sets of files to be processed. Use patterns to include or exclude files or directories that match a pattern.

The following syntax is used in Ant:

- \* – Matches zero or more characters. For example, \*.java matches all files in the current directory ending in .java.
- ? – Matches one character. For example, ?? .java matches all files in the current directory that end with .java and whose file name prefix is composed of two characters, such as Ac.java, cy.java, and so on.
- \*\* – Matches zero or more directories. For example, src/\*\*/\* .java matches all files in any directory under src that end in .java, such as src/mod1/\*.java or src/\*.java.

### The `fileset` Element

A *fileset* is a group of files. Use the `fileset` element to identify a particular set of files. The `fileset` element occurs inside specific tasks if the task supports filesets, or it occurs as a child of the `project` element (at the same level as `target` elements).

The `fileset` element has several attributes, but only the `dir` attribute is required. The `dir` attribute specifies the root of the directory tree for the `fileset` being defined.

When defining a fileset, patterns are often used to identify the files within the fileset. For example, the following defines a fileset that includes all files ending with .java in any directory beneath the directory indicated by \${myclasses.src}:

```
<fileset dir="${myclasses.src}" >
  <patternset >
    <include name="**/*.java" />
  </patternset>
</fileset>
```

## Filtering

Ant provides a filter task and several tasks have an attribute called filtering. A project can specify a set of token filters that are automatically expanded when a task is performed that supports filtering and for which the filtering attribute is “yes.” Token filters are used by all tasks that perform file copying operations.

Tokens have the following form:

@token@

and are defined using the filter task.

---

**Note** – The token string must not contain the separator character @.

---



### The filter Task

The filter task is used to define or specify token filters for a project. Explicitly define the token in the Ant build file using the attributes token and value, or you can specify a file from which the filters are read using the filtersfile attribute.

For example, the following token filter indicates to replace “version” with “Java 2 v1.3” when performing any task that has filtering set to “yes” and @version@ is encountered in the file contents.

```
<filter token="version" value="Java 2 v1.3"/>
```

# Basic Built-in Ant Tasks

This section describes some basic built-in Ant tasks.

## The copy Task

Use the `copy` task to copy a file or a fileset to a new file or directory. Unless you use the `overwrite` attribute, files are only copied if the destination file does not exist or if the file to be copied is newer than the file to which you are copying.

### Commonly Used Attributes

Attributes that are often used with the `copy` task include:

- `file` – Indicates the file to copy; this is a required attribute.
- `tofile` – Indicates the file to copy to; this attribute or the `todir` attribute is required.
- `todir` – Indicates the directory to copy to; this attribute or the `tofile` attribute is required.
- `filtering` – Indicates whether filtering is done during the copy; default value is “no.”



**Warning** – If you copy binary files with `filtering` turned on, you can corrupt the files. The `filtering` attribute of the `copy` task should be used with text files only.

---

### Examples

The following task copies the file `/tmp/MyTest.java` to the file `/java/src/MyTest.java`.

```
<copy file="/tmp/MyTest.java"
      tofile="/java/src/MyTest.java"
      />
```

This next task copies all files in the directory tree under /java/src that do not end in .java to the directory /java/myclasses.

```
<copy todir="/java/myclasses">
    <fileset dir="/java/src" >
        <exclude name="**/*.java" />
    </fileset>
</copy>
```

The following copy task uses filtering to recursively copy all files from the \${notes} directory to the \${readme} directory replacing all the occurrences of @version@ within the files being copied by the string "Java 2, v1.3".

```
<filter token="version" value="Java 2, v1.3" />
<copy todir="${readme}" filtering="yes">
    <fileset dir="${notes}" />
</copy>
```

Therefore, if the following was the content of a file in the \${notes} directory:

This course covers @version@ technology.

then when this file is copied to the \${readme} directory during the Ant build, the contents of the copied file becomes:

This course covers Java 2, v1.3 technology.

## The delete Task

Use the delete task to remove a file, a set of files, or a directory and its subdirectories.

### Attributes

The delete task has many attributes, but you are only required to specify the file or dir attribute, indicating the file to delete or the directory and all files and subdirectories to delete.

### Examples

The following delete task deletes the file /tmp/MyTest.java.

```
<delete file="/tmp/MyTest.java" />
```

The next delete task removes all files ending with .bak from the entire directory tree starting at /java/src.

```
<delete>
    <fileset dir="/java/src" includes="**/*.bak" />
</delete>
```

## The mkdir Task

Use the mkdir task to create a directory. This task has just one attribute called dir; and it is required. The dir attribute specifies the directory to create. For example, the following mkdir task creates a directory called \${java\_install}/mysrc.

```
<mkdir dir="${java_install}/mysrc" />
```

## The echo Task

The echo task echoes a message to a file or to System.out.

### Attributes

The echo task has three attributes:

- message – Specifies the message to echo; this attribute is required unless the message is included between the <echo> and </echo> tags.
- file – Indicates the file to which the message is written.
- append – Indicates whether to append to an existing file; the default is false.

### Examples

The following echo tasks echo the message “Welcome to the Ant Build Tool” to System.out:

```
<echo message="Welcome to the Ant Build Tool" />

<echo>
    Welcome to the Ant Build Tool
</echo>
```

## The javac Task

Use the `javac` task to compile a source tree from within the Ant virtual machine (VM).

### Attributes

The `javac` task has many attributes. A few are described here. Refer to the Ant User Manual for details on all attributes.

- `srcdir` – Specifies the location of the `.java` files to compile; this attribute is required unless `<src>` sub-elements are used.
- `destdir` – Specifies the location to store the `.class` files that are created from the compilation.
- `classpath` – Specifies the class path to use during compilation.

### Example

The following `javac` task compiles all `.java` files in the `${mycode}/src` directory and stores the compiled files in `${myclasses}/classes`.

```
<javac srcdir="${mycode}/src  
        destdir="${myclasses}/classes"  
      />
```

## The javadoc Task

The `javadoc` task generates code documentation using the `javadoc` tool.

### Attributes

The `javadoc` task has many attributes. Refer to the Ant User Manual for details on all of them, as well as additional examples. A few attributes are described here:

- `sourcepath` – Specifies where to find the source files; this attribute or `sourcepathref` is required.
- `sourcepathref` – Specifies where to find source files using a reference to a path defined elsewhere; this attribute or `sourcepath` is required.
- `destdir` – Indicates the destination directory for output files; this attribute is required.

- `sourcefiles` – Specifies a list of source files, separated by commas; this attribute or `packagenames` is required.
- `packagenames` – Specifies a list of package files, separated by commas; this attribute or `sourcefiles` is required.

### Example

The following `javadoc` task generates HTML documentation in the directory `/java/docs/api` for source files in the package `java.lang` located in the directory `/java/src` (that is, generates documentation for all files `/java/src/java/lang/*.java`.)

```
<javadoc packagenames="java.lang"
          sourcepath="/java/src"
          destdir="/java/docs/api"
/>
```

## The `jar` Task

Use the `jar` task to create a JAR file.

### Attributes

The following attributes are often used with the `jar` task:

- `jarfile` – Specifies the JAR file to create; this is required.
- `basedir` – Specifies the base directory from which the `jar` command begins. All files under this directory are included in the JAR file unless patterns are used to further define the files to be included in the JAR file.

### Example

The following `jar` task creates a JAR file called `myjar.jar` that includes all files in the tree structure starting at `/java/classes` but excludes any files named `MyTest.class` found in this directory tree.

```
<jar jarfile="myjar.jar" basedir="/java/classes"
      excludes="**/Test.class"
/>
```

## Complete Ant Build File

Code D-1 shows a complete Ant build file that can be used to build a Web application. Filtering is used in the second copy task under the comment "WEBAPP: Copy Static Web Files."

**Code D-1** Complete Ant Build File, build.xml

```

<project name="SL-314 Database Integration Example" default="webapp"
         basedir=". ">

    <!-- ===== Initialize Property Values ===== -->
    <property name="webapp.name"      value="database" />
    <property name="build.dir"        value="${tomcat.home}/webapps" />
    <property name="servlet.jar"      value="${servletapi.home}/lib/servlet.jar" />

    <!-- ===== Convenient Synonyms ===== -->
    <target name="clean" depends="clean-webapp" />
    <target name="all" depends="clean,webapp" />
    <target name="help">
        <echo>
Build structure of the Web Application
Toplevel targets:
    help      -- this message
    clean     -- cleans the whole development environment
    all       -- cleans and then builds the whole system
    webapp   -- (re)builds the Web Application
        </echo>
    </target>

    <!-- ===== WEBAPP: All ===== -->
    <target name="webapp" depends="clean-webapp,compile-webapp" />
    <!-- ===== WEBAPP: Clean WebApp Directory ===== -->
    <target name="clean-webapp">
        <delete dir="${build.dir}/${webapp.name}" />
    </target>
    <!-- ===== WEBAPP: Compile Web Components ===== -->
    <target name="compile-webapp" depends="static-webapp">
        <javac  srcdir="src"
                 destdir="${build.dir}/${webapp.name}/WEB-INF/classes"
                 classpath="${servlet.jar};lib/conn_pool.jar"
                 deprecation="off" debug="on" optimize="off">
        </javac>
    </target>

```

## Basic Built-in Ant Tasks

---

```
<!-- ===== WEBAPP: Copy Static Web Files ===== -->
<target name="static-webapp" depends="prepare-webapp">
    <copy todir="${build.dir}/${webapp.name}">
        <fileset dir="web" />
    </copy>
    <filter token="PB_HOST" value="${pointbase.host}" />
    <copy todir="${build.dir}/${webapp.name}/WEB-INF" filtering="yes">
        <fileset dir="etc" />
    </copy>
    <copy todir="${build.dir}/${webapp.name}/WEB-INF/lib">
        <fileset dir="lib" />
    </copy>
</target>
<!-- ===== WEBAPP: Create WebApp Directories ===== -->
<target name="prepare-webapp">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${build.dir}/${webapp.name}" />
    <mkdir dir="${build.dir}/${webapp.name}/WEB-INF" />
    <mkdir dir="${build.dir}/${webapp.name}/WEB-INF/classes" />
    <mkdir dir="${build.dir}/${webapp.name}/WEB-INF/lib" />
</target>

</project>
```

# Executing Ant

The syntax for running Ant is:

```
ant [options] [target [target2 [target3] ...]]
```

If no arguments or options are specified, Ant looks in the current directory for a build file named `build.xml` and builds the default target, that is, the target whose name matches the value of the `default` attribute of the `project` element. In the code shown in Code D-1, the default target is `webapp`. Ant provides the set of command line options listed in Table D-1.

