

D65271GC11  
Edition 1.1  
July 2010  
D68405

**ORACLE®**

# **Web Component Development With Servlet and JSP™ Technologies**

**Student Guide**

**SL-314-EE6**

**Copyright © 2010, Oracle and/or its affiliates. All rights reserved.**

**Disclaimer**

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

**Sun microsystems Disclaimer**

This training manual may include references to materials, offerings, or products that were previously offered by Sun microsystems. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

**Restricted Rights Notice**

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

**U.S. GOVERNMENT RIGHTS**

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

**Trademark Notice**

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

This page intentionally left blank.

This page intentionally left blank.

# Table of Contents

<b>About This Course .....</b>	<b>Preface-xiii</b>
Course Goals.....	Preface-xiii
Course Map.....	Preface-xiv
Topics Not Covered.....	Preface-xv
How Prepared Are You?.....	Preface-xvi
How to Learn From This Course .....	Preface-xvi
Introductions .....	Preface-xvii
How to Use Course Materials .....	Preface-xviii
Conventions .....	Preface-xix
Icons .....	Preface-xix
Typographical Conventions .....	Preface-xx
Additional Conventions.....	Preface-xxi
<b>Introduction to Java Servlets .....</b>	<b>1-1</b>
Objectives .....	1-1
Relevance.....	1-2
Additional Resources .....	1-3
Web Application Technologies .....	1-4
HTTP Client-Server Architecture .....	1-4
Web Site Structure .....	1-5
Web Sites and Web Applications.....	1-7
Execution of CGI Programs.....	1-7
Advantages and Disadvantages of CGI Programs .....	1-9
Using Java in the Web .....	1-10
Execution of Java Servlets.....	1-10
Advantages and Disadvantages of Java Servlets .....	1-11
Java Servlets.....	1-13
A First Java Servlet.....	1-14
HTTP Methods .....	1-16
Summary .....	1-17
<b>Introduction to Java Server Pages .....</b>	<b>2-1</b>
Objectives .....	2-1
Relevance.....	2-2

Additional Resources .....	2-3
A Weakness in Servlets .....	2-4
Addressing the Problem With JSPs .....	2-5
Key Elements of JSPs .....	2-6
How a JSP Is Processed .....	2-7
The request and response Variables .....	2-7
Reading Input Parameters From the Browser .....	2-8
Reading Parameters in JSP .....	2-8
Sending Parameters in an HTTP GET Request .....	2-8
Remaining Problems With the JSP Approach .....	2-9
Three Distinct Problems .....	2-9
Model, View, and Controller .....	2-10
Toward a Practical MVC .....	2-10
Summary .....	2-12
<b>Implementing an MVC Design .....</b>	<b>3-1</b>
Objectives .....	3-1
Relevance .....	3-2
Additional Resources .....	3-3
Developing the MVC Solution .....	3-4
Responsibilities of the Controller .....	3-4
Connecting the Components .....	3-4
Using Data in the JSP .....	3-6
A Complete MVC Example .....	3-7
A Design Question .....	3-8
Summary .....	3-10
<b>The Servlet's Environment .....</b>	<b>4-1</b>
Objectives .....	4-1
Relevance .....	4-2
Additional Resources .....	4-3
HTTP Revisited .....	4-4
Hypertext Transfer Protocol .....	4-4
HTTP GET Method .....	4-6
HTTP Request .....	4-6
HTTP Request Headers .....	4-7
HTTP Response .....	4-8
HTTP Response Headers .....	4-9
Collecting Data From the User .....	4-10
HTML Form Mechanism and Tag .....	4-10
Input Types for Use With Forms .....	4-11
Text Input Component .....	4-11
Drop-Down List Component .....	4-12
Submit Button .....	4-13
An Example HTML Form .....	4-14
How Form Data Are Sent in an HTTP Request .....	4-16
Form Data in the HTTP Request .....	4-16

HTTP GET Method Request.....	4-17
HTTP POST Method Request.....	4-18
HTTP GET and POST Methods .....	4-19
Web Container Architecture.....	4-20
Request and Response Process .....	4-21
Sequence Diagram of an HTTP GET Request.....	4-25
The HttpServlet API.....	4-26
Handling Errors in Servlet Processing.....	4-30
The HTTP Protocol and Client Sessions .....	4-31
The HttpSession API.....	4-32
Storing Session Attributes .....	4-32
Retrieving Session Attributes.....	4-33
Closing the Session .....	4-34
Architectural Consequences of Sessions.....	4-35
Session Configuration .....	4-35
Using Cookies for Client-Specific Storage.....	4-36
Cookie API .....	4-37
Using Cookies Example .....	4-38
Performing Session Management Using Cookies .....	4-38
Using URL-Rewriting for Session Management .....	4-40
URL-Rewriting Implications .....	4-41
Summary .....	4-42
<b>Container Facilities for Servlets and JSPs .....</b>	<b>5-1</b>
Objectives .....	5-1
Relevance.....	5-2
Additional Resources .....	5-3
Packaging Web Applications .....	5-4
The Default Context Root .....	5-4
Essential Structure of a WAR file .....	5-5
Deployment Descriptors .....	5-7
The web-fragment.xml Files .....	5-7
Controlling Configuration.....	5-8
Dynamic Configuration of Web Applications .....	5-9
Servlet Mapping.....	5-10
Multiple and Wildcard URL Patterns .....	5-11
Mapping Using Annotations.....	5-11
Invoking a Servlet by Name.....	5-12
Servlet Context Information .....	5-13
Context Parameters.....	5-13
Supplying Context Parameters .....	5-13
Reading Context Parameters.....	5-14
Servlet Initialization Parameters.....	5-15
Reading Servlet Initialization Parameters.....	5-16
Other Configuration Elements.....	5-17
Summary .....	5-18

<b>More View Facilities .....</b>	<b>6-1</b>
Objectives .....	6-1
Relevance.....	6-2
Additional Resources .....	6-3
Scopes .....	6-4
More Details About the Expression Language (EL) .....	6-6
Syntax Overview .....	6-6
EL and Scopes.....	6-6
EL Implicit Objects.....	6-7
The Dot Operator In EL.....	6-8
Array Access Syntax With EL .....	6-8
EL and Errors.....	6-9
EL Arithmetic Operators.....	6-10
Comparisons and Logical Operators .....	6-11
Configuring the JSP Environment.....	6-13
Presentation Programming.....	6-14
Standard Custom Tags.....	6-14
Tag Example .....	6-15
Developing JSP Pages Using Custom Tags.....	6-15
Key View Programming Tags.....	6-16
JSTL if Tag.....	6-16
JSTL forEach Tag.....	6-17
Summary .....	6-19
<b>Developing JSP Pages .....</b>	<b>7-1</b>
Objectives .....	7-1
Relevance.....	7-2
Additional Resources .....	7-3
JavaServer Pages Technology .....	7-4
How a JSP Page Is Processed.....	7-7
Writing JSP Scripting Elements.....	7-14
Comments .....	7-15
Directive Tag.....	7-16
Declaration Tag .....	7-16
Scriptlet Tag.....	7-17
Expression Tag .....	7-19
Implicit Variables.....	7-20
Using the page Directive .....	7-21
Including JSP Page Segments.....	7-23
Using the include Directive.....	7-23
Using Standard Tags .....	7-24
Other Standard Tags.....	7-30
Using the jsp:include Standard Action .....	7-31
Using the jsp:param Standard Action.....	7-32
t .....	7-34
Summary .....	7-35



<b>Developing JSP Pages Using Custom Tags .....</b>	<b>8-1</b>
Objectives .....	8-1
Relevance.....	8-2
Additional Resources .....	8-3
The JSTL .....	8-4
The Java EE Job Roles Involved in Web Application Development .....	8-4
Designing JSP Pages With Custom Tag Libraries .....	8-5
Custom Tag Library Overview .....	8-6
Custom Tag Syntax Rules .....	8-7
JSTL Sample Tags.....	8-9
Using a Custom Tag Library in JSP Pages.....	8-12
JSTL Tags.....	8-13
Summary .....	8-19
<b>More Controller Facilities .....</b>	<b>9-1</b>
Objectives .....	9-1
Relevance.....	9-2
Additional Resources .....	9-3
Servlet Life Cycle Overview .....	9-4
Servlet Class Loading .....	9-5
One Instance Per Servlet Definition .....	9-6
The <code>init</code> Life Cycle Method .....	9-7
The <code>ServletConfig</code> API .....	9-8
The <code>service</code> Life Cycle Method.....	9-9
The <code>destroy</code> Life Cycle Method.....	9-10
Servlet Lifecycle and Annotations.....	9-11
Lifecycle Method Annotations.....	9-12
Servlets and Threading .....	9-13
Handling Concurrency.....	9-13
Data Shared Between Invocations by a Single Client .....	9-13
Sharing Data Between Multiple Clients .....	9-14
Web Container Request Cycle.....	9-15
Web Container Request Processing.....	9-15
Applying Filters to an Incoming Request.....	9-16
Applying Filters to a Dispatched Request.....	9-18
Filter API .....	9-19
Developing a Filter Class .....	9-20
The <code>PerformanceFilter</code> Class.....	9-20
The <code>init</code> Method .....	9-21
The <code>doFilter</code> Method .....	9-21
The <code>destroy</code> Method .....	9-22
Configuring the Filter .....	9-23
Configuring a Filter Using Annotations.....	9-23
Declaring a Filter in the <code>web.xml</code> File .....	9-23
Declaring a Filter Mapping in the <code>web.xml</code> File.....	9-24

Handling Multipart Forms .....	9-26
Summary .....	9-29
<b>More Options for the Model .....</b>	<b>10-1</b>
Objectives .....	10-1
Relevance.....	10-2
Additional Resources .....	10-3
The Model as a Macro-Pattern .....	10-4
The View Helper Pattern .....	10-5
Database and Resource Access .....	10-6
Data Access Object (DAO) Pattern.....	10-6
DAO Pattern Advantages .....	10-8
JDBC API.....	10-9
Developing a Web Application Using a Database .....	10-10
Traditional Approaches to Database Connections.....	10-10
Using a DataSource and the Java Naming and Directory Interface API .....	10-10
Executing SQL .....	10-13
Configuring a DataSource and the JNDI API .....	10-14
Object Relational Mapping Software .....	10-15
Java Persistence API Structure.....	10-18
Entity Class Requirements.....	10-19
Declaring the Entity Class.....	10-19
Verifying and Overriding the Default Mapping.....	10-21
Life Cycle and Operational Characteristics of Entity Components..	10-23
Persistence Units .....	10-23
The persistence.xml file .....	10-23
The Persistence Context .....	10-24
The EntityManager .....	10-24
Entity Instance Management.....	10-24
User Transactions.....	10-26
Java Persistence API Example.....	10-27
Summary .....	10-28
<b>Asynchronous Servlets and Clients .....</b>	<b>11-1</b>
Objectives .....	11-1
Relevance.....	11-2
Additional Resources .....	11-3
Asynchronous Servlets.....	11-4
Separating Request Receipt from Response Generation...	11-4
Asynchronous Servlet Example.....	11-5
Forwarding and Filtering.....	11-8
Asynchronous Listeners.....	11-9
Asynchronous JavaScript Clients .....	11-10
Simple Asynchronous Client Example .....	11-10
Server Response Content in an AJAX System .....	11-12