**Table D-1** Ant Command Line Options

Option	Description
<code>-help</code>	Prints a list of all Ant options
<code>-find file</code>	Searches for the build file <i>file</i> in the current directory, and if not found, searches the parent directory, and its parent directory, and so forth, up to the root of the file system. If <i>file</i> is not specified, the search is for <code>build.xml</code> .
<code>-buildfile file</code>	Uses the specified file <i>file</i> as the build file.
<code>-Dproperty=value</code>	Sets the value of <i>property</i> to <i>value</i> .
<code>-projecthelp</code>	Prints project help information.
<code>-version</code>	Prints the version information and exits.
<code>-quiet</code>	Runs extra quiet.
<code>-verbose</code>	Runs with verbose output.
<code>-debug</code>	Prints debugging information.
<code>-emacs</code>	Produces logging information without adornments.
<code>-logfile file</code>	Uses <i>file</i> for log output.
<code>-logger classname</code>	Specifies that <i>classname</i> is to perform logging.
<code>-listener classname</code>	Adds an instance of <i>classname</i> as a project listener.

## Install Instructions

Source and binary versions of Ant, complete with API documentation and an Ant User Manual are available from  
<http://jakarta.apache.org/ant>.

## System Requirements

The following are required in order to use Ant:

- JAXP-compliant XML parser installed and available on your CLASSPATH. All distributions of Ant provide the reference implementation parser provided with JAXP 1.0, Java Project X.
- JDK™ 1.1 or later installed on your system.

## Environment Setup for Ant

You can download the Ant binary builds as a ZIP file, a GZIP file, or a JAR file. After choosing a directory location for installation of your Ant files, perform an `unzip`, `gunzip`, or `jar xf` command (as appropriate) to extract the Ant software files.

The following directories are created as part of your Ant installation: `bin`, `docs`, and `lib`. To set up your environment to run Ant:

- Add the Ant `bin` directory to your path.
- Set the `ANT_HOME` environment variable to the directory in which you installed Ant.
- Add the file `ant.jar`, and any JAR files or classes needed for your chosen JAXP-compliant XML parser, to your class path.

- If you intend to use Java technology-related tasks, such as `javac` or `javadoc`, then:
  - For JDK 1.1 release, the `classes.zip` file must be added to your class path.
  - For the Java 2 Platform v1.2 or v1.3, the `tools.jar` file must be added to your class path.



**Note** – The scripts supplied in the Ant bin directory add the required Java technology classes automatically if the `JAVA_HOME` environment variable is set.

- If you are executing platform-specific applications, such as the `exec` task or the `cvs` task, set the property `ant.home` to the directory in which you installed the Ant software.



**Note** – The scripts in the Ant bin directory set the value of `ant.home` to the value of the `ANT_HOME` environment variable.

In summary, if you set `ANT_HOME`, `JAVA_HOME` and add the Ant bin directory to your path, the other requirements (settings for class path and the value of the `ant.home` property) will be taken care of automatically when you execute the `ant` script.



# Quick Reference for XML

---

## Objectives

Upon completion of this appendix, you should be able to:

- Define XML
- Describe basic XML syntax
- Differentiate between well-formed XML documents and valid XML documents
- Define and describe DTDs and XML schemas

## Additional Resources



**Additional resources** – The following references provide additional information on the topics described in this module:

- Extensible Markup Language (XML). [Online]. Available at: <http://www.w3.org/XML/>.
- W3C XML Schema. [Online]. Available at: <http://www.w3.org/XML/Schema>.
- St. Laurent, Simon. *XML: A Primer*. MIS: Press, 1998.
- Ceponkus, Alex and Faraz Hoodbhoy. *Applied XML: A Toolkit for Programmers*. New York: John Wiley & Sons, Inc., 1999.
- McLaughlin, Brett. *Java and XML*. Sebastopol: O'Reilly and Associates, Inc., 2000.

# Introduction to XML

XML stands for eXtensible Markup Language. XML 1.0 is a standard of the World Wide Web Consortium (W3C), and the recommendation is available at <http://www.w3.org/TR/REC-xml>. The recommendation describes XML documents, but also briefly addresses how computer programs process (parse) them.

XML does not define any of the basic tags with predefined meanings that exist in HTML. XML allows you to create your own unique tags that are meaningful for your data, thus the term *extensible*.

## Simple Example

The syntax of XML is very much like that of HTML, as shown in Figure E-1.

```
<?xml version="1.0" ?>————— Processing  
<OrderList> instruction, indicates  
  <Customer> XML document  
    <Name>  
      <FirstName>Bill</FirstName>  
      <LastName>King</LastName>  
    </Name>  
    <CardInfo>  
      <CardNum>99999999</CardNum>  
      <Validate>08/31/2002</Validate>  
    </CardInfo>  
  </Customer>  
  <Item>  
    <ItemNo>S-1000</ItemNo>  
    <Title>Shoes</Title>  
    <Look>  
      <Color>Blue</Color>  
      <Size unit="cm">22.5</Size>  
    </Look>  
    <Price unit="yen">15,000</Price>  
  </Item>  
</OrderList>
```

Figure E-1 Simple XML File, `orderlist.xml`

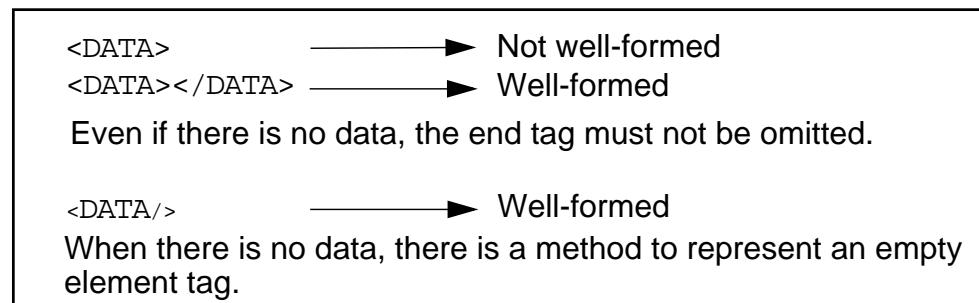
## Basic Syntax

This section describes the syntax of XML documents.

### Well-Formed XML Documents

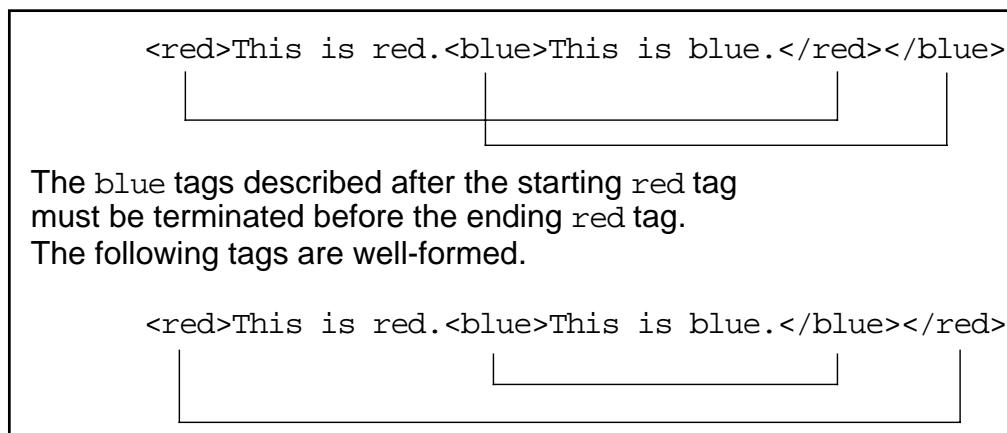
An XML document is said to be well-formed if it adheres to the rules laid out in the XML specification. The following are examples of some of these XML rules:

- Use a start tag and an end tag.  
Do not omit the end tag. If an element has no data, you can use empty element tags. Several examples are shown in Figure E-2.



**Figure E-2** Matching Start and End Tags

- Pay attention to tag nesting. Elements cannot overlap. This is shown in Figure E-3.



**Figure E-3** Elements Cannot Overlap

- A root element must exist.  
You must include a tag called the root element that encloses the entire document body but is not included in other elements.

The root element in Figure E-1 on page E-3 is <OrderList>.



**Note** – Having an explicit root element makes processing of the XML document easier when traversing the relevant tree structure.

- Enclose attribute values in " " or ' ' (double or single quotes).  
An attribute describing an element tag can be specified in any start tag, but the attribute value must be put in double (" ") or single (' ') quotes.

```
<Size unit="cm">22.5</Size>
```

- Characters are case-sensitive in tag and attribute names. (HTML does not care about case.) Uppercase and lowercase characters are distinguished from each other. This is shown in Figure E-4.

<pre>&lt;FirstName&gt;Bill&lt;/FirstName&gt; &lt;FirstNAME&gt;Bill&lt;/FirstNAME&gt; &lt;firstName&gt;Bill&lt;/firstName&gt;</pre>	These tags are all recognized differently.
<pre>&lt;firstname&gt;Bill&lt;/firstName&gt;</pre>	This is not well-formed because the start tag does not match the end tag.

**Figure E-4** Case Sensitivity

## Validity and DTDs

The XML specification defines an XML document as valid if it has an associated document type declaration (DTD) and if the document complies with the constraints expressed in it.

A DTD defines the data structure of an XML document, such as the order in which tags should be specified in XML documents and which tags and how many tags are to be specified. Think of the DTD as the vocabulary and syntax rules governing your XML documents.

Figure E-5 shows a possible DTD for the XML document `orderlist.xml` from Figure E-1 on page E-3.

```
<?xml version="1.0" ?>
<!DOCTYPE OrderList[
  <!ELEMENT OrderList(Customer, Item+)>
  <!ELEMENT Customer(Name, CardInfo+)>
    <!ELEMENT Name (FirstName, MiddleName?, LastName)>
      <!ELEMENT FirstName (#PCDATA)>
      <!ELEMENT MiddleName (#PCDATA)>
      <!ELEMENT LastName (#PCDATA)>
    <!ELEMENT CardInfo (CardNum, Validate)>
      <!ELEMENT CardNum (#PCDATA)>
      <!ELEMENT Validate (#PCDATA)>

    <!ELEMENT Item (ItemNo, Title, Look, Price)*>
      <!ELEMENT ItemNo (#PCDATA)>
      <!ELEMENT Title (#PCDATA)>
      <!ELEMENT Look (Color*, Size)>
        <!ELEMENT Color (#PCDATA)>
        <!ELEMENT Size (#PCDATA)>
        <!ATTLIST Size unit CDATA "0">
      <!ELEMENT Price (#PCDATA)>
      <!ATTLIST Price unit CDATA "0">
    ]>
  <OrderList>
  ...
</OrderList>
```

**Figure E-5** Sample DTD for `orderlist.xml`

## DTD-specific Information

The main syntax for DTDs is described here.

- Document type declaration `<!DOCTYPE...>` – Makes a declaration to describe a DTD. Within this declaration, the declaration of the element type as an XML document structure and other declarations are made.
  - Identifies a *root* element name after the string `<!DOCTYPE`.
  - Describes the DTD within the document type declaration.
- Element type declaration `<!ELEMENT...>` – Declares what elements are specified, the order the elements are specified, and the number of elements.
  - Describes element names after the string `<!ELEMENT` (see Figure E-6).

<code>&lt;!ELEMENT Customer (Name, CardInfo+)&gt;</code>	Include the Name and CardInfo, tags in the Customer tag.
<code>&lt;!ELEMENT Line EMPTY&gt;</code>	The Line element is an empty element tag, and EMPTY is a fixed specification.
<code>&lt;!ELEMENT Other ANY&gt;</code>	The contents of any element can be included in the Other tag.
<code>&lt;!ELEMENT FirstName (#PCDATA)&gt;</code>	Mixed-content declaration.

**Figure E-6** The ELEMENT Declaration

- Attribute list declaration `<!ATTLIST...>` – For certain elements, declares an attribute name, an attribute type, and a default value for the attribute (see Figure E-7).

<code>&lt;!ATTLIST</code>	<code>Size</code>	<code>unit</code>	<code>CDATA</code>	<code>"0"&gt;</code>
	Element name	Attribute name	Attribute type	Default value

**Figure E-7** The ATTRIBUTE Declaration

Additional notation is used to indicate the multiplicity of sub-elements within an element declaration. This notation is listed in Table E-1.

**Table E-1** Multiplicity Indicators

Symbol	Multiplicity
+	Repeat one or more times
*	Repeat zero or more times
?	Indicates an element that is optional; it occurs once or not at all.

Examples:

```
<!ELEMENT Customer (Name, CardInfo+)>  
<!ELEMENT Name (FirstName, MiddleName?, LastName)>  
<!ELEMENT Item (ItemNo, Title, Look, Price)*>
```

Additionally, there are two pre-defined, special element types. These are listed in Table E-2.

**Table E-2** Pre-Defined Element Types

Symbol	Description
#PCDATA	The character data to be analyzed. Child elements and character data can be included in elements. It is also called <i>parsed character data</i> . It is processed (parsed) by an XML parser.
CDATA	Represents character data that is not analyzed (not parsed).

# Schemas

XML schema has been developed by the W3C. Schemas provide a means for defining the structure, content, and semantics of XML documents.

The XML schema specification is composed of three parts:

- Part 0: Primer – Provides an easily readable description of the XML schema facilities and enables the developer to quickly understand how to create schemas using the XML schema language.
- Part 1: Structures – Specifies the XML schema definition language, which offers facilities for describing the structure and constraining the contents of XML 1.0 documents, including those which exploit the XML namespace facility.
- Part 2: Datatypes – Defines a means to apply datatypes to elements and attributes.

Unlike DTDs, XML schemas adhere to the XML specification (are themselves well-formed XML documents) and better support XML namespaces. The `schema` element is the first element in an XML schema file and the prefix `xsd:` is used to indicate the XML schema namespace:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
</xsd:schema>
```

The main sub-elements in an XML Schema document are:

- `element` – Declares an element.
- `attribute` – Declares an attribute.
- `complexType` – Defines elements that can contain other elements and attributes.
- `simpleType` – Defines a new simple type based on simple types defined in the XML schema specification. For example, you can define a new simple type based on the type “integer” by restricting the range of integers allowed.

Simple types cannot contain other elements or attributes.

Figure E-8 shows an XML schema syntax for parts of the XML document in Figure E-1 on page E-3.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"> - schema element  
  <xsd:annotation> - and XML schema  
    <xsd:documentation xml:lang="en"> - namespace  
      OrderList schema for orderlist.xml  
    </xsd:documentation>  
  </xsd:annotation>  
  ...  
</xsd:schema>
```

Used to provide description of the document

```
<xsd:element name="OrderList" type="OrderListInfo"/> - Schema syntax  
  <xsd:complexType name="OrderListInfo">  
    <xsd:sequence>  
      <xsd:element name="Customer" type="CustomerInfo"/>  
      <xsd:element name="Item" type="ItemInfo"/>  
    </xsd:sequence>  
</xsd:complexType>
```

for OrderList element

```
<xsd:complexType name="ItemInfo">  
  <xsd:sequence>  
    <xsd:element name="ItemNo" type="xsd:string"/>  
    <xsd:element name="Title" type="xsd:string"/>  
    <xsd:element name="Look" type="LookInfo"/>  
    <xsd:element name="Price" type="xsd:string"/>  
    <xsd:element name="ItemNo" type="xsd:string"/>  
  </xsd:sequence>  
</xsd:complexType>
```

Schema syntax  
for ItemInfo element

**Figure E-8** XML Schema for orderlist.xml

The OrderList element is composed of Customer and Item elements, both are complex types because these can contain other elements. Therefore, the complex types for these CustomerInfo and ItemInfo have to be specified. The ItemInfo type includes another complex type called LookInfo. Sample syntax for this type is shown in Figure E-9 on page E-11.

```

<xsd:complexType name="LookInfo">
  <xsd:sequence>
    <xsd:element name="Color" type="xsd:string" />
    <xsd:element name="Size" >
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:decimal" >
            <xsd:attribute name="unit" type="xsd:string" />
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

```

Inline  
anonymous  
type

**Figure E-9** LookInfo Element Schema Syntax

The LookInfo complex type contains the sub-elements Color and Size. The Size element is allowed to have an attribute unit (refer to Figure E-1 on page E-3), so Size must be a complex type. Recall that only complex types can have an attribute associated with them.

Instead of naming this complex type, an inline definition is provided. The Size element does not contain sub-elements, so it is defined as simpleContent of decimal type data.

**Note** – Refer to the XML Schema Primer for a detailed XML example and schema file, complete with full explanations of all syntax and terminology.





# Quick Reference for UML

---

This module is designed to serve as a quick reference for the Unified Modeling Language (UML).

## Additional Resources



**Additional resources** – The following references can provide additional details on the topics discussed in this appendix:

- Booch, Rumbaugh, James Grady, and Ivars Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison-Wesley, 1999. [ISBN: 0-201-57168-4]
- Alhir, Si. *UML in a Nutshell*. Sebastopol: O'Reilly and Associates, Inc., 1998. [ISBN: 1-56592-448-7]
- Fowler, Martin. *UML Distilled*. Reading: Addison-Wesley, 1997. [ISBN: 0-201-32563-2]
- *UML v1.3 Specification*. 2000. [Online]. Available at: [http://www.omg.org/technology/documents/formal/omg\\_modeling\\_specifications\\_avai.htm](http://www.omg.org/technology/documents/formal/omg_modeling_specifications_avai.htm)
- Additional UML resources are available online at: <http://www.omg.org/uml/>.

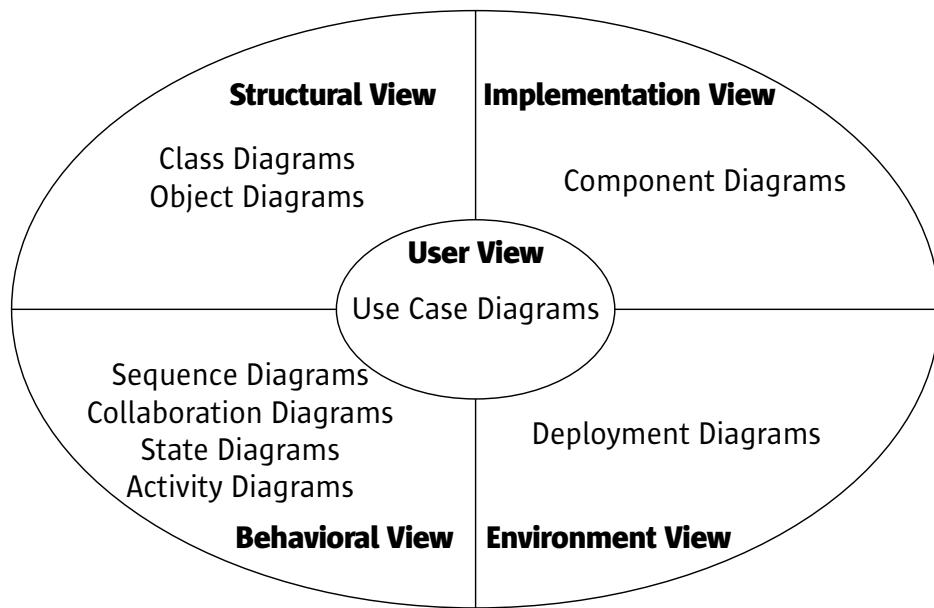
## What Is UML?

The Unified Modeling Language (UML) is a graphical language for modeling software systems. It was created in the early 1990's by three leaders in the object modeling world: Grady Booch, James Rumbaugh, and Ivars Jacobson. Their goal was to unify the three major modeling languages at the time: the Booch method, Object Modeling Technique (OMT), and Object-Oriented Software Engineering (OOSE, best known for the introduction of Use Case analysis). UML is now a standard of the Object Management Group (OMG); version 1.3 was adopted by the OMG November 1999. There is a Request for Proposal (RFP) for UML version 2.0.

## Modeling in UML

A system's architecture and design enables you to manage diverse viewpoints about the system and control the overall system development process. For example, different stakeholders of the system bring different agendas to a project and look at the project in different ways. The agenda of an end user is different from that of a developer or tester.

UML allows you to model a system for many perspectives or views (see Figure F-1). The diagrams generated from the processes of project management are the long-lived artifacts that document the system's purpose, scope, structure, behavior, deployment strategy, and so on.



**Figure F-1** UML Views and Diagrams

## User View

The user view represents the system from the perspective of the users of the system. A Use Case Diagram represents the functionality provided by the system to external users. The Use Case Diagram is composed of actors, use cases, and their relationships.

## Structural View

The structural view represents the static aspects of the system. Class Diagrams represent the static structure of a system as it is declared. These diagrams are composed of classes and their relationships.

Object Diagrams represent the static structure of a system at a particular instance in time. These diagrams are composed of object nodes and associations.

## Behavioral View

The behavioral view represents the dynamic aspects of the system. A Sequence Diagram represents a time sequence of messages exchanged between several objects to achieve a particular behavior.

A Collaboration Diagram represents a particular behavior shared by several objects. These diagrams are composed of objects, their associations, and the message exchanges that accomplish the behavior.

A State Diagram represents the states and responses of a class to external and internal triggers. This can also be used to represent the life cycle of an object.

An Activity Diagram represents the activities or actions of a process without regard to the objects that perform these activities.

## Implementation View

The implementation view represents the structural and behavioral aspects of the system's realization. A Component Diagram represents the organization and dependencies among software implementation components.

## Environment View

The environment view represents the physical space in which the system is realized. A Deployment Diagram represents the network of processing resource elements and the configuration of software components on each physical element.

## General Elements

In general, UML diagrams represent concepts, depicted as symbols (also called nodes), and relationships among concepts, depicted as paths (also called links) connecting the symbols. These nodes and links are specialized for the particular diagram. For example, in Class Diagrams, the nodes represent object classes and the links represent associations between classes and generalization (inheritance) relationships.

There are other elements that augment these diagrams. This section describes the following elements: packages, stereotypes, annotations, constraints, and tagged values.

## Packages

In UML, packages allow you to arrange your modeling elements into groups. UML packages are a generic grouping mechanism and should not be directly associated with Java technology packages. However, they can be used to model Java technology packages. Figure F-2 demonstrates a package diagram that contains a group of classes in a class diagram.