Combining Asynchronous Servlets With Asynchronous JavaScript	11-13
Summary .....	11-14
<b>Implementing Security .....</b>	<b>12-1</b>
Objectives .....	12-1
Relevance.....	12-2
Additional Resources .....	12-3
Security Considerations .....	12-4
Confusion of Code and Data .....	12-5
SQL Injection Example .....	12-5
Other Code as Data Attacks .....	12-6
Preventing Code as Data Attacks .....	12-7
Authentication and Authorization .....	12-8
Authenticating the Caller.....	12-11
Caller Authentication .....	12-12
Establishing User Identities.....	12-13
Examining the Java EE Authorization Strategies .....	12-16
Using Declarative Authorization.....	12-18
Creating a Credentials Database.....	12-18
Declaring Security Roles .....	12-18
Mapping Users to Roles .....	12-19
Declaring Permission Requirements.....	12-20
Using Programmatic Authorization.....	12-22
Using the <code>isUserInRole</code> Method .....	12-22
Enforcing Encrypted Transport .....	12-24
Using an Annotation to Mandate Encrypted Transport .....	12-25
Summary .....	12-26
<b>Introducing JavaServer™ Faces Technology.....</b>	<b>A-1</b>
Objectives .....	A-1
Relevance.....	A-2
Additional Resources .....	A-3
JavaServer Faces Technology Overview.....	A-4
Introducing JavaServer Faces Technology .....	A-4
Components of a JavaServer Faces Technology-Based Web	
Application .....	A-6
Comparison of JavaServer Faces Technology With Struts	
Framework.....	A-10
Key JavaServer Faces Concepts .....	A-11
UI Component Model .....	A-12
UI Component Classes .....	A-12
DVD Library Sample Project: Use Cases .....	A-14
<b>Quick Reference for HTML .....</b>	<b>B-1</b>
Objectives .....	B-1
Additional Resources .....	B-2

HTML and Markup Languages .....	B-3
Definition .....	B-3
Types of Markup .....	B-3
Simple Example .....	B-4
Creating an HTML Document .....	B-5
Tag Syntax .....	B-5
Comments .....	B-6
Spaces, Tabs, and New Lines Within Text .....	B-6
Character and Entity References .....	B-6
Creating Links and Media Tags .....	B-8
The A Element and the HREF Attribute .....	B-8
The IMG Element and the SRC Attribute .....	B-9
The APPLET Element .....	B-9
The OBJECT Element .....	B-10
Text Structure and Highlighting .....	B-11
Text Structure Tags .....	B-11
Text Highlighting .....	B-14
HTML Forms .....	B-15
The FORM Tag .....	B-15
HTML Form Components .....	B-17
Input Tags .....	B-18
Text Fields .....	B-19
Submit Buttons .....	B-20
Reset Button .....	B-21
Checkboxes .....	B-22
Radio Buttons .....	B-23
Password .....	B-24
Hidden Fields .....	B-24
The SELECT Tag .....	B-25
The TEXTAREA Tag .....	B-26
Creating HTML Tables .....	B-27
Advanced HTML .....	B-30
JavaScript™ Programming Language .....	B-30
CSS .....	B-32
Frames .....	B-36
<b>Quick Reference for HTTP .....</b>	<b>C-1</b>
Objectives .....	C-1
Additional Resources .....	C-2
HTTP Overview .....	C-3
Definition .....	C-3
Request Structure .....	C-4
HTTP Methods .....	C-5
Request Headers .....	C-6
Response Structure .....	C-8
Response Headers .....	C-9

---

Status Codes.....	C-10
CGI Overview.....	C-12
Environment Variables Set .....	C-12
Data Formatting .....	C-14
<b>Quick Reference for XML.....</b>	<b>D-1</b>
Objectives .....	D-1
Additional Resources .....	D-2
Introduction to XML.....	D-3
Simple Example.....	D-3
Basic Syntax .....	D-4
Well-Formed XML Documents.....	D-4
Validity and DTDs.....	D-5
DTD-specific Information.....	D-7
Schemas .....	D-9



## Preface

---

## About This Course

---

### Course Goals

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

- Write servlets using the Java™ programming language (Java servlets)
- Create robust web applications using Struts, session management, filters, and database integration
- Write pages created with JavaServer Pages™ technology (JSP™ pages)
- Create easy to maintain JSP pages using the Expression Language, JSP Standard Tag Library (JSTL), and the Struts Tiles framework
- Create robust web applications that integrate Struts and JSP pages

This course describes how to create dynamic web content using Java technology servlets and JSP technology. The course describes how to construct small to medium scale web applications and deploy them onto the GlassFish™ Application Server (Application Server), which is the reference implementation for the servlet and JSP specifications.

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

# Course Map

## Web Component Development With Servlet and JSP™ Technologies

Introduction to Java Servlets	Introduction to Java Server Pages	Implementing an MVC Design
The Servlet's Environment	Container Facilities For Servlets and JSPs	More View Facilities
Developing JSP Pages	Developing JSP Pages Using Custom Tags	More Controller Facilities
More Options For The Model	Asynchronous Servlets and Clients	Implementing Security



## Topics Not Covered

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

- Java technology programming – Covered in SL-275: *The Java™ Programming Language*
- Object-oriented design and analysis – Covered in OO-226: *Object-Oriented Analysis and Design Using UML*
- Java Platform, Enterprise Edition – Covered in WJT-310: *Java™ 2 Platform, Enterprise Edition: Technology Overview*
- Enterprise JavaBeans™ technology – Covered in SL-351: *Enterprise JavaBeans™ Programming*
- JavaServer™ Faces technology – Covered in DTJ-3108: *Developing JavaServer™ Faces Components With AJAX*

Refer to the Sun Learning 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 such as Unified Modeling Language (UML)?
- Can you create a simple web page using Hypertext Markup Language (HTML)?

## How to Learn From This Course

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

- Ask questions
- Participate in the discussions and exercises
- Read the code examples
- Use the online documentation for Java Platform, Standard Edition (Java SE™ platform), 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.



**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+rw 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 System (Solaris OS) is different from a command used in the Microsoft Windows platform, both commands are shown; for example:

If working in the Solaris OS:

```
$CD SERVER_ROOT/BIN
```

If working in Microsoft Windows:

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





## Module 1

---

# Introduction to Java Servlets

---

## Objectives

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

- Outline Java™ servlet technology

## 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?
- Did your web technology allow you to achieve your goals?

## 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 Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans™ IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>

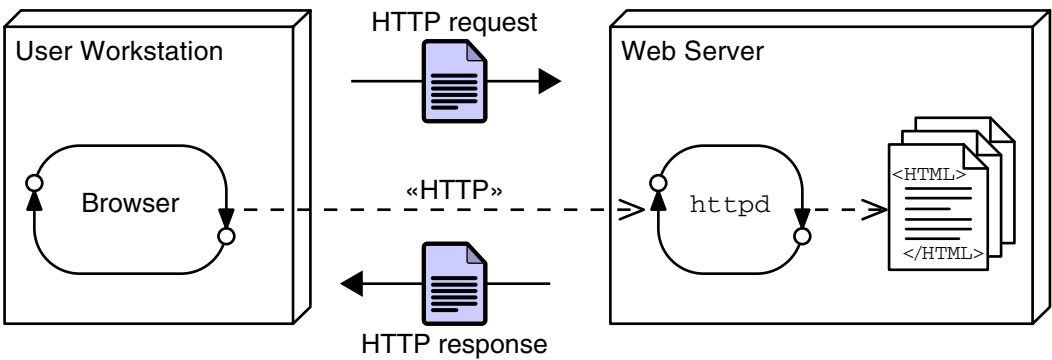
# Web Application Technologies

The Hypertext Transfer Protocol (HTTP) was created in conjunction with the related Hypertext Markup Language (HTML) standard. HTTP is used to transfer instructions and data between machines, while HTML is a document display language that lets users create visually pleasing documents and link from one document to another. For example, a paper about quarks stored at the European Organization for Nuclear Research (CERN) in Switzerland could include a “See also” link to a paper on strangeness and charm stored on a computer in Fargo, North Dakota.

HTML 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. Therefore, HTTP can be used to transmit any file. The data type of the file is described using the Multipurpose Internet Mail Extensions (MIME) specification. The combination of the HTTP protocol and the HTML page description language are the foundation technologies of the World Wide Web.

## HTTP Client-Server Architecture

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



**Figure 1-1** 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.

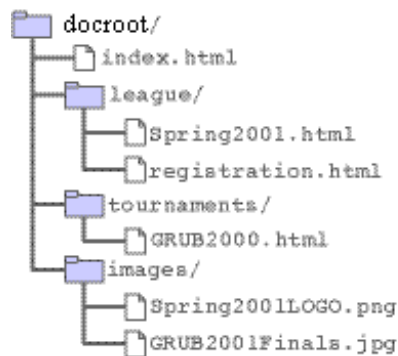
The request information includes the location of the requested file or resource, and information about the browser and its environment. The response contains the requested resource and other information. The request is typically in plain text. The response can be plain text or part plain text, part binary data. (Graphics, for example, must be sent in binary form.)

## Web Site Structure

A web site is a collection of HTML pages and other media files that contains 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. Internally, the web site is composed of that directory hierarchy. This structure might be visible to the end user in the Uniform Resource Locators (URLs) that identify each page.

A second structure is defined within the documents themselves. This structure defines the links between documents and is not subject to a hierarchical structure, but can be any kind of network of links, including links to documents on other sites. This structure is visible to the end user in the links that support navigation between pages.

Figure 1-2 shows an example of a web site.



**Figure 1-2** Web Site Directory Structure Example



**Note** – The `index.html` file is a special file used when the user requests a 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 locates a specific resource on the Internet. It consists of several parts, and is similar in some ways to a directory and file specification:

*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 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 Sites and Web Applications

A web site is a collection of *static* files, HTML pages, graphics, and various other files. A web application is an element of a web site that provides dynamic functionality on the server. A web application runs a program or programs on the server. An example of an interaction with a web application might be:

1. A browser makes a request to the server for an HTML form.
2. The server responds by sending the HTML form back to the browser in an HTTP request stream.
3. The browser sends another request, along with data provided by the user in the form, to the server.
4. The server passes the request and data to program code that performs some computation using those data and responds by sending another HTML page back to the browser to present results from the computation.



---

**Note** – Do not confuse web applications with web services. Web services are services that are offered by one application to another over the World Wide Web. They typically involve the transfer of data in Extensible Markup Language (XML) format. In effect, the client of a Web service is another computer system, rather than a user.