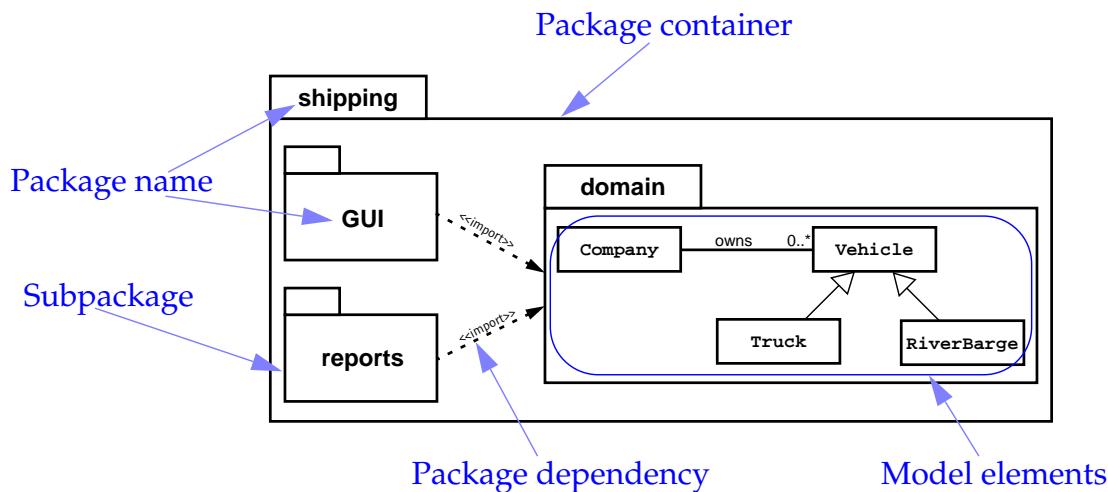


Figure F-2 Example Package

## Mapping to Java Technology Packages

The mapping to Java technology packages implies that the classes would contain the package declaration of package `shipping.domain`. For example, in the file `Vehicle.java`:

```
package shipping.domain;

public class Vehicle {
    // declarations
}
```

Figure F-2 on page F-5 also demonstrates a simple hierarchy of packages. The shipping package contains three subpackages: GUI, reports, and domain. The dashed arrow from one package to another indicates that the package at the tail of the arrow “uses” (imports) elements in the package at the head of the arrow. For example, reports uses elements in the domain package:

```
package shipping.reports;

import shipping.domain.*;

public class VehicleCapacityReport {
    // declarations
}
```



**Note** – In Figure F-2 on page F-5, the shipping.GUI and shipping.reports packages have their names in the body of the package box rather than in the head of the package box. This is done only when the diagram does not expose any of the elements in that package.

---

## Stereotypes

The designers of UML understood that they could not build a modeling language that would satisfy every programming language and every modeling need. They built several mechanisms into UML to allow modelers to create their own semantics for modeling elements (nodes and links). Figure F-3 shows the use of a stereotype tag <<interface>> to declare that the class node Set is a Java technology interface declaration. Stereotype tags can adorn relationships as well as nodes. There are over a hundred standard stereotypes, and you can create your own to model your own semantics.

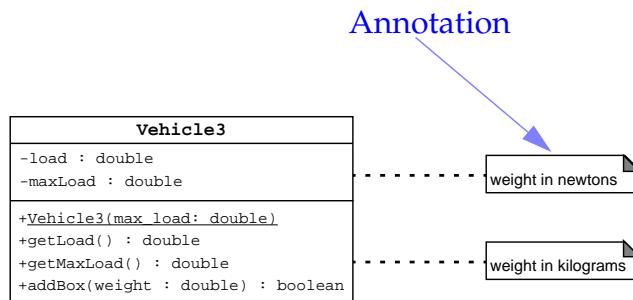
Stereotype tag



**Figure F-3** Example Stereotype

## Annotation

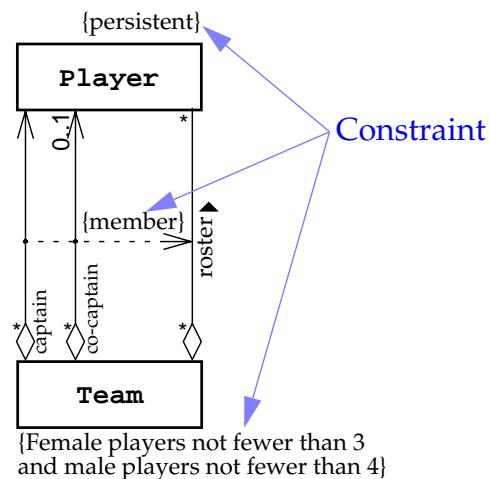
The designers of UML also built into the language a method for annotating the diagrams. Figure F-4 shows a simple annotation.



**Figure F-4** Example Annotation

## Constraints

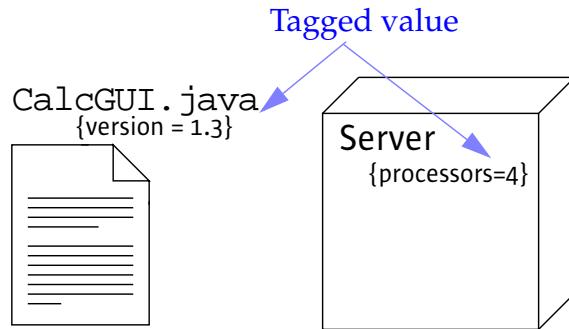
Constraints allow you to model certain conditions that apply to a node or link. Figure F-5 shows several constraints. The topmost constraint specifies that the `Player` objects must be stored in a persistent database. The middle constraint specifies that the captain and co-captain must also be members of the team's roster. The constraint on the bottom specifies the minimum number of players by gender.



**Figure F-5** Example Constraints

## Tagged Values

Tagged values allow you to add new properties to nodes in a diagram. Figure F-6 shows several examples.

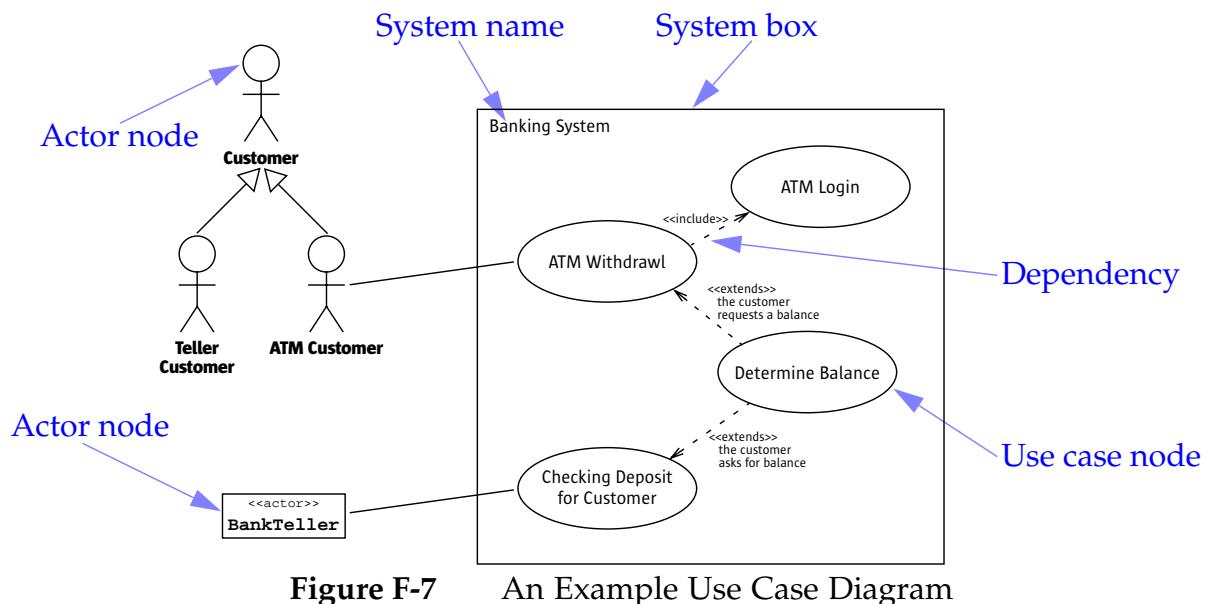


**Figure F-6** Example Tagged Values

# Use Case Diagrams

A Use Case Diagram represents the functionality provided by the system to external users. The Use Case Diagram is composed of actors, use case nodes, and their relationships. Actors can be humans or other systems (see Figure F-7).

Figure F-7 shows a simple banking Use Case Diagram. An actor node can be denoted as a “stick figure” (as in the three Customer actors) or as a class node (see “Class Nodes” on page F-10) with the stereotype of <>actor>. There can be a hierarchy of actors.



**Figure F-7** An Example Use Case Diagram

A use case node is denoted by a labeled oval. The label indicates the activity summary that the system performs for the actor. Use case nodes are grouped into a system box, which is usually labeled in the top left corner. The relationship “actor uses the system to” is represented by the plain, solid line from the actor to the use case node.

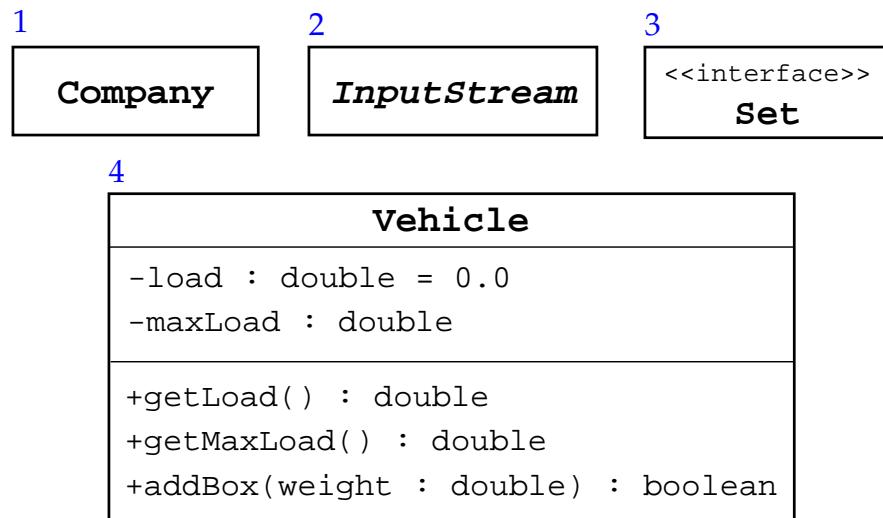
Use case nodes can depend on other use cases. For example, the “ATM Withdrawl” Use Case uses the “ATM Login” Use Case. Use case nodes can also extend other use cases to provide additional functionality. For example, the “Checking Deposit for Customer” use case node extends the “Determine Balance” use case node. The notation for extending a use case node uses a dependency relation and not a generalization relation.

# Class Diagrams

A Class Diagram represents the static structure of a system as it is declared. These diagrams are composed of classes and their relationships.