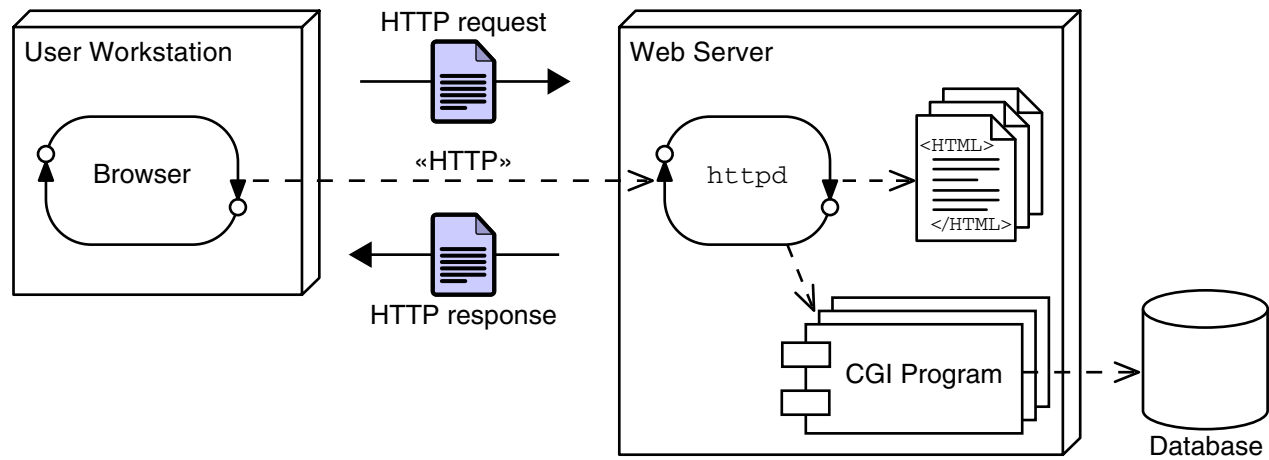
---

## Execution of CGI Programs

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

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. Figure 1-3 shows an example web application architecture that uses CGI programs.



**Figure 1-3** Web Server Architecture With CGI Programs

At runtime, a CGI program is launched by the web server as a separate process. The process executes the code of the CGI program, which resides within the server’s file system. Figure 1-4 on page 1-11 shows the runtime performance of one CGI request.

However, each new CGI request launches a new process on the server and this results in slow response, high resource usage, and poor scalability in the server.



## Advantages and Disadvantages of CGI Programs

CGI programs have the following advantages:

- Programs can be written in a variety of languages, although they are primarily written in Perl.
- A CGI program with bugs does not crash the web server.
- Because CGI programs execute in separate processes, concurrency issues are isolated at the database.
- CGI support is very common.

CGI program also have the following distinct disadvantages:

- The response time of CGI programs is high because the creation of a new process is a heavyweight activity for the operating system (OS).
- CGI does not scale well.
- 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.

## Using Java in the Web

The history of Java and the World Wide Web are closely entwined. However, when Java was first released, it did not directly support the creation of Web applications. At that time, many web pages were still largely static, and those that were dynamic were commonly built around CGI.

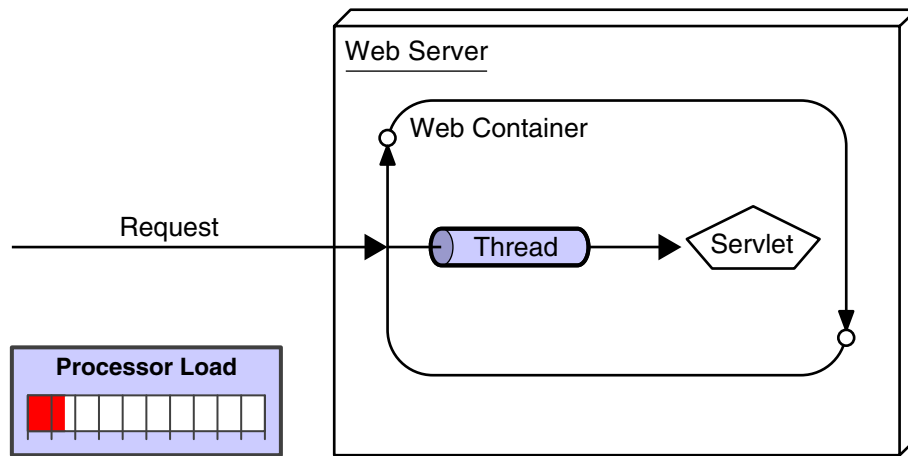
Soon, Java gained the ability to service HTTP requests and take its place in the development of dynamic web sites.

The earliest technology developed for this was the servlet. Servlets provide a simple framework that allows Java program code to process an HTTP request and create an HTML page in response.

## Execution of Java Servlets

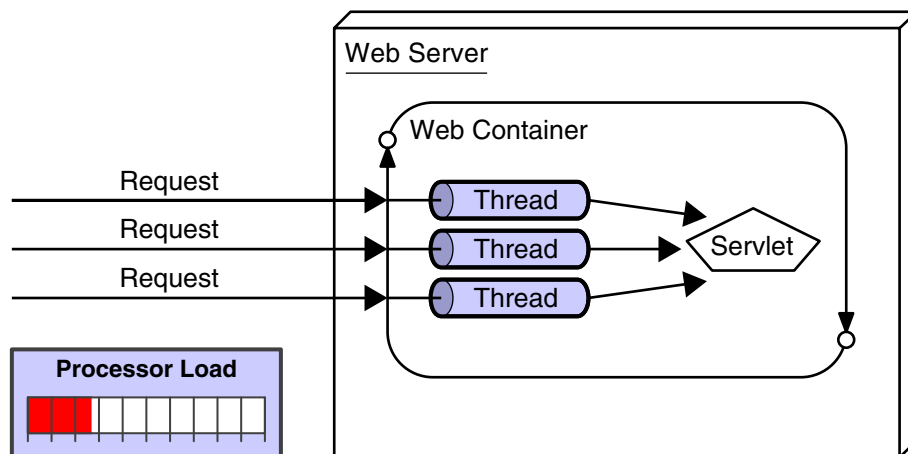
A Java servlet is similar to a CGI program in that it responds to HTTP requests and runs on the server. The types of tasks that you can run with servlets are the same as those that you can run with CGI. However, the underlying executing architecture is different.

The basic processing steps for Java servlets are quite similar to the steps 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. This is opposed to a CGI script in which a new process is created for each request. Figure 1-4 shows that a servlet executes as a thread within the web container's process.



**Figure 1-4** Running a Single Instance of a Servlet

When the number of requests for a servlet rises, no additional instances of the servlet or OS processes are created. Each request is processed concurrently using one Java thread per request. Figure 1-5 shows the effect of additional clients requesting the same servlet.



**Figure 1-5** 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 within a single process, so servlet request processing is significantly faster than traditional CGI processing.

## Using Java in the Web

---

- Servlets are much more scalable than CGI. Many more requests can be executed because the web container uses a thread rather than an OS process. Threads are lightweight and the host OS can support many more of them.
- Servlets benefit from the simple, robust, platform independent, and object-oriented nature of the Java programming language.
- Servlets have access to standardized and easy-to-use logging capabilities.
- The web container provides additional services to the servlets, such as error handling and security.

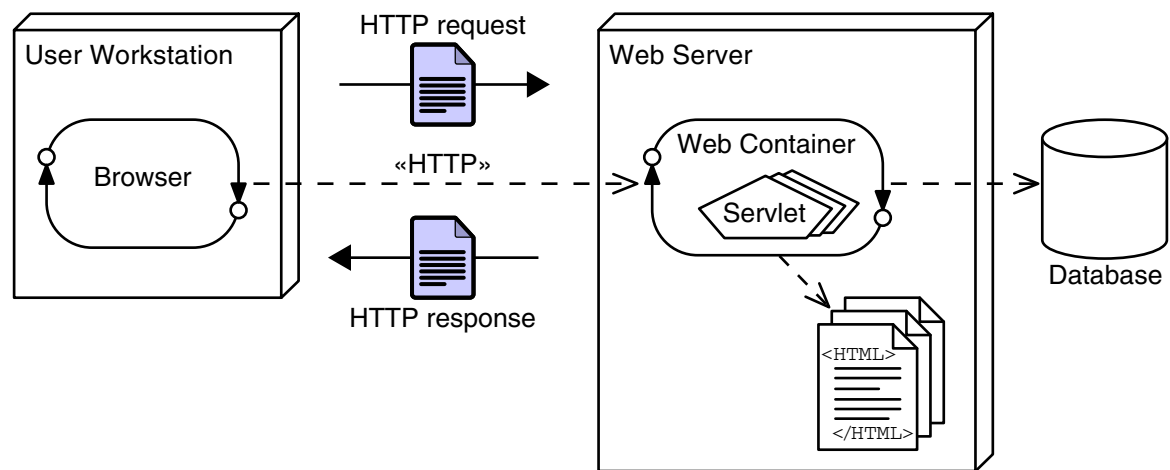
Servlets have the following disadvantages:

- Servlets can only be written in the Java programming language, so developers are required to be competent with this language.
- Servlets might introduce new concurrency issues not found in CGI.

# Java Servlets

Servlets run within the Java Platform Enterprise Edition (Java EE) component container architecture. This container is called the web container (also known as the servlet engine). The web container is a Java™ technology program that implements the servlet application programming interface (API). Servlets are components that respond to HTTP requests. The web container performs initial processing, and selects the intended servlet to handle the request. The container also controls the life-cycle of the servlet.

Figure 1-6 shows this architecture.



**Figure 1-6** Sample 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.

# A First Java Servlet

A servlet is invoked by the web container when an appropriate request is received by that container. In Java EE 6, an annotation specifies the URL (or URL pattern) that the servlet is used to respond to.

The method in the servlet that is invoked depends on the type of HTTP request so, for example, an HTTP GET request will invoke the `doGet` method in the servlet.

The `doGet` method takes two parameters, one which carries information related to the original request, and another which provides for control of the response.

The servlet's job is two-fold. First, it must perform the required computation; second, it must present the results in a well-formed HTML page.

The code for a very simple servlet is shown in Code 1-1 below.

```
1 package sl351.m1;
2
3 import java.io.IOException;
4 import java.io.PrintWriter;
5 import javax.servlet.ServletException;
6 import javax.servlet.annotation.WebServlet;
7 import javax.servlet.http.HttpServlet;
8 import javax.servlet.http.HttpServletRequest;
9 import javax.servlet.http.HttpServletResponse;
10
11
12 @WebServlet(name="HelloServlet", urlPatterns={"/HelloServlet"})
13 public class HelloServlet extends HttpServlet {
14
15     @Override
16     protected void doGet(HttpServletRequest request,
17                          HttpServletResponse response)
18         throws ServletException, IOException {
19         response.setContentType("text/html;charset=UTF-8");
20         PrintWriter out = response.getWriter();
21         try {
22             out.println("<html>");
23             out.println("<head>");
24             out.println("<title>HelloServlet</title>");
25             out.println("</head>");
26             out.println("<body>");
27             out.println("<h1>HelloServlet says \"Hello, World!\"</h1>");
```

```
28         out.println("</body>");
29         out.println("</html>");
30     } finally {
31         out.close();
32     }
33 }
34 }
```

### Code 1-1 A Simple Servlet

Notice the following features of the code:

- The class extends `javax.servlet.http.HttpServlet` (lines 13 and 7).
- The class overrides the method `doGet`, which provides the home for the code that will service the HTTP request (lines 15-18).
- The `doGet` method creates an entire HTML page and sends it to the browser (lines 22-29).
- The servlet is associated with a URL within the web server by means of the `@WebServer` annotation (line 12).

In the simplest form, that's all that is necessary to create a dynamic web page using a servlet. Of course, in reality there are a great many additional things to learn beyond this, and those are the subject of following modules.

## HTTP Methods

In the example you saw a servlet that provided a method called `doGet` and handled HTTP requests in the body of that method.

You might be aware that HTTP provides several ways to send requests, and the GET method is just one of these. Another very common HTTP request is the POST method. As you might guess, the servlet provides a `doPost` method to handle such a request. The arguments of the `doPost` method are identical to those of `doGet`.

In many cases, a servlet should behave identically regardless of whether it was invoked using a GET or a POST method, so it's quite common to see both these methods delegate to another method of the programmer's own invention. You will see that NetBeans creates template servlets that already have this delegation code written, and the method used to contain the body of the servlet's behavior is called `processRequest`. You should know that the `processRequest` method is not part of the servlet specification.



## Summary

This module introduced web application basics, HTML, HTTP, and CGI. You then discovered the first Java technology for the web tier: servlets. To summarize:

- CGI provided hooks for web servers to execute application programs.
- Java servlets are similar to CGI, but they execute in a Virtual Machine for the Java platform (Java Virtual Machine, or JVM™ machine) using threading.
- Servlets extend `javax.servlet.http.HttpServlet`
- Servlets provide HTML responses to the browser.
- The `doGet` method of a servlet can be used to implement the required behavior of the servlet.
- The `@WebServlet` annotation can be used to tie a servlet to a URL.



## Module 2

---

# Introduction to Java Server Pages

---

## Objectives

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

- Describe a significant weakness in servlets when used alone
- Write a simple Java Server Page (JSP)
- Describe the translation of a JSP into a servlet
- Understand the basic goals of MVC

## Relevance



**Discussion** – The following questions are relevant to understanding how to develop a view component:

- How easy would it be to edit HTML that is created by a Java program?
- What would make a good separation of concerns for a web application?

## Additional Resources



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

- “The Java EE 6 Tutorial: Java Servlet Technology,”  
[<http://java.sun.com/javaee/6/docs/tutorial/doc/Servlets.html>], accessed 02 December 2006.

This course is based on the latest servlet specification version 2.5.

- Basham, Bryan, Kathy Sierra, and Bert Bates. *Head First Servlets and JSP: Passing the Sun Certified Web Component Developer Exam (SCWCD)*. O'Reilly Media, Inc., 2004.

An introduction to JSP pages and servlets, as well as material on passing the Sun Certified Web Component Developer exam.

## A Weakness in Servlets

The servlet system described in the previous module provides a simple and effective way to use Java to respond to an HTTP request. When combined with the powerful and varied library APIs available to a Java program, almost anything can be achieved. However, there is a problem with the approach as presented in a real-world system.

Most modern web applications have elegant and complex user interface designs. The designs are often created by dedicated web designers, and even if that is not the case, the HTML pages that implement the designs are frequently created using “What You See Is What You Get” (WYSIWYG) Graphical User Interface (GUI) based design tools. Such design tools work on HTML, not on HTML embedded in Java program code.

Because of this, the pure servlet approach becomes unmanageable—often catastrophically—when maintenance is required on the web pages. Having to re-type all the HTML code back into `out.print(...)` statements would be a very time consuming and error prone process.

## Addressing the Problem With JSPs

Soon after the original introduction of servlets, Java Server Pages, usually called simply JSPs, were introduced to address this problem.

While a servlet is a Java source file containing embedded HTML, a JSP might be considered to be an HTML file with Java embedded in it. Code 2-1 below shows an example of a very simple JSP.

```
1  <%--  
2      Simple Hello World JSP example  
3  --%>  
4  
5  <%@page contentType="text/html" pageEncoding="UTF-8"%>  
6  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
7      "http://www.w3.org/TR/html4/loose.dtd">  
8  
9  <html>  
10     <head>  
11         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
12         <title>JSP Page</title>  
13     </head>  
14     <body>  
15         <h1><%= "Hello World! I'm a JSP, it is " + new java.util.Date() %></h1>  
16     </body>  
17 </html>
```

**Code 2-1** A Simple JSP

## Key Elements of JSPs

Notice these aspects of the JSP shown in Code 2-1 on page 2-5:

- The bulk of the document is a regular HTML file; indeed, it is entirely legal HTML, and can be edited in a WYSIWYG editor.
- Additional features related to the Java aspects of the JSP are denoted using the `<% ... %>` pairs.
- Within `<% ... %>` pairs, three distinct variations are exemplified. These are:
  - Comments (line 1 through 3)
  - A “page directive” (line 5)
  - An “expression” (line 15)

In this example, the purpose of the page directive is to provide information about the content type of the JSP. Such directives can be used to provide several other types of “page-wide” configuration information.

The expression is computed using Java and the result is converted to a string. That string is then embedded into the HTML page to become part of the output that is sent to the browser.



## How a JSP Is Processed

Before a JSP is executed for the first time, it is converted to an equivalent servlet. This is similar to the compilation phase that converts a clean, maintainable source language such as Java into a potentially messy, hard-to-maintain, language such as the Java bytecode or a hardware-specific machine code. In this case, however, the conversion is performed automatically by the web container. Given that the essential problem outlined with servlets was difficulty of maintenance, the use of a servlet as a “machine language” is not a problem.

### The request and response Variables

The fact that the JSP is converted to a servlet suggests that the JSP might be able to access the same programming elements as were available in the original servlet. This is in fact the case. So, a JSP can refer to the variables request and response (along with others) that are provided as part of the standard for a JSP.

These variables, and others that will be introduced in later modules, provide many useful facilities that help in writing an effective JSP.

## Reading Input Parameters From the Browser

When a browser submits a form, the request commonly includes parameters, that is, data provided by the user. These parameters are available to a servlet through the `getParameter` method of the request variable. As you have just seen, this means that a JSP also has access to the same data. Parameters from a form are supplied in name/value pairs, and the `getParameter` method takes that name as a lookup key, and returns the associated value.

### Reading Parameters in JSP

Code 2-2 below illustrates how a JSP can access a parameter called `customerName` from an HTTP request. Note the use of the `getParameter` method on line 9.

```
1  <html>
2      <head>
3          <meta http-equiv="Content-Type"
4              content="text/html; charset=UTF-8">
5          <title>JSP Page</title>
6      </head>
7      <body>
8          <h1>Hello
9              <%= request.getParameter("customerName") %>
10         </h1>
11     </body>
12 </html>
```

**Code 2-2** Reading Input Data From the Browser in a JSP

### Sending Parameters in an HTTP GET Request

The HTTP standard makes it easy to create a GET request that includes a parameter, even without using a form. For example, the following URL would drive the JSP shown above in Code 2-2 and send it a value of “Corporal Jones” for the `customerName` parameter.

`http://localhost:8080/SL314m2ex2/index.jsp?customerName=Corporal+Jones`

Note that the space is encoded as a plus sign (+). Other encodings are required for many other symbols, but such conversions are beyond the scope of this module.

## Remaining Problems With the JSP Approach

Using JSPs provides a significant easing of the maintenance problem outlined in the previous module. The HTML that describes the visual aspect of the page can now be edited in a visual tool, and maintenance does not involve a massive cut, paste, and edit cycle.

However, the problem is still not solved. In effect, the servlet approach results in HTML embedded in a Java source file. The JSP approach results in Java code embedded in an HTML source file. Because of this, programmers and web-designers must share the same file, and changes made by one group might cause bugs in the work of the other group.

In fact, the root of the problem is that one of the basic tenets of good software design has been broken. That tenet states that unrelated concerns should be separated in the source code (object oriented designers talk about “separation of concerns”). In this case, there are in fact three distinct concerns being addressed.

### Three Distinct Problems

The first concern is the handling of the HTTP related elements of the problem. The request arrives as an HTTP request, with arguments and configuration data embedded in the HTTP headers. There is also a closely related issue that the response must be sent using the HTTP protocol.

The second concern is that there is a computational problem to be solved. That is, the essence of the request must be answered. Notice that this is separate from the issue of using HTTP to carry the request and response; the same request might be made in different circumstances by a Swing-based GUI program.

The third concern is that the browser requires the response to be formatted using HTTP. It would be possible, for example, to take the results of the computation and present them on paper.

## Model, View, and Controller

These three concerns have a long history, starting with Smalltalk programmers who identified essentially these same three concerns in general GUI-based programs.

Based on that historical origin, the code components that handle these three aspects are now called the model, view, and controller. Approaches using these elements are often referred to as MVC or Model-View-Controller designs.

The *model* is a computational model of the essential problem to be solved. It does not consider how the input is provided, nor the presentation of the output.

The *view* is responsible for taking the results of the computation from the model and presenting it to the user. Note that in some systems, the “user” might not be human, but might be another computer system. This does not alter the essential design.

The *controller* is responsible for taking the input, and pre-processing it, so that it can be presented to the model in a generic form that is appropriate to that model. In the web-based system, the controller extracts the essence of the request and argument data from the HTTP request.

## Toward a Practical MVC

We have not yet introduced enough of the JSP and Servlet technologies to know how to implement an MVC design, but this will be the subject of the next module. First, consider what tools are appropriate for implementing each of the elements of MVC.

The model is responsible for general computation. Depending on the nature of the problem to be solved, the model might need the help of a database or other systems. Given that the job of the model is essentially a pure computing problem, plain Java code would be an excellent approach for implementing the model.

The view is responsible for presenting the result data from the model as HTML. It is really only concerned with appearance and layout. HTML is an excellent tool—or at least is the tool supported on the web—so we might like to use plain HTML for the view.

The controller is responsible for receiving and interpreting the HTTP request. It must extract the essence of the request and the arguments that support it. This is a computational job, and so Java code is appropriate. However, the job requires an intimate relationship with HTTP and the nature of the HTTP request. That's exactly what a servlet does.

So, ideally, it would be good to create a controller using a servlet, pass the request to a plain Java object (usually referred to as a "Plain Old Java Object" or POJO) and have the results embedded in an HTML page.

Unfortunately, plain HTML does not provide a mechanism for extracting data from a Java object and embedding that data into the output. However, a JSP can—you can imagine how the expression syntax introduced above would allow you to extract a value from a Java object and embed that value into the HTML. Of course, this requires that the person working on the HTML file understand enough Java syntax to do this correctly, and that the same person not accidentally call methods that have unwanted side effects.

In fact, the history of the JSP shows four distinct techniques to embed values from Java objects into an HTML page. Each technique provided benefits over its predecessor. Often those benefits were aimed at minimizing the amount of new syntax that the HTML author needed to learn.

In the next module, you will learn how to make these three elements work together to form a practical MVC system.

## Summary

This module reviews the following information:

- A weakness in servlets when used alone
- How to write a simple Java Server Page (JSP)
- The translation of a JSP into a servlet
- Basic goals of MVC

## Module 3

---

# Implementing an MVC Design

---

## Objectives

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

- Implement the controller design element using a servlet
- Implement the model design element using a POJO
- Implement the view design element using a JSP and the Expression Language (EL)
- Connect the model, view, and controller elements to implement a working MVC solution

## Relevance



**Discussion** – The following questions are relevant to understanding how to develop a controller component:

- How can you implement the controller?
- How can you implement the model?
- How can you implement the view?