## Class Nodes

Figure F-8 shows several *class nodes*. You do not have to model every aspect of an entity every time that entity is used. A class node can just be the name of the class, as in Examples 1, 2, and 3. Example 1 is a concrete class, where no members are modeled. Example 2 is an abstract class (name is in italics). Example 3 is an interface. Example 4 is a concrete class, where members are modeled.



**Figure F-8** Several Class Nodes

A fully specified class node has three basic compartments: the name of the class in the top, the set of attributes under the first bar, and the set of methods under the second bar. A complex example of a class node is illustrated in Figure F-9.

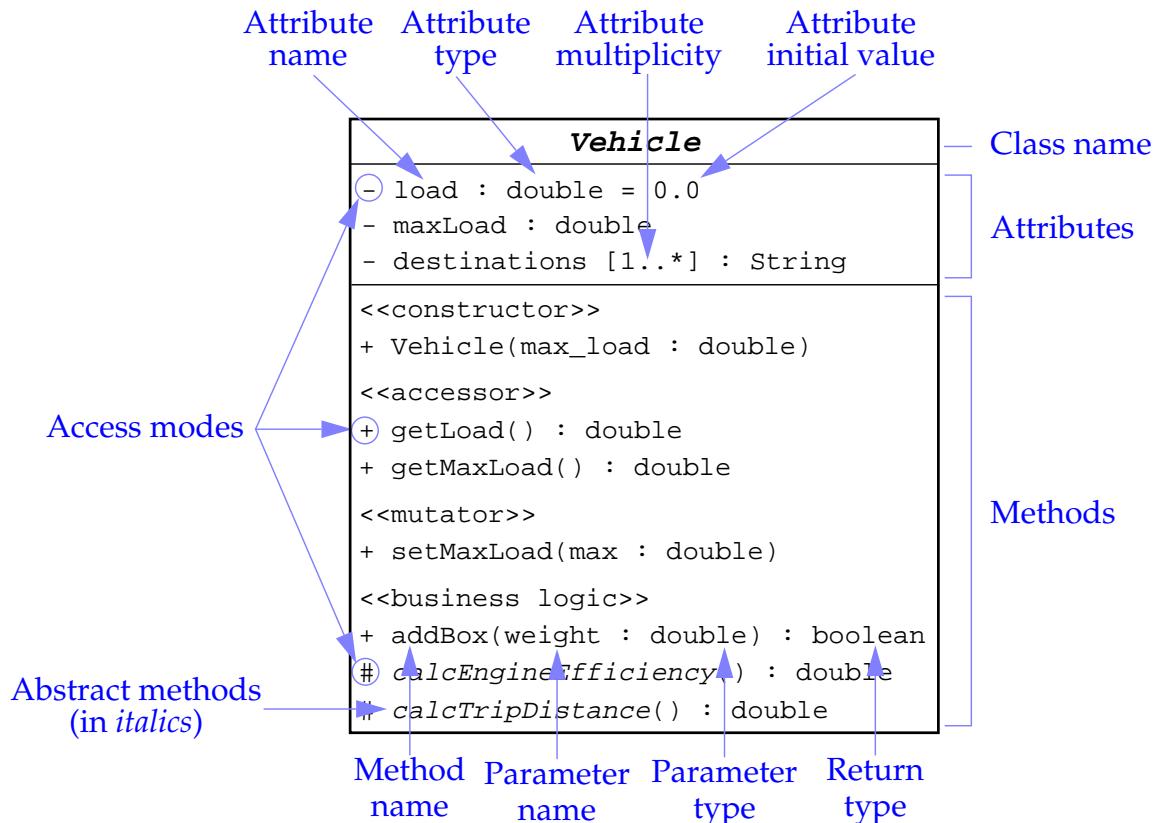


Figure F-9 Elements of a Class Node

An attribute is specified by five elements: access mode, name, multiplicity, data type, and initial value; all but the name are optional. A method is specified by four elements: access mode, name, parameter list (a comma-delimited list of parameter name and type), and the return type; all but the name are optional. If the return value is not specified, then no value is returned (`void`). The name of an abstract method is marked in *italics*.

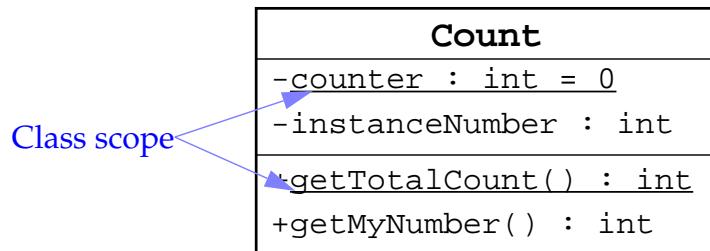
Stereotypes can be used to group attributes or methods together. For example, accessor, mutator, and business logic methods can be separated from each other for clarity. And because there is no UML-specific notation for constructors, you can use the `<<constructor>>` stereotype to mark the constructors in your method block.

Table F-1 shows the valid UML access mode symbols. There is no symbol for the Java technology default access mode. You can simply leave the access mode blank on the UML declaration (as you would in the Java technology declaration), or you could make up your own symbol.

**Table F-1** UML Defined Access Modes and Their Symbols

access mode	symbol
private	-
protected	#
public	+

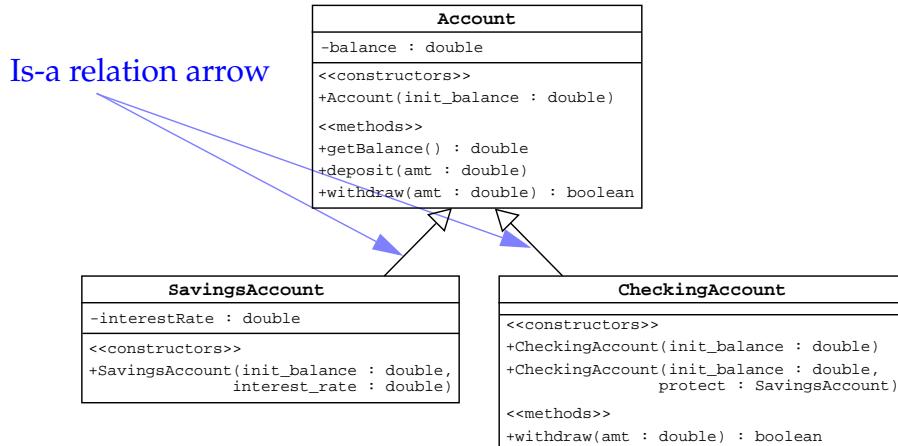
Figure F-10 shows an example class node with elements that have class (or static) scope. This is denoted by the underline under the element. For example, counter is a static data attribute and getTotalCount is a static method.



**Figure F-10** An Example Class Node With Static Elements

## Inheritance

Figure F-11 shows class inheritance through the “is-a” relationship arrow.



**Figure F-11** Class Inheritance Relationship

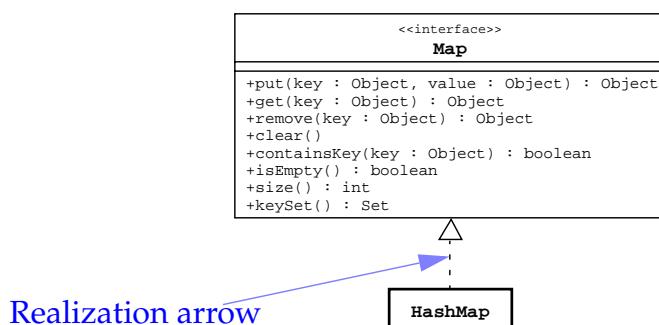
This is implemented in the Java programming language with the `extends` keyword. For example:

```

public class SavingsAccount extends Account {
    // declarations here
}
  
```

## Interface Implementation

Figure F-12 shows how to model a class implementing an interface, using the “realization” arrow.



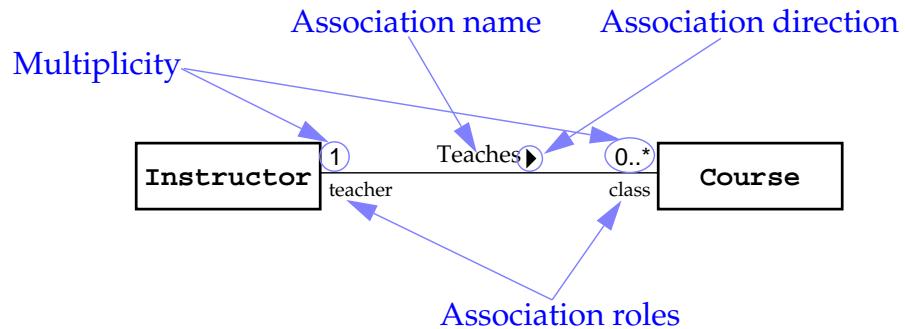
**Figure F-12** An Example of a Class Implementing an Interface

```

public class HashMap implements Map {
    // definitions here
}
  
```

## Association, Roles, and Multiplicity

Figure F-13 shows an example association. An *association* is a link between two types of objects and implies the ability for the code to navigate from one object to another.



**Figure F-13** Class Associations and Roles

In this diagram, “Teaches” is the name of the association with a directional arrow pointing to the right. This association can be read as “an instructor teaches a course.” You can also attach roles to each end of the association. Here the “teacher” role indicates that the instructor is the teacher for a given course. All of these elements are optional if the association is obvious.

This example also demonstrates how many objects are involved in each role of the association. This is called *multiplicity*. In this example, there is only one teacher for every class; this is denoted by the “1” on the Instructor side of the association. Also any given teacher may teach zero or more courses; this is denoted by the “0..\*” on the Course side. You can leave out the multiplicity for a given role if it is always one. You can also abbreviate “zero or more” as just “\*”. Multiplicity can be a range of values (for example, “2..10” means “at least 2 and up to 10”), a disjoint set of values (for example, “2,4,6,8,10”), or a disjoint set of values or ranges (“1..3,6,8..11”). However, the most common values are exactly one (“1” or left blank), zero or one (“0..1”), zero or more (“\*”), or one or more (“1..\*”).

Associations are typically represented in the Java programming language as an attribute in the class at the tail of the relationship (specified by the direction indicator). If the multiplicity is greater than one, then some sort of collection or array is necessary to hold the elements.

Also, in Figure F-13 on page F-14 the association between an instructor and a course might be represented in the Instructor class as:

```
public class Instructor {
    private Course[] classes = new Course[MAXIMUM];
}
```

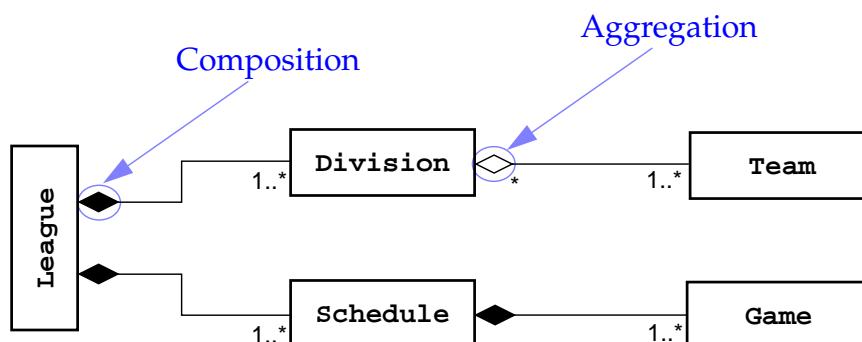
or as:

```
public class Instructor {
    private List classes = new ArrayList();
}
```

The latter representation is preferable if you do not know the maximum number of courses any given instructor might teach.

## Aggregation and Composition

An *aggregation* is an association in which one object contains a group of parts that make up the “whole” object (see Figure F-14). For example, a car is an aggregation of an engine, wheels, body, and frame. Composition is a specialization of aggregation in which the parts cannot exist independently of the “whole” object. For example, a human is a composition of a head, two arms, two legs, and a torso; if you remove any of these (without surgical intervention), the whole person is going to die.

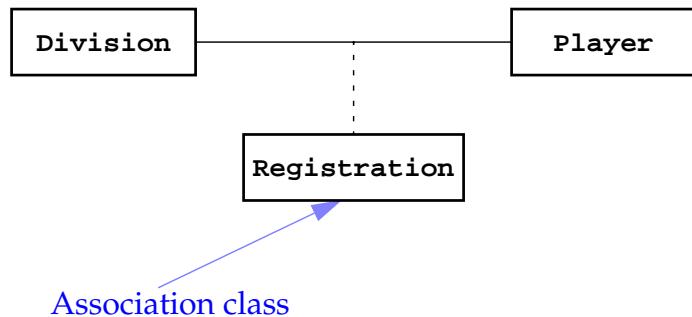


**Figure F-14** Example Aggregation and Composition

In the example in Figure F-14, a sports league (a sports event that occurs seasonally every year) is a composition of divisions and schedules. A division is an aggregation of teams. A team may exist independently of a particular season; in other words, a team may exist for several seasons (leagues) in a row. Therefore, a team may still exist even if a division is eliminated. Moreover, a game can only exist in the context of a particular schedule of a particular league.

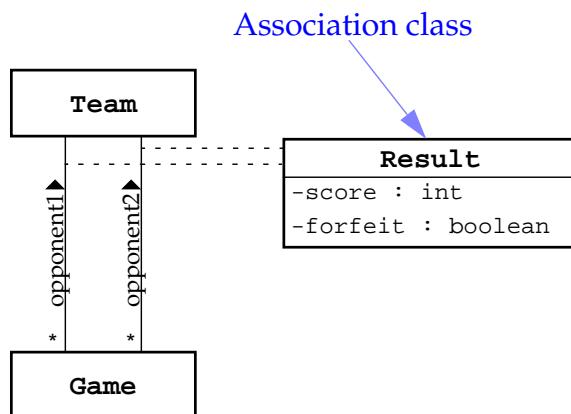
## Association Classes

An association between two classes may have properties (and behavior) of its own. Association classes are used to model this characteristic. For example, players may be required to register for a particular division within a sports league; Figure F-15 shows this relationship. The association class is attached to the association link by a dashed line.



**Figure F-15** A Simple Association Class

Figure F-16 shows an association class that is used by two associations and that it has two private attributes. This example indicates that a Game object is associated with two opposing teams and each team will have a score and a flag specifying whether that team forfeited the game.



**Figure F-16** A More Complex Association Class

An association class can be represented using the Java programming language in several different ways. One way is to code the association class as a standard Java technology class. For example, registration could be coded as:

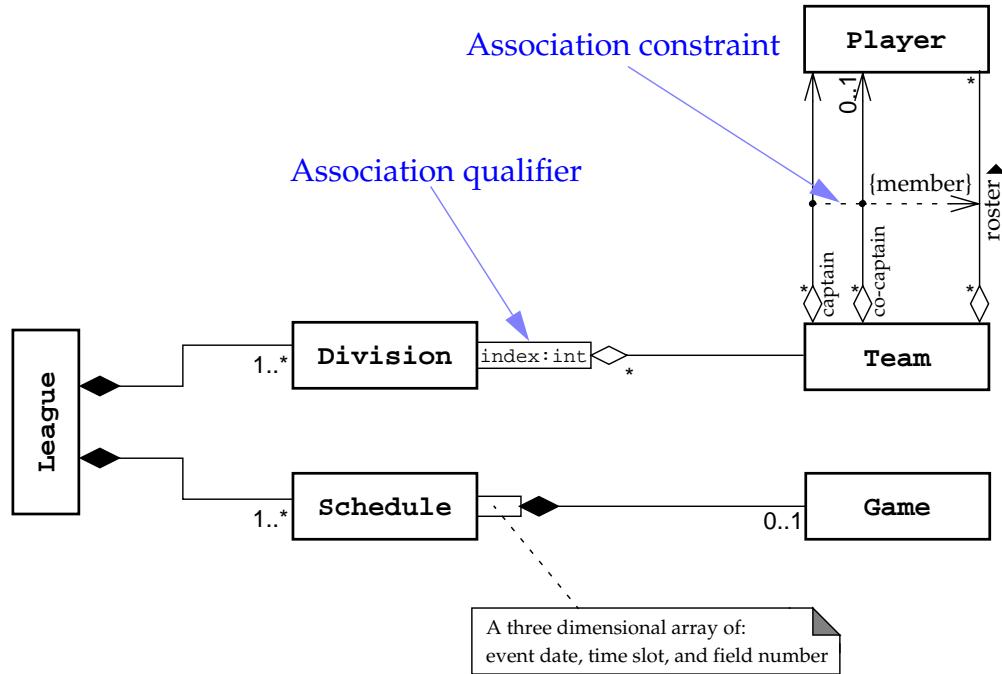
```
public class Registration {  
    private Division division;  
        private Player player;  
    // registration data  
    // methods...  
}  
public class Division {  
    // data  
    public Registration retrieveRegistration(Player p) {  
        // lookup registration info for player  
        return new Registration(this, p, ...);  
    }  
    // methods...  
}
```

Another technique is to code the association class attributes directly into one of the associated classes. For example, the Game class might include the score information as follows:

```
public class Game {  
    // first opponent and score details  
    private Team opponent1;  
    private int opponent1_score;  
    private boolean opponent1_forfeit;  
    // second opponent and score details  
    private Team opponent2;  
    private int opponent2_score;  
    private boolean opponent2_forfeit;  
    // methods...  
}
```

## Other Association Elements

There are several other parts of associations. In this section, you see two important elements: constraints and qualifiers (see Figure F-17).



**Figure F-17** Other Associations Elements

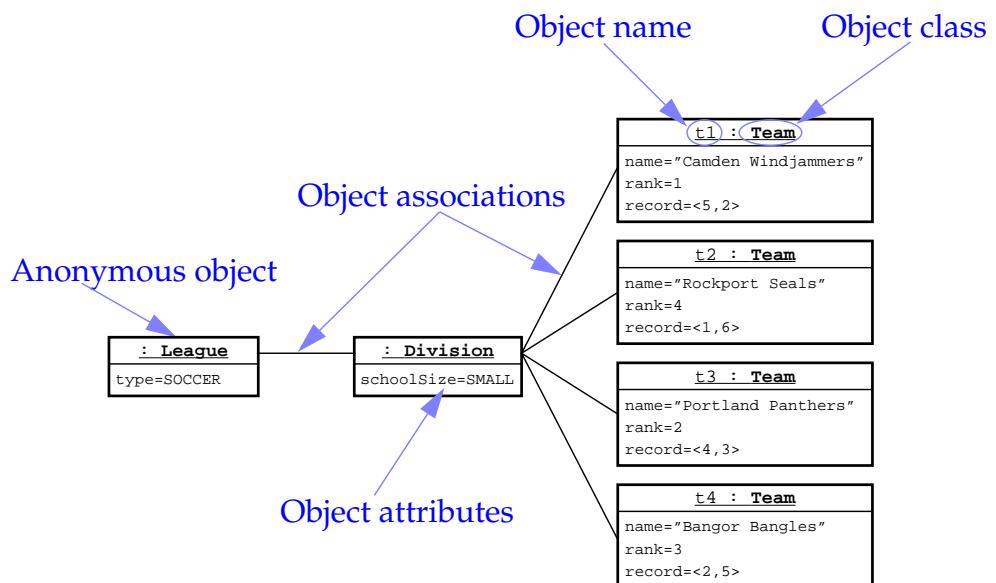
An association constraint allows you to augment the semantics of two or more associations by attaching a dependency arrow between them and tagging that dependency with a constraint. For example in Figure F-17, the captain and co-captain of a team are also members of the team's roster.

An association qualifier provides a modeling mechanism to state that an object at one end of the association can lookup another object at the other end. For example, a particular game in a schedule is uniquely identified by an event date (for example, Saturday August 12th, 2000), a time-slot (for example, 11:00am to 12:30pm), and a particular field number (for example, field #2). One particular implementation might be a three dimension array (for example, Game[ ][ ][ ]) in which each qualifier element (date, time-slot, field#) is mapped to an integer index.

# Object Diagrams

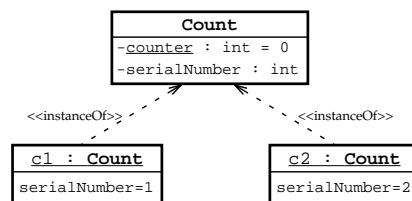
An Object Diagram represents the static structure of a system at a particular instance in time. These diagrams are composed of object nodes, associations, and sometimes class nodes.

Figure F-18 shows a hierarchy of objects that represent a set of teams in a single division in a soccer sports league. This diagram shows one configuration of objects at a specific point of time in the system. Object nodes only show instance attributes because methods are elements of the class definition. Also, an Object Diagram does need not to show every associated object; it just needs to be representative.



**Figure F-18** An Example Object Diagram

In Figure F-19, two objects are shown, **c1** and **c2**, with their instance data. They refer to the class node for **Count**, and the dependency arrow indicates that the object is an instance of the class **Count**. The objects do not include the counter attribute because it has class scope.



**Figure F-19** An Example Object Diagram

# Sequence Diagrams

A Sequence Diagram represents a time sequence of messages exchanged between several objects to achieve a particular behavior. A Sequence Diagram allows you to understand the flow of messages and events that occur in a certain process or collaboration. In fact, a Sequence Diagram is just a time-ordered view of a Collaboration Diagram (see page F-22).

Figure F-20 shows the elements of a Sequence Diagram. An important aspect of this type of diagram is that time is moving from top to bottom. This diagram shows the time-based interactions between a set of objects (or roles). Roles can be named or anonymous and usually have a class associated with them. Roles can also be actors.