## Additional Resources



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

- “The Java EE 5 Tutorial: Java Servlet Technology,”  
[<http://java.sun.com/javaee/5/docs/tutorial/doc/Servlets.html>], accessed 02 December 2006.

This course is based on the latest servlet specification version 2.5.

- HTML 4.01 Specification. [Online]. Available:  
[<http://www.w3.org/TR/html4/>], accessed 13 November 2006.

## Developing the MVC Solution

You saw in the previous module that a design based around three components, a model, view, and controller, would make a good separation of concerns. In this module, you will see how to implement that design.

You will also recall from the previous module that the model will be implemented using regular Java objects, the controller will be implemented as a servlet, and the view will be implemented using a JSP.

### Responsibilities of the Controller

The controller is the point of contact that the web container knows about. Specifically, the web-container will invoke the controller in response to an HTTP request. The controller must then select a model to process the request, and a view to present the results. It is possible—even probable—that the model and view components might be specific to the particular request, or to details of the request, that has been received.

### Connecting the Components

For the MVC system to work, communication between the components is necessary. Specifically, the controller must pass request parameter data to the model to support the computation, and the controller must pass results from the model to the view.

Passing the request parameter data from the controller to the model is simple. Both of these components are regular Java code, and the controller invokes the model using a simple method call. However, the view is implemented using a JSP, and some other connection mechanism must be provided.

## Forwarding From a Servlet to a JSP

The mechanism used to invoke the JSP page is called *forwarding*. Code to perform this is shown below.

```
1  RequestDispatcher rd = request.getRequestDispatcher("view.jsp");  
2  rd.forward(request, response);
```

Notice that the forwarding is done in two parts. First, a `RequestDispatcher` object is created by the request. The `RequestDispatcher` is configured on creation to transfer control to a page called “view.jsp”. Next, the `RequestDispatcher` is used to pass the original request and response objects into the new page for processing.



---

**Note** – `RequestDispatchers` can also be obtained from the `ServletContext`.

---

## Passing Data From the Servlet to the JSP

While the technique just illustrated transfers control from the servlet to the JSP, it does not—of itself—answer the need for transferring data between the two. This need is handled by means of the request object too. In the previous module, you saw that the request object contains a map-like feature that gives access to the parameters of the request. This map is directly accessible in both the servlet and the JSP, but as you would probably expect, the parameters of the request are essentially read-only.

In fact, there is another map-like feature of the request that is specifically intended to carry information from one page (usually a servlet) so that it will be available to another page (usually a JSP) after a forward operation. This is called the attribute map, and may be read and written using the methods:

```
ServletRequest.getAttribute(String key)  
ServletRequest.setAttribute(String key, Object value)
```

Using this map, the controller servlet can send data to the view JSP.

## Using Data in the JSP

In the JSP, data in the request attribute map are accessible in a number of ways. Perhaps the cleanest and simplest way to access data om the model is using the expression language, usually referred to simply as EL. The EL provides access to the attribute map through a simple syntax that is very similar to that used for UNIX shell environment variables.

So, if a controller servlet writes a data item into an attributed using the following code:

```
request.setAttribute("aValue", 99);
```

Then the value can be read in a JSP page using this syntax:

```
${aValue}
```



**Note** – In older versions of the JSP specification, EL processing was disabled by default. In servers compliant with these older specifications, it is necessary to deliberately enable EL processing by including the following directive near the top of the JSP page:

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

EL is a very powerful and flexible mechanism that allows access to data items stored in the request attributes, and also to several other places not yet introduced. EL also allows access to complex structured data stored in Java Beans using a dotted notation similar to the dot operator in Java and other programming languages. So, for example, if a Java Bean is stored in the request scope and is called “aBean” and has an attribute called “myAttribute” then the syntax `${aBean.myAttribute}` would access that data item. The details of these facilities will be described in detail in a later module.

## A Complete MVC Example

The sample code just introduced allows passing of data from the controller to the view in a very simple way. However, for a real MVC example, the data for the view come from the model. The model is invoked by the controller.

One possible way to handle this is to have the model follow JavaBeans™ conventions, and actually present a reading interface that allows extraction of data directly from it. In this case, the controller can pass the entire model (or at least a facade component of the model) to the view.

To support this approach, the EL allows a dotted naming convention that will be entirely familiar to object oriented programmers. This is illustrated in the following complete example:

```
1  @WebServlet(name="FullControllerExample",
urlPatterns={"/FullControllerExample"})
2  public class FullControllerExample extends HttpServlet {
3
4      @Override
5      protected void doGet(HttpServletRequest request,
HttpServletRequest response)
6          throws ServletException, IOException {
7          RequestDispatcher rd =
            request.getRequestDispatcher("fullView.jsp");
8          ExampleModel model = new ExampleModel();
9          String name = request.getParameter("name");
10         if (name == null) {
11             name = "Secret";
12         }
13         model.setName(name);
14         request.setAttribute("model", model);
15         rd.forward(request, response);
16     }
17 }
```

### Code 3-1 The Controller Servlet

```
1  package sl314.m3;
2
3  public class ExampleModel {
4
5      private String name;
6
7      public void setName(String s) {
```

## A Complete MVC Example

```

8         name = s;
9     }
10
11     public String getName() {
12         return name;
13     }
14
15     public int getNameLength() {
16         return name.length();
17     }
18 }

```

### Code 3-2 The Model

```

1  <%@page isELIgnored="false"%>
2  <%@page contentType="text/html" pageEncoding="UTF-8"%>
3  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4      "http://www.w3.org/TR/html4/loose.dtd">
5
6  <html>
7      <head>
8          <meta http-equiv="Content-Type"
9              content="text/html; charset=UTF-8">
10         <title>A View Built With A JSP</title>
11     </head>
12     <body>
13         <h1>Name information</h1>
14         The name of the user is ${model.name}.
15         That name contains ${model.nameLength} characters.
16     </body>
17 </html>

```

### Code 3-3 The View JSP

## A Design Question

Notice that in the example above, the controller inserts the name parameter value into the model, and the view then reads the value from the model. You might consider simply extracting the name from the request directly into the view.

As you would expect, EL is capable of accessing parameters in this way, however this approach is probably poor design.

Consider that good design separates unrelated things, and tends to separate things that change independently. Suppose the form designer chose to change the name of a given parameter; clearly it's unavoidable that something in the processing will change. However, if the controller handles the parameter, and passes it to the model as a method argument, then the model need not be modified. Similarly, if the view accesses the parameter directly from the request object, the view too would have to change.

However, in the example above, the model echoes the parameter that was passed to it and the view only accesses elements of the model. With this approach, then if the form parameter name is changed, no change is needed to either the model or the view. In this way, the design exhibits looser coupling, and is more robust in the face of change or reuse.

Despite that admonition, later modules will address how to access parameters, and many other aspects of the original request, using EL.

## Summary

This module reviews the following information:

- How to implement the controller design element using a servlet
- How to implement the model design element using a POJO
- How to implement the view design element using a JSP and EL
- Connecting the model, view, and controller elements to implement a working MVC solution



## Module 4

---

# The Servlet's Environment

---

## Objectives

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

- Describe the environment in which the servlet runs
- Describe HTTP headers and their function
- Use HTML forms to collect data from users and send it to a servlet
- Understand how the web container transfers a request to the servlet
- Understand and use HttpSession objects

## Relevance

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

- What environment does the servlet run in?
- What are HTTP headers, and what are they for?
- How can you get data from a browser to a server?
- How can a server recognize a browser that is making successive calls?

## 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:

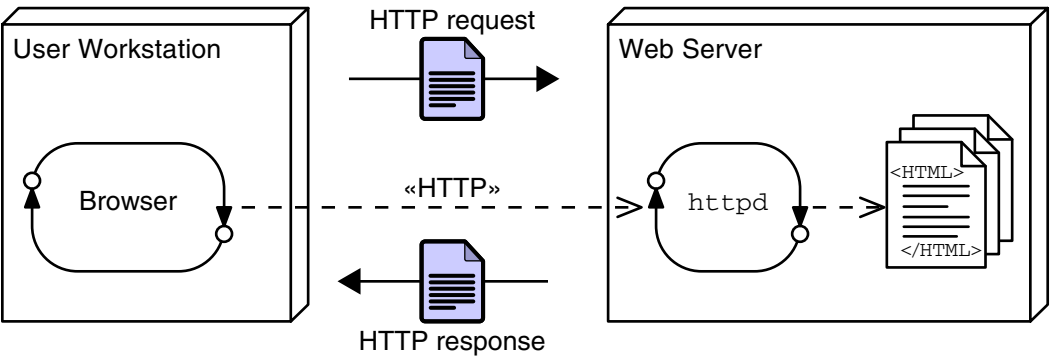
- Java Servlets Specification. [Online]. Available:  
<http://java.sun.com/products/servlet/>
- JavaServer Pages Specification. [Online]. Available:  
<http://java.sun.com/products/jsp/>
- Java Platform, Enterprise Edition 5 API Specification. Available:  
<http://java.sun.com/javaee/6/docs/api>
- Java Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- The Common Gateway Interface. [Online]. Available:  
<http://hoohoo.ncsa.uiuc.edu/cgi/>

# HTTP Revisited

This section describes the HTTP protocol in more detail. In particular, this section defines the structure of the request and response 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. Figure 4-1 shows this client-server architecture.



**Figure 4-1** HTTP Client-Server Architecture

## HTTP Methods

An HTTP request can be made using a variety of methods. Table 4-1 lists the HTTP methods and their descriptions.

**Table 4-1** HTTP Methods and Descriptions

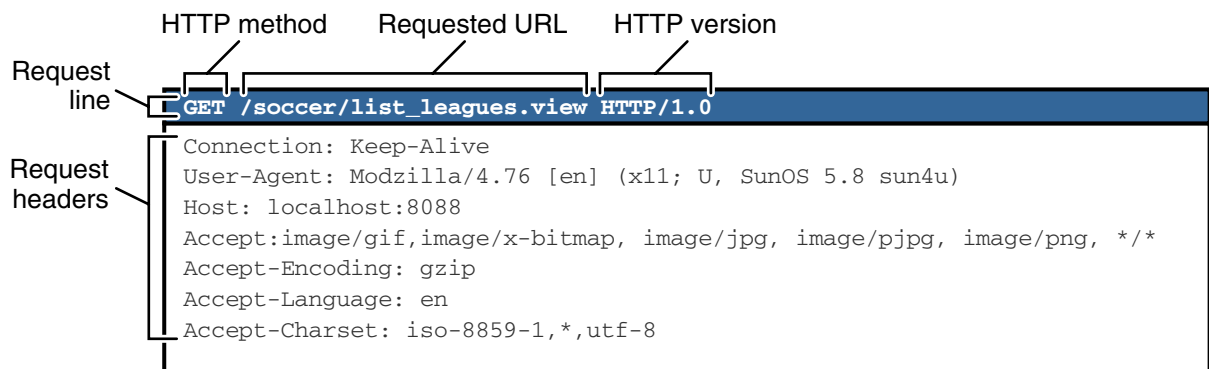
HTTP Method	Description
OPTIONS	Request for the communication options available on the request/response chain
GET	Request to retrieve information identified by the Request-URL
HEAD	Identical to the GET except that it does not return a message-body, only the headers
POST	Request for the server to accept the entity enclosed in the body of the HTTP message
PUT	Request for the server to store the entity enclosed in the body of the HTTP message
DELETE	Request for the server to delete the resource identified by the Request-URI
TRACE	Request for the server to invoke an application-layer loop-back of the request message
CONNECT	Reserved for use with a proxy that can switch to being a tunnel

## HTTP GET Method

One of the two most common HTTP methods is the GET request. A GET method is used whenever the user clicks 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 Navigator™ and FireFox) or the Address field (for Microsoft Internet Explorer). While processing a web page, the browser also issues GET requests for images, applets, style sheet files, and other linked media.