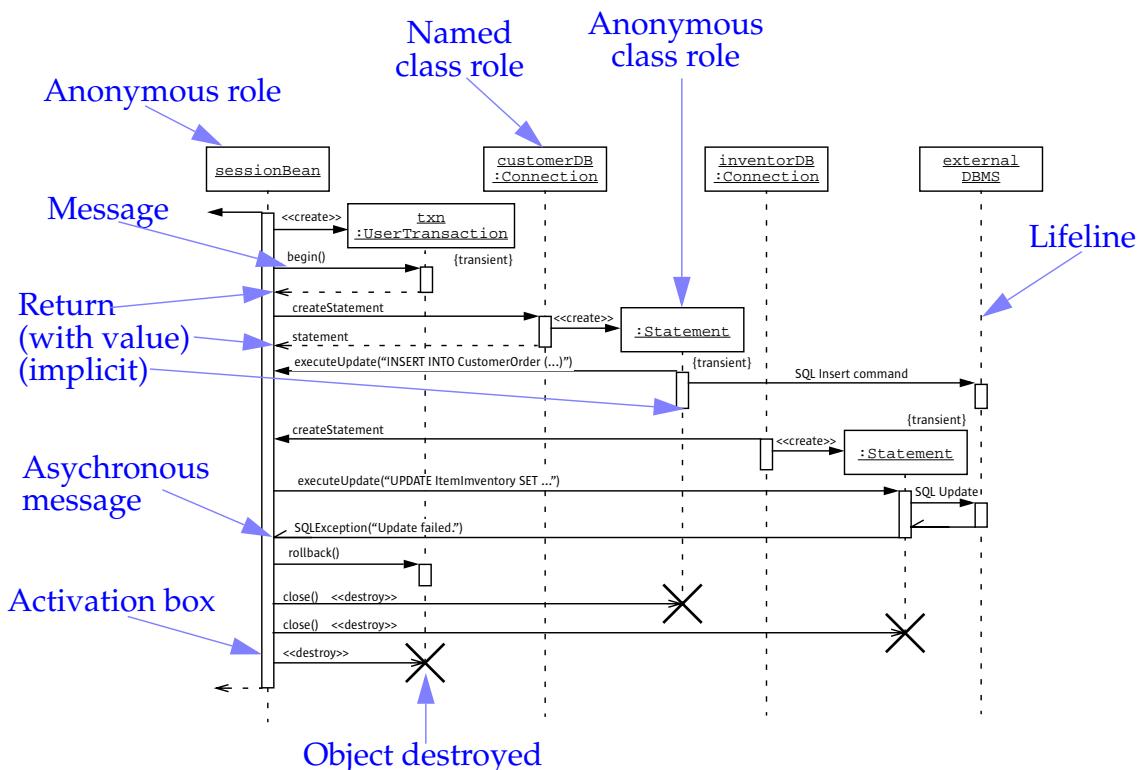


Figure F-20 An Example Sequence Diagram

Every role has a lifeline that extends from the base of the object node vertically. Roles at the top of the diagram existed before the entry message (into the left-most role).

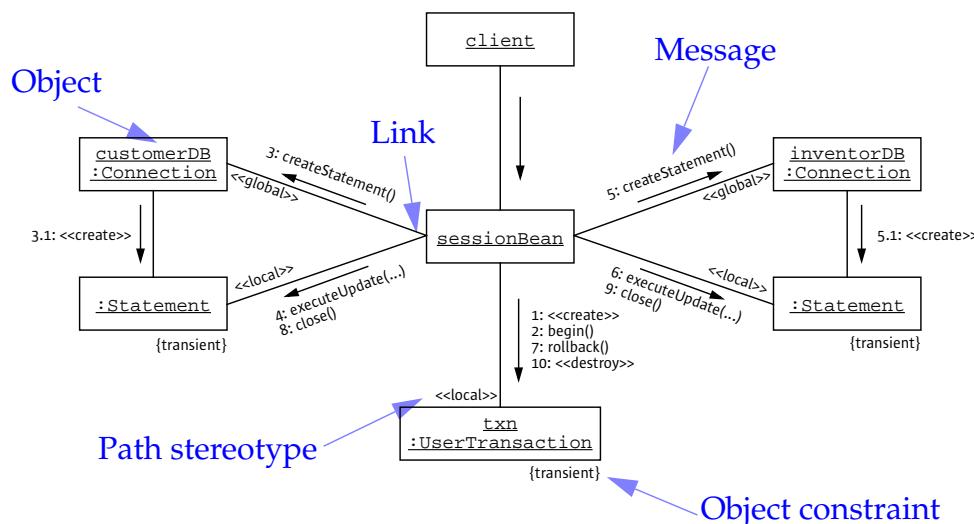
A message initiates an activation. A message is usually associated with a particular operation or method. A return message indicates that the activation is complete. A return value may be specified. The return message may also be left implicit; the end of the activation bar indicates that the operation has halted. The destroy message is a special operation that terminates the object. The lifeline ends with a cross at the end point of the destroy message.

This example illustrates an EJB Session bean that performs two database modifications within a single transaction context. The first database insert succeeds, but the second one fails. This model shows how the Session bean manages the failure by rolling back the transaction and then cleaning up (destroying the two local SQL statement objects).

# Collaboration Diagrams

A Collaboration Diagram represents a particular behavior shared by several objects. These diagrams are composed of objects, their associations, and the message exchanges that accomplish the behavior. A Collaboration Diagram is essentially a different view of a Sequence Diagram (see page F-20).

Figure F-21 shows a Collaboration Diagram for the Sequence Diagram in Figure F-20 on page F-20. Collaboration Diagrams are built like Object Diagrams with messages annotating the links between objects.



**Figure F-21** An Example Collaboration Diagram

The messages are numbered to give you the order-based sequence of the methods calls.

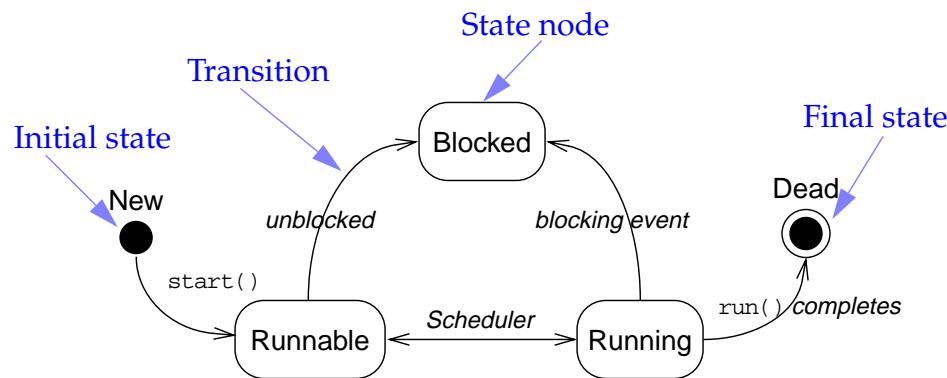
The links can be labeled with a stereotype to indicate if the object is global or local to the call sequence. In this example, the connection objects are global, and the statement and transaction objects are local.

Objects can also be labeled with a constraint to indicate if the object is transient.

# State Diagrams

A State Diagram represents the states and responses of a class to external and internal triggers. This can also be used to represent the life cycle of an object. The definition of an object state is dependent on the type of object and the level of depth you want to model.

Figure F-22 shows an example State Diagram. Every State Diagram should have an initial state (the state of the object at its creation) and a final state. By definition, no state can transition into the initial state and the final state cannot transition to any other state.



**Figure F-22** An Example State Transition Diagram

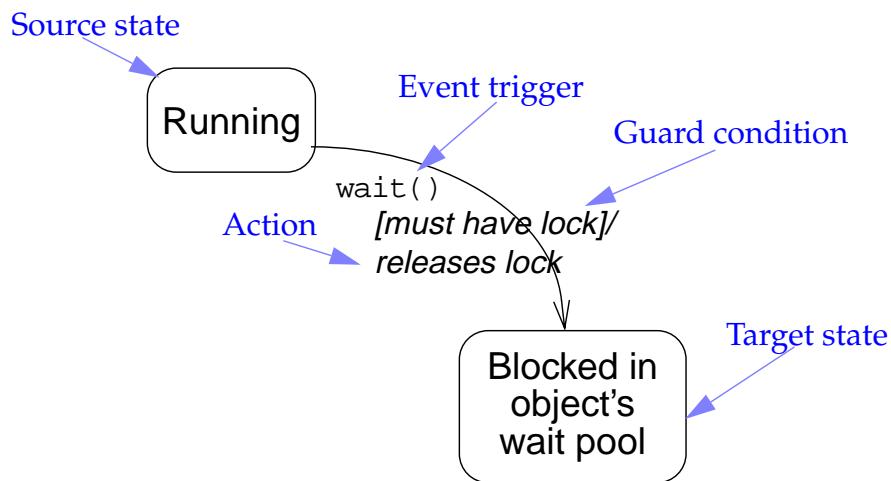
There is no pre-defined way of implementing a State Diagram. For complex behavior, you might consider using the State design pattern.

## Transitions

A transition has five elements:

- Source state – The state affected by the transition
- Event trigger – The event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied
- Guard condition – A Boolean expression used to determine if the state transition should be made when the event trigger occurs
- Action – A computation or operation performed on the object making the state transition
- Target state – The state that is active after the completion of the transition

A detailed transition is illustrated in Figure F-23.

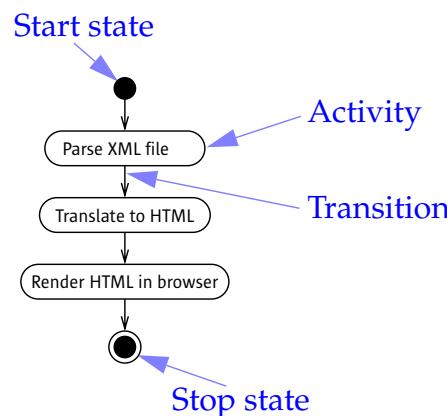


**Figure F-23** An Example State Transition

# Activity Diagrams

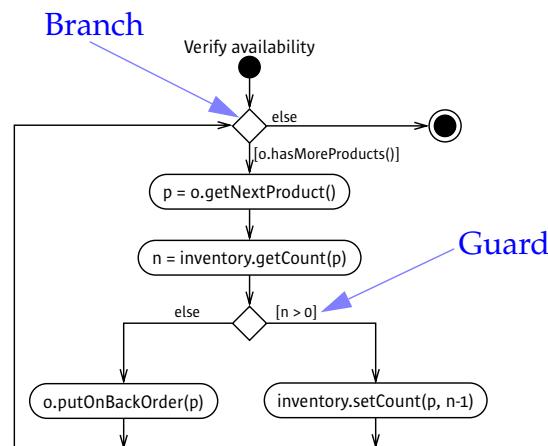
An Activity Diagram represents the activities or actions of a process without regard to the objects that perform these activities.

Figure F-24 shows the elements of an Activity Diagram. An Activity Diagram is similar to a flowchart. There are activities and transitions between them. Every Activity Diagram starts with a single start state and ends in a single stop state.