## 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. Figure 4-2 shows an example HTTP request stream.



**Figure 4-2** HTTP Request Stream Example

## HTTP Request Headers

Table 4-2 illustrates some of the HTTP headers that the browser can place in the request. The headers could then be used to determine how the server processes the request.

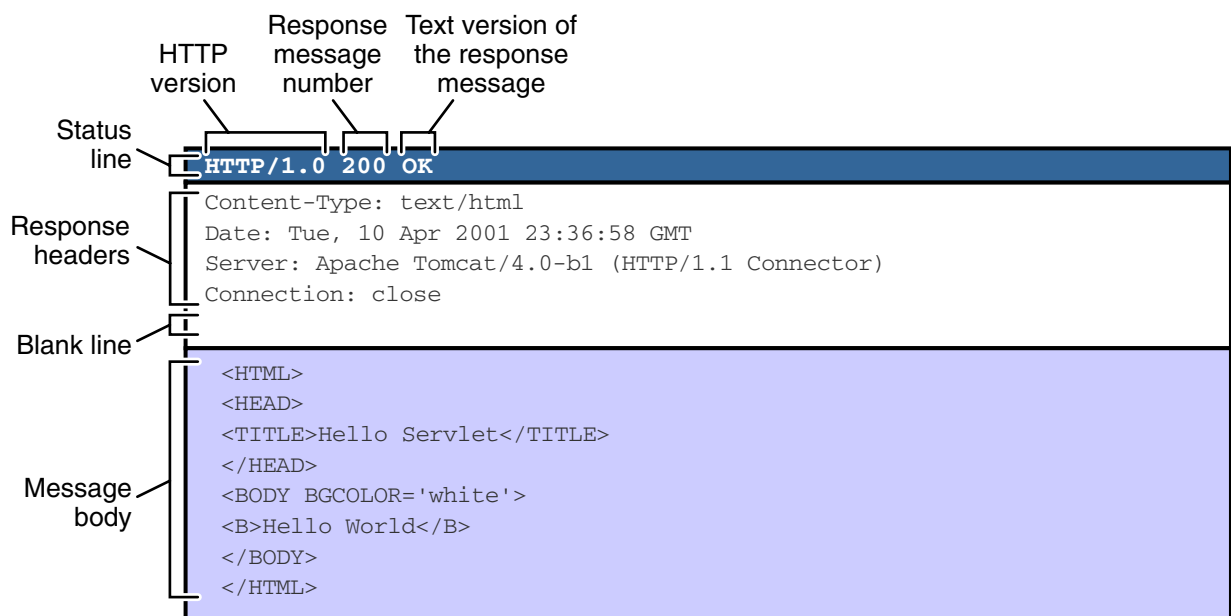
**Table 4-2** Some HTTP Request Headers

Header	Use
Accept	The MIME types the client can receive
Host	The Internet host and port number of the resource being requested
Referer	The address from which the Request-URI was obtained
User-Agent	The information about the client originating the request

## 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. Figure 4-3 shows an example HTTP response stream.



**Figure 4-3** HTTP Response Stream Example



## HTTP Response Headers

Table 4-3 illustrates some of the HTTP headers that the server can place in the response. The headers could then be used to determine how the browser processes the response.

**Table 4-3** Some HTTP Response Headers

Header	Use
Content-Type	A MIME type (such as <code>text/html</code> ), which classifies the type of data in the response
Content-Length	The length (in bytes) of the payload of the response
Server	An informational string about the server that responded to this HTTP request
Cache-Control	A directive for the web browser (or proxies) to indicate whether the content of the response should be cached

## Collecting Data From the User

You saw in “Sending Parameters in an HTTP GET Request” on page 2-8 how parameter data may be encoded in the URL for a GET request. Clearly the user is not normally expected to enter their information in this way. This section examines the HTML “form” mechanism, and shows how user input is commonly achieved.

### HTML Form Mechanism and Tag

If users are to provide input to a web application, they must be provided with a page that prompts for the desired information, and allows them to enter, or select, their desired values. There must also be a mechanism on the page (usually called simply a *form*) to allow the user to indicate that the input is complete and should be sent to the server. This indication is usually called “submitting” the form, and a button that triggers this is called a “submit” button.



---

**Note** – The standard form mechanism discussed here uses a traditional HTTP request/response cycle. Nothing is sent to the server until the user indicates that the form is completed. Increasingly, modern web applications use alternative techniques that allow each keystroke to be sent to the server. In this way, for example, a proposed password might be checked for acceptability, and an indication given with each keystroke of whether the password is good or not. This module does not address such techniques.

---

The HTML form mechanism is invoked using the `<FORM>` tag. The tag allows regular HTML elements to describe a visual layout with prompts, and additional HTML tags to specify input fields that the user can interact with. The tag also specifies the URL that the browser should request when the the form is submitted. This URL is called the “action” of the form.

A single web page might contain many forms. However, you cannot nest form tags inside each other. It is possible to have a form in a web page that has no external presence and no submit button. You must use JavaScript™ code to submit such hidden forms.

# Input Types for Use With Forms

Several input types are provided by HTML for use with forms. The following sections introduce three that have very general utility.

## Text Input Component

When a user must enter a single line of text, the text input component may be used.

Figure 4-4 shows a text input component along with prompts:

This form allows you to create a new soccer league.

Year:

**Figure 4-4** Textfield Component in Netscape

The `input` tag is used to create a textfield component, but you must specify `type='text'` to get a textfield component. The `name` attribute of the `input` tag specifies the field name. The field name and the value entered in the textfield are paired together when the form is submitted to the web container.

This code would create the prompt and the input field shown above:

```
<p>Year: <input type='text' name='theYear' /></p>
```

In the example shown in Figure 4-4 above, submission of the form would result in a GET request something like this:

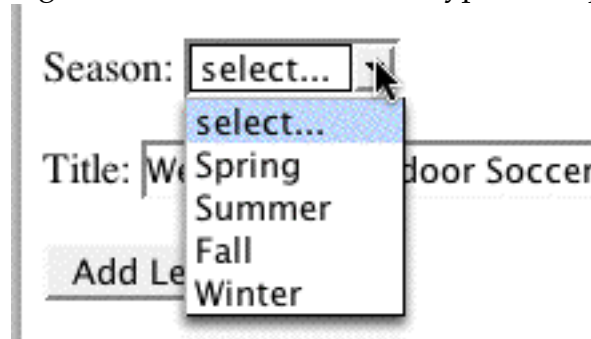
```
GET /path/to/action?theYear=2003
```

Notice how the name specified in the input tag becomes the key, and the user's text becomes the value, in the key/value pair submitted to the server.

## Drop-Down List Component

Often a good user interface will provide a list of options for a user to choose from, rather than require the user to type something longhand. The option tag may be used to create a drop-down list that satisfies this need.

Figure 4-5 below illustrates a typical drop-down list:



**Figure 4-5** A Typical Drop-Down List

Code 4-1 shows the HTML content for this component. The `select` tag is used to create a drop-down list component. Similar to the `input` tag, the `select` tag uses the `name` attribute to specify the name of the form field. One or more `option` tags must be embedded in the `select` tag. Each `option` tag provides a single element in the drop-down list. The data sent in the HTTP request on form submission is based on the `value` attribute of the `option` tag. The text between the start and end `option` tags is the content that is displayed in the browser drop-down list. For example, notice that the first option has the value `UNKNOWN`, but the display text is "select...".

```

1 Season: <select name='season'>
2         <option value='UNKNOWN'>select...</option>
3         <option value='Spring'>Spring</option>
4         <option value='Summer'>Summer</option>
5         <option value='Fall'>Fall</option>
6         <option value='Winter'>Winter</option>
7     </select> <br/><br/>

```

### Code 4-1 Drop-Down List Component HTML Tag

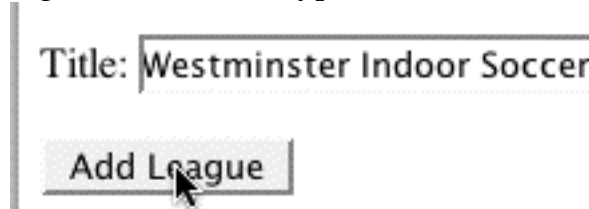
If the form were submitted from the selection shown, the key/value pair that would become part of the HTTP request would be:

season=select...

## Submit Button

When all the data have been provide in the form, the user must press the submit button to have the form data sent to the server for processing. A submit button can be given a text label at the designer's discretion, though browsers will usually mark the button "Submit" or "Submit Query" if nothing is done to explicitly choose the label.

Figure 4-6 shows a typical submit button:



**Figure 4-6** A Typical Submit Button

Code 4-2 below shows the HTML code for this component. The `value` attribute of the `input` tag becomes the label of the button. When a submit button is clicked by the user, the web browser is responsible for generating an HTTP request to the web container. By default, no data is sent in the request for the submit button component.

```
<input type='submit' value='Add League' />
```

**Code 4-2** Submit Button HTML Tag

## An Example HTML Form

Now that you have seen all of the pieces of form, review the whole page. Code 4-3 below shows the HTML content while Figure 4-7 on page 4-15 shows the resulting output.

```

1  <html>
2
3  <head>
4  <title>Duke's Soccer League: Add a New League</title>
5  </head>
6
7  <body bgcolor='white'>
8
9  <!-- Page Heading -->
10 <table border='1' cellpadding='5' cellspacing='0' width='400'>
11 <tr bgcolor='#CCCCFF' align='center' valign='center' height='20'>
12   <td><h3>Duke's Soccer League: Add a New League</h3></td>
13 </tr>
14 </table>
15
16 <p>
17 This form allows you to create a new soccer league.
18 </p>
19
20 <form action='add_league.do' method='POST'>
21 Year: <input type='text' name='year' /> <br/><br/>
22 Season: <select name='season'>
23         <option value='UNKNOWN'>select...</option>
24         <option value='Spring'>Spring</option>
25         <option value='Summer'>Summer</option>
26         <option value='Fall'>Fall</option>
27         <option value='Winter'>Winter</option>
28     </select> <br/><br/>
29 Title: <input type='text' name='title' /> <br/><br/>
30 <input type='submit' value='Add League' />
31 </form>
32
33 </body>
34 </html>

```

**Code 4-3** Example Complete Form in an HTML Page

The screenshot shows a Netscape browser window with the title "Duke's Soccer League: Add a New League - Netscape". The address bar displays "http://localhost:8080/controller/admin/add\_lea". The main content area has a purple header bar with the text "Duke's Soccer League: Add a New League". Below this, a message states "This form allows you to create a new soccer league." The form includes three input fields: "Year:" with the value "2003a", "Season:" with a dropdown menu showing "select...", and "Title:" with the value "Westminster Indoor Soccer". At the bottom of the form is a button labeled "Add League". The browser's status bar at the bottom shows "Done" and various navigation icons.