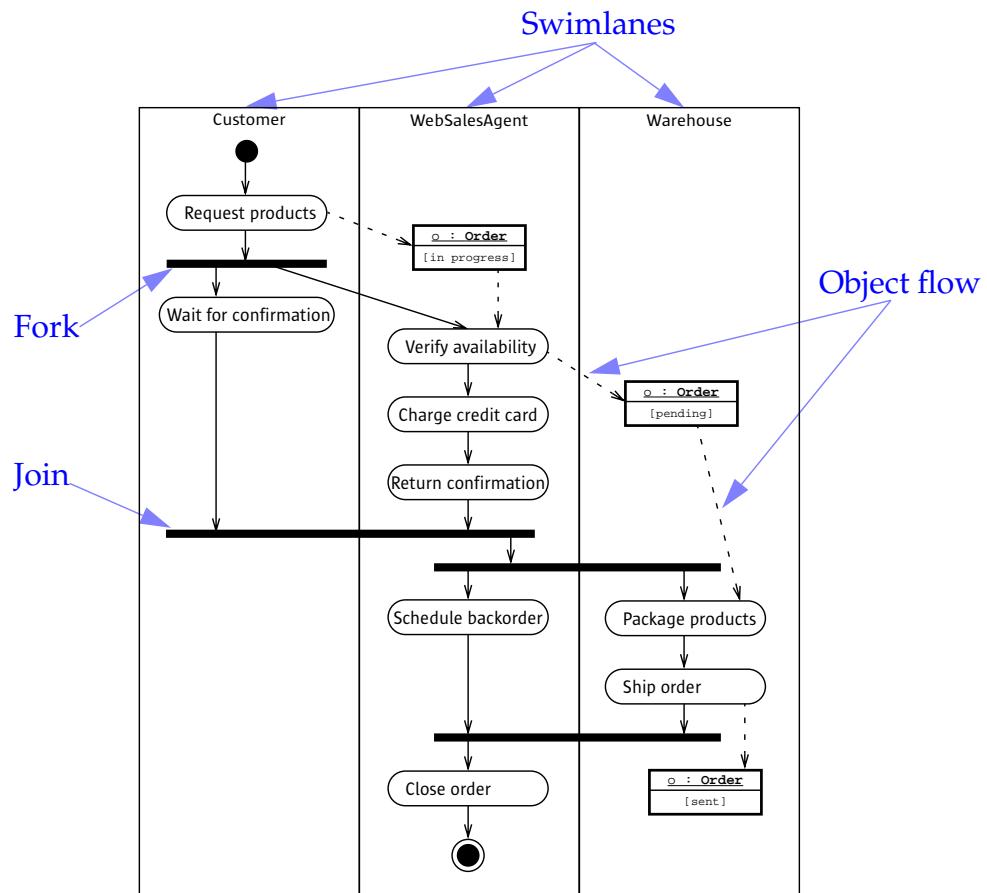
**Figure F-24** Activities and Transitions

Figure F-25 demonstrates branching and looping in Activity Diagrams. The diagram models the higher level activity of “Verify availability” of products in a purchase order. The top-level branch node forms a simple while loop. The guard on the transition below the branch is true if there are more products in the order to be processed. The “else” transition halts this activity.



**Figure F-25** Branching and Looping

Figure F-26 shows a richer Activity Diagram. In this example, swim lanes are used to isolate the actor of a given set of activities. These actors might include humans, systems, or organization entities. This diagram also demonstrates the ability to model concurrent activities. For example, the Customer initiates the purchase of one or more products on the company's Web site. The Customer then waits as the WebSalesAgent software begins to process the purchase order. The fork bar splits a single transition into two or more transitions. The corresponding join bar must contain the same number of inbound transitions.

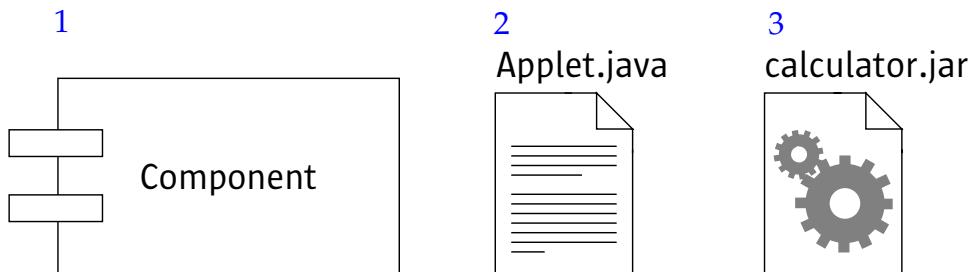


**Figure F-26** An Example Activity Diagram

# Component Diagrams

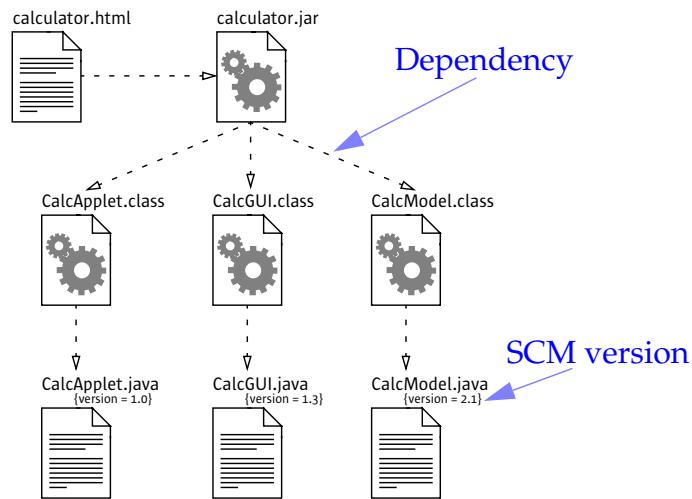
A Component Diagram represents the organization and dependencies among software implementation components.

Figure F-27 shows three types of icons that can represent software components. Example 1 is a generic icon. Example 2 is an icon that is used to represent a source file. Example 3 is an icon that represents a file that contains executable (or object) code.



**Figure F-27** Example Component Nodes

Figure F-28 shows the dependencies of packaging an HTML page that contains an applet. The HTML page depends on the JAR file, which is constructed from a set of class files, compiled from the corresponding source files. You can use a tagged value to indicate the source control version numbers on the source files.



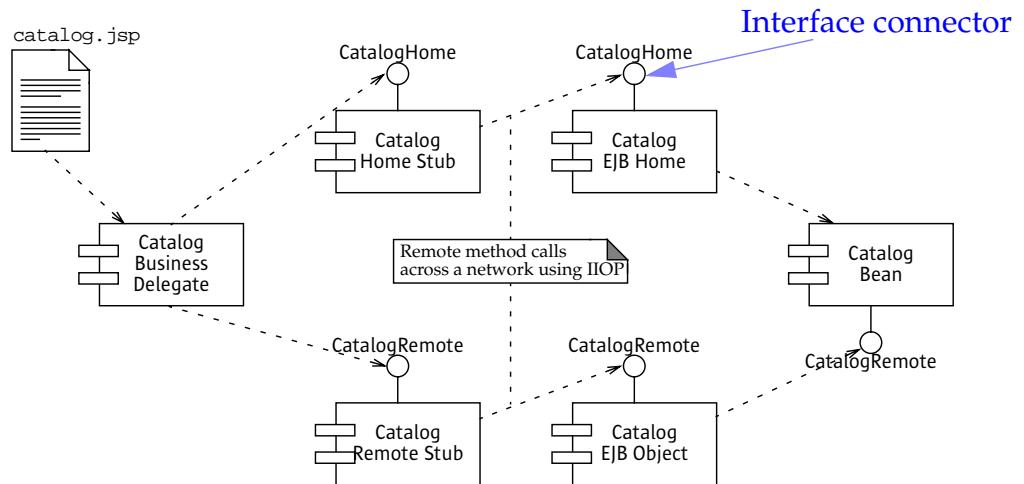
**Figure F-28** An Example Component Diagram

---

**Note** – SCM is Source Control Management.



Figure F-29 shows another Component Diagram. In this diagram, several components have an interface connector. The component attached to the connector implements the named interface. The component that has an arrow pointing to the connector depends on the fact that that component realizes that interface.



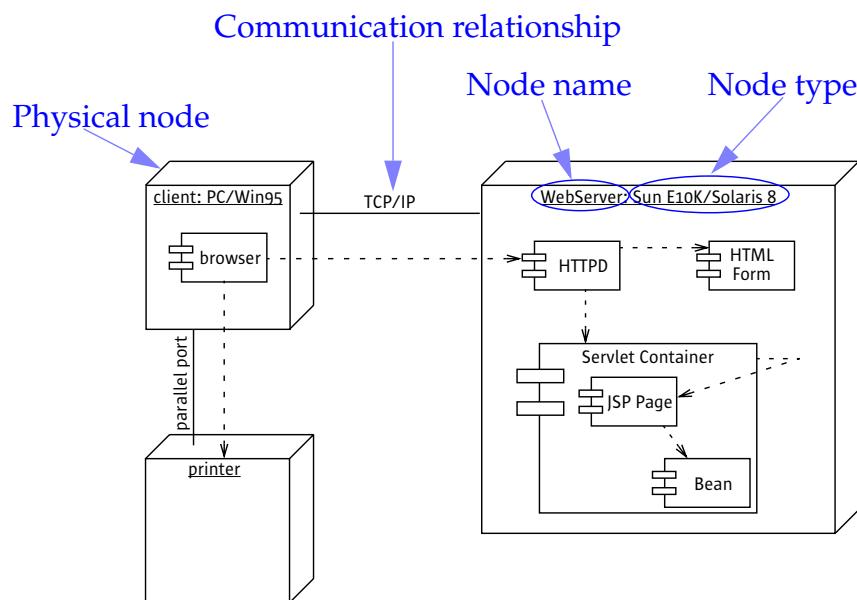
**Figure F-29** A Component Diagram With Interfaces

In this J2EE technology example, the Web tier includes a catalog JSP page which uses a catalog Business Delegate JavaBeans component to communicate to the EJB tier. Every enterprise bean must include two interfaces. The home interface allows the client to create new enterprise beans on the EJB server. The remote interface allows the client to call the business logic methods on the (remote) enterprise bean. The business delegate communicates with the catalog bean through local stub objects that implement the proper home and remote interfaces. These objects communicate over a network (using the Internet Inter-ORB protocol) with remote “skeletons” (in EJB technology terms, these are called EJBHome and EJBObject). These objects communicate directly with the catalog bean that implements the true business logic.

# Deployment Diagrams

A Deployment Diagram represents the network of processing resource elements and the configuration of software components on each physical element.

A Deployment Diagram is composed of hardware nodes, software components, software dependencies, and communication relationships. Figure F-30 shows an example with an end user with a Microsoft Windows compatible PC with a printer on the parallel port. The user is using a Web browser to communicate (over TCP/IP) with the Web server on a Sun Enterprise E10K. The Web server uses HTML pages and a Java Servlet engine for process a JSP page request.



**Figure F-30** An Example Deployment Diagram

This type of diagram can be used to show how the logical tiers of an application architecture are configured into a physical network.