**Figure 4-7** Appearance of the Complete Form

## How Form Data Are Sent in an HTTP Request

The data that the user enters in the HTML form must be sent to the server when the submit button is selected. The web browser is responsible for creating an HTTP request using the URL in the action attribute of the form tag. The browser also collects all of the data in the form fields and packages it into the HTTP request. This section describes how this is done.

### 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. Notice the key=value pairs that are separated from each other using the ampersand (&) symbol. Notice also that spaces are encoded using a plus sign. A general encoding scheme using hexadecimal character codes is also specified for many special characters in URL parameters.

Syntax:

```
fieldName1=fieldValue1&fieldName2=fieldValue2&...
```

Examples:

```
username=Fred&password=C1r5z
```

```
season=Winter&year=2004&title=Westminster+Indoor+Soccer
```



## HTTP GET Method Request

Form data are carried in the URL of the HTTP GET request:

```
GET /admin/add_league.do?year=2003&season=Winter&title=Westminster+Indoor
HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.4
Gecko/20030624 Netscape/7.1
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain
video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

In this example, the parameter and their values are as listed below:

**Table 4-4** Parameters and Values

Parameter Name	Parameter Value
year	2003
season	Winter
title	Westminster Indoor

## HTTP POST Method Request

Form data is contained in the body of the HTTP request:

```
POST /admin/add_league.do HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O; en-US; rv:1.
Gecko/20030624 Netscape/7.1
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plai
0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://localhost:8080/controller/admin/add_league.html
Content-Type: application/x-www-form-urlencoded
Content-Length: 55

year=2003&season=Winter&title=Westminster+Indoor
```

The parameters and values illustrated here are the same as in the previous example.

## HTTP GET and POST Methods

The HTTP GET method is used in these conditions:

- The processing of the request is idempotent.  
This means that the request does not have side-effects on the server or that multiple invocations produce no more changes than the first invocation.
- The amount of form data is small.
- You want to allow the request to be bookmarked along with its data.

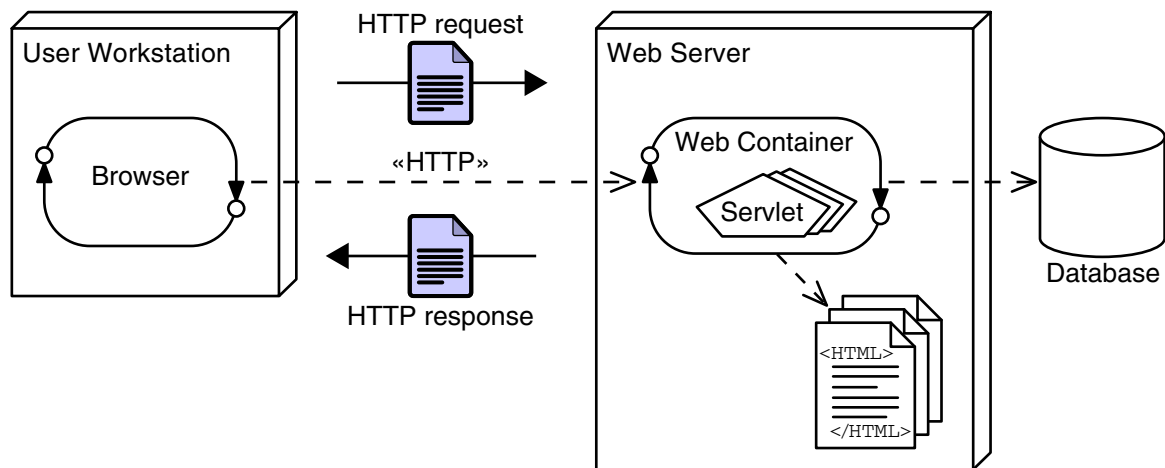
The HTTP POST method is used in these conditions:

- The processing of the request changes the state of the server, such as storing data in a database.
- The amount of form data is large.
- The contents of the data should not be visible in the URL (for example, passwords).

## Web Container Architecture

Java servlets are components that must exist in a web container. The web container is built on top of the Java Platform, Standard Edition (Java SE) platform and implements the servlet API and all of the services required to process HTTP (and other Transmission Control Protocol/Internet Protocol [TCP/IP]) requests.

Figure 4-8 shows the architecture for a web server that uses a web container.



**Figure 4-8** Web Container Architecture

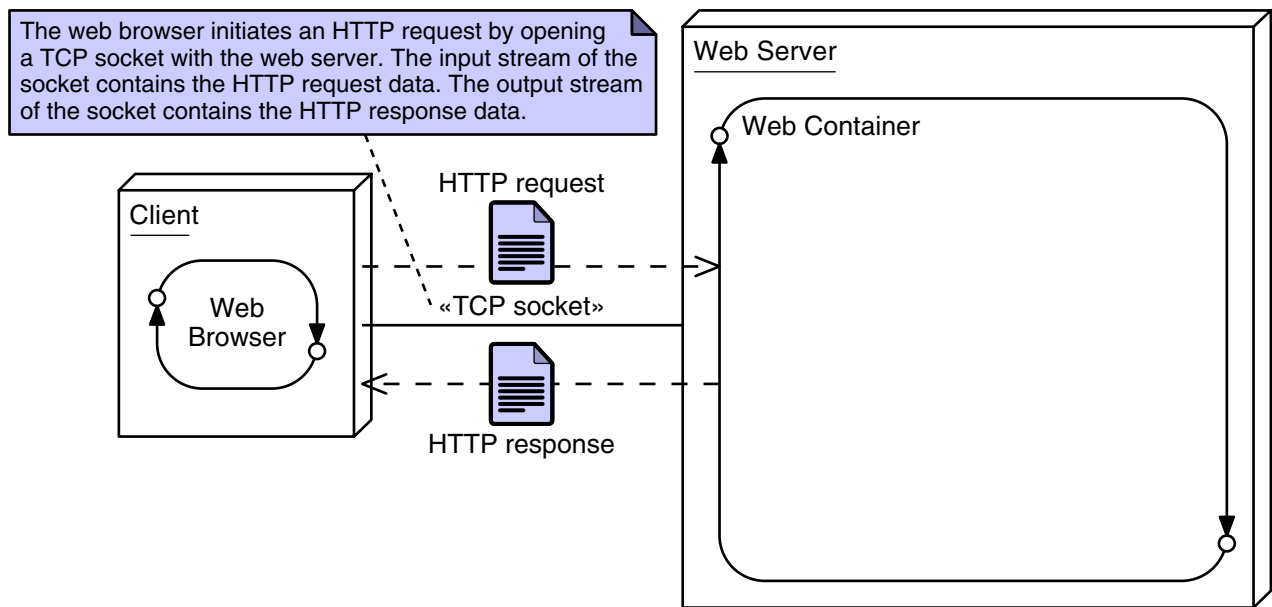
The web container activates the servlet that matches the request URL by calling the service method on an instance of the servlet class. Specifically, the activation of the service method for a given HTTP request is handled in a separate thread within the web container process.

## Request and Response Process

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

### Browser Connects to the Web Container

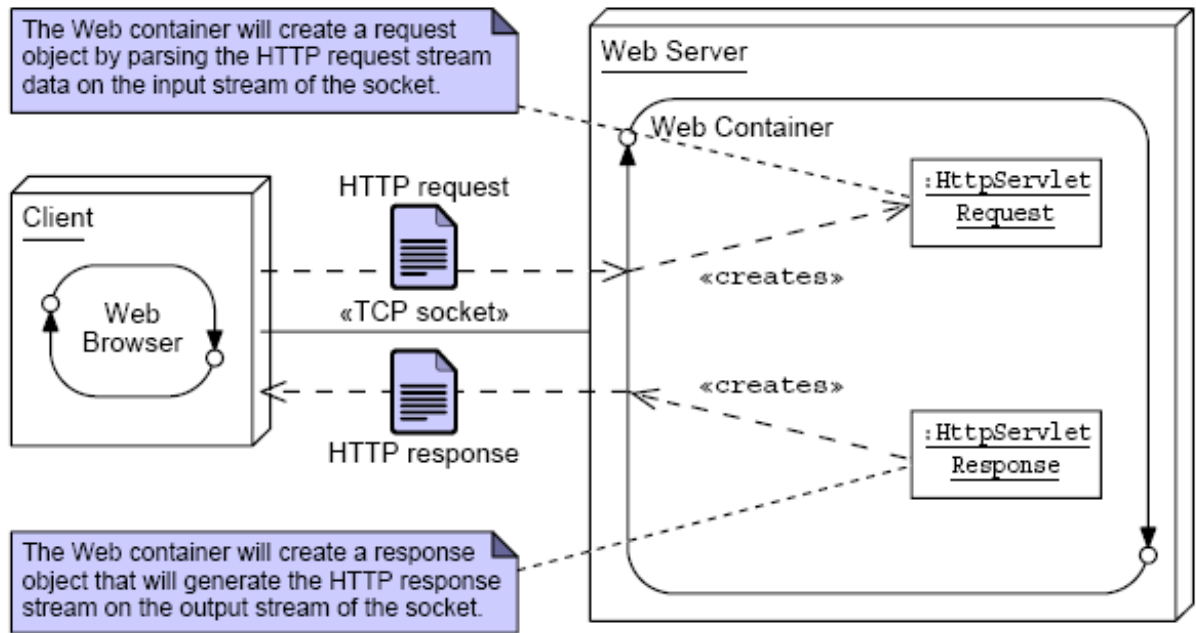
The first step in this process is the web browser sending an HTTP request to the web container. To process a request, the browser makes a TCP socket connection to the web container. Figure 4-9 shows this step.



**Figure 4-9** Web Browser Connects to the Web Container

## Web Container Objectifies the Input/Output Streams

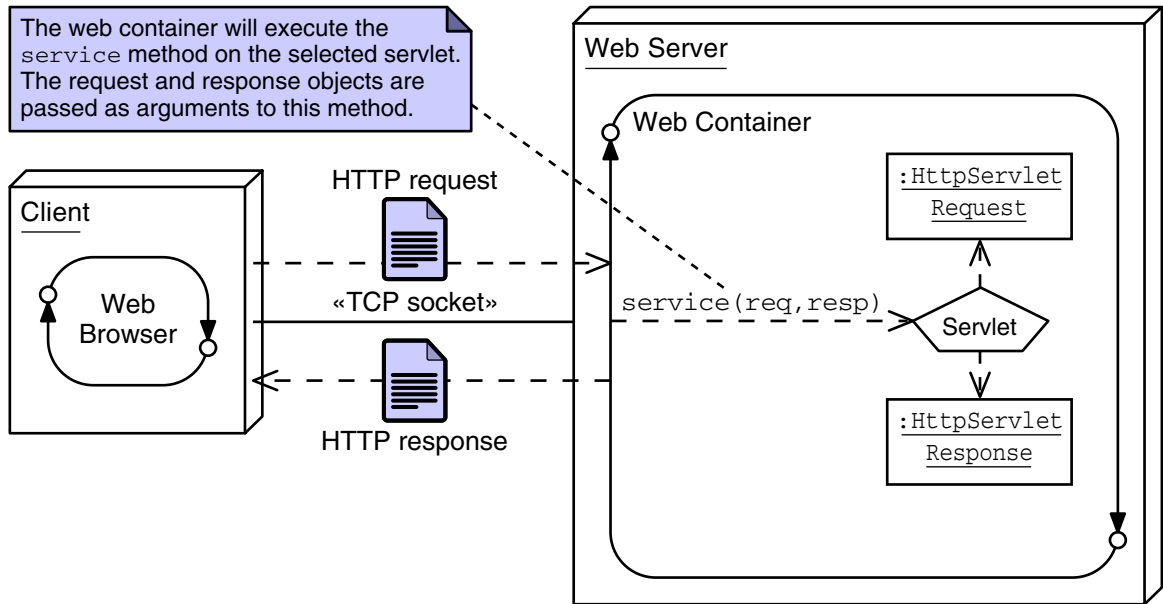
Next, the web container creates an object that encapsulates the data in the request and response streams. These two objects represent all of the information available in the HTTP request and response streams. Figure 4-10 shows this step.



**Figure 4-10** Web Container Constructs the Request and Response Objects

## Web Container Executes the Servlet

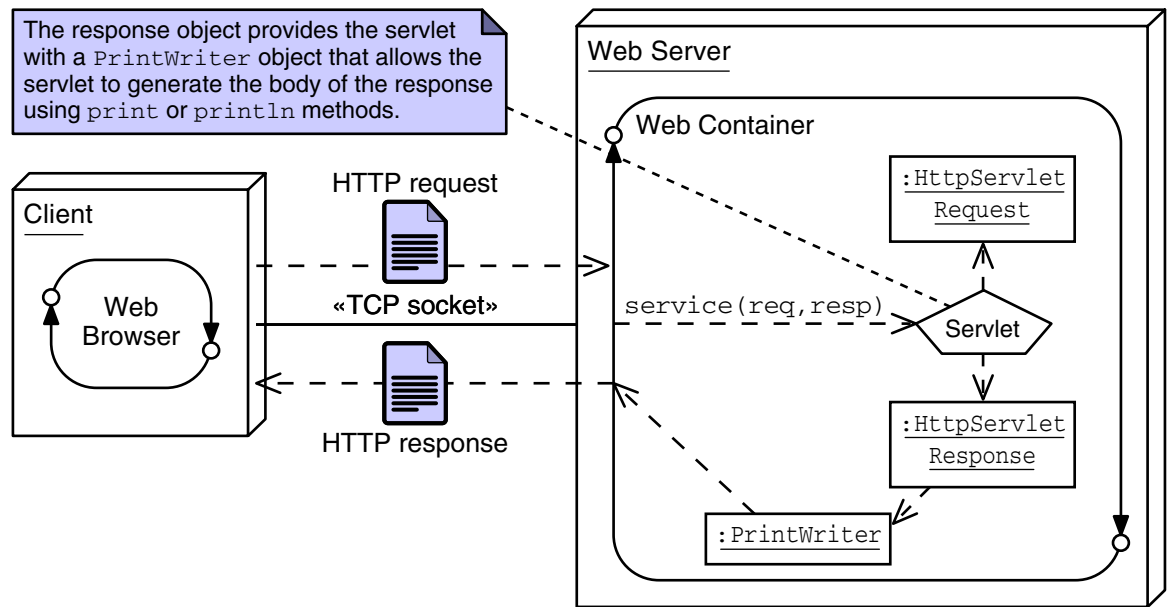
The web container then executes the `service` method on the requested servlet. The request and response objects are passed as arguments to this method. The execution of the `service` method occurs in a separate thread. Figure 4-11 shows this step.



**Figure 4-11** Web Container Executes the Servlet

### Servlet Uses the Output Stream to Generate the Response

Finally, the text of the response generated by the servlet is packaged into an HTTP response stream, which is returned to the web browser. Figure 4-12 shows this step.

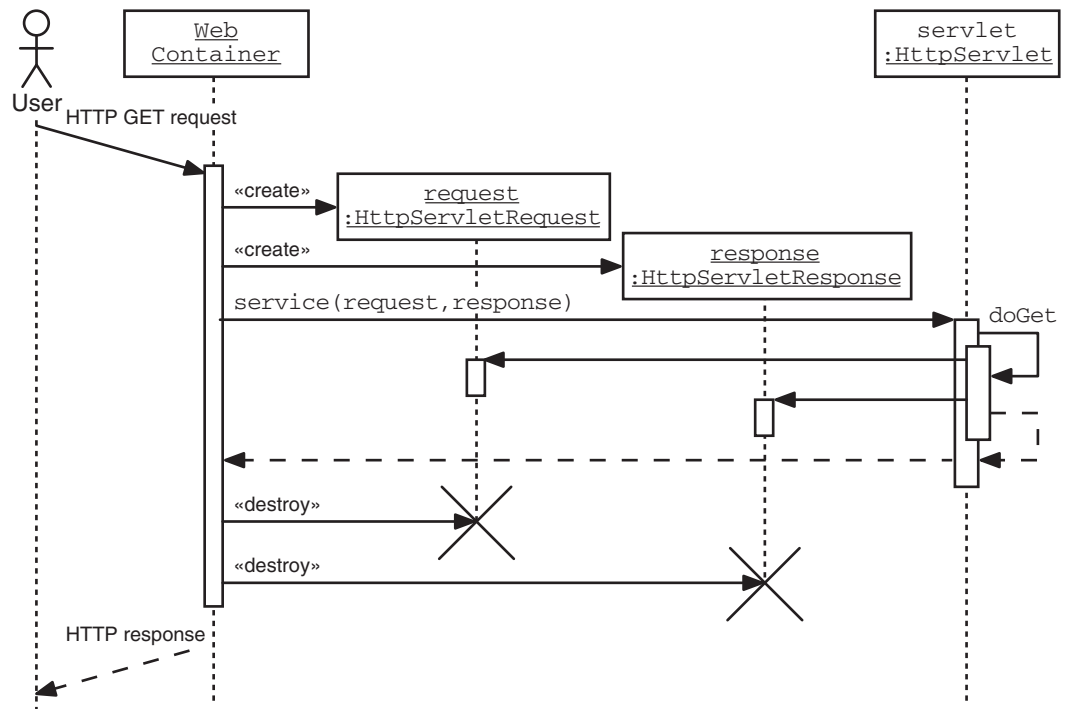


**Figure 4-12** Servlet Generates a Dynamic Response



## Sequence Diagram of an 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. Figure 4-13 shows this process.



**Figure 4-13** HTTP GET Request Sequence Diagram

# The HttpServlet API

To create a servlet that responds to an HTTP request, you should create a class that extends the HttpServlet abstract class (provided in the javax.servlet.http package). The Unified Modeling Language (UML) class diagram in Figure 4-14 shows the relationships between your servlet class and the interfaces to access data in the request and response streams.

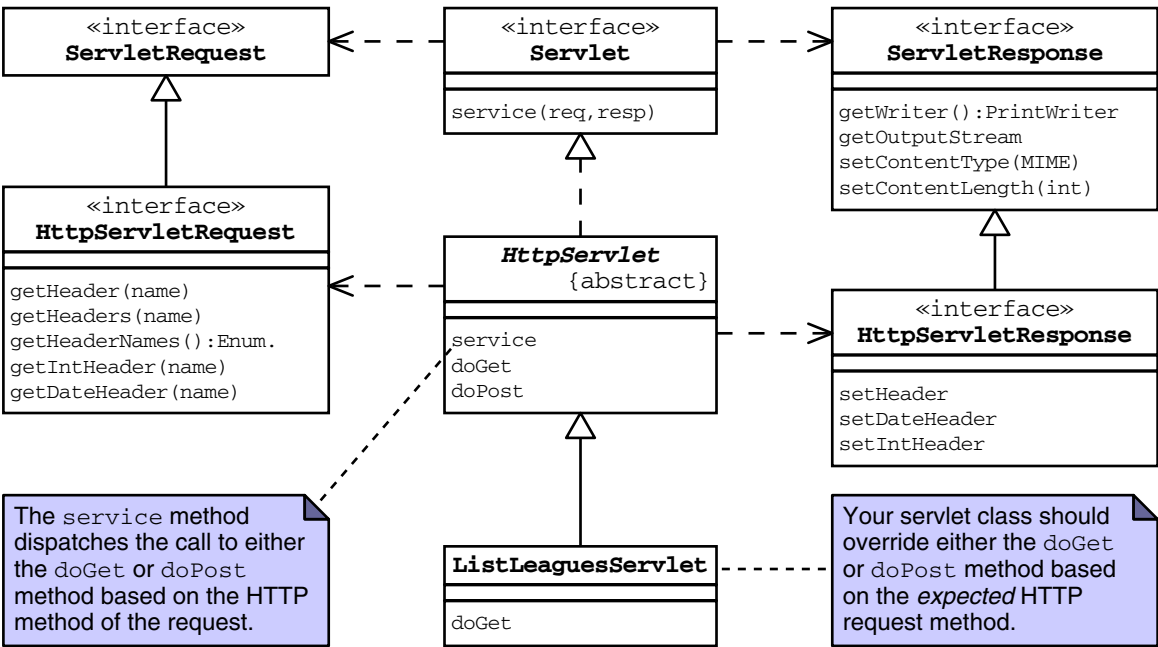


Figure 4-14 The HttpServlet API

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. For example, if the servlet needs to respond to an HTTP GET request, you should override the `doGet` method.

Table 4-5 lists the HTTP request methods and their corresponding HttpServlet methods.

**Table 4-5** Corresponding HttpServlet Methods

HTTP Method	Corresponding HttpServlet Method
OPTIONS	doOptions
GET	doGet
HEAD	doHead
POST	doPost
PUT	doPut
DELETE	doDelete
TRACE	doTrace
CONNECT	doConnect

## 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 might represent integer or date information. There are two convenience methods, `getIntHeader` and `getDateHeader`, that perform the conversion for you.



**Note** – Figure 4-14 on page 4-26 shows only a small portion of the API for the `HttpServletRequest` interface. There are more descriptions of the methods in the API throughout this course.

The following code highlights demonstrate how header information can be used to select the content that gets generated for the client.

```
boolean displayXHTML = false;
String userAgent = request.getHeader("User-Agent");
if(userAgent.startsWith("Mozilla/5.0")) {
    // browser can handle XHTML content
    displayXHTML = true;
}
```

## The `HttpServlet` API

---

```
if(displayXHTML) {  
    // XHTML content output here  
} else {  
    // regular HTML content output here  
}
```



**Discussion** – What are some ways that you could use request headers to augment a web application?

## The `HttpServletResponse` API

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

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 similar to a letter in an envelope. The web container is responsible for packaging the response text with the header information.

You are responsible for generating the response text. As you have seen, this is commonly done using a JSP acting as the view in an MVC architecture, but it can be done directly by the servlet. If your servlet is to create the output directly, it must retrieve the response body stream using either the `getWriter` or `getOutputStream` method. To generate an HTML response, use the `getWriter` method, which returns a character stream, specifically a `PrintWriter` object. If you generate a binary response, such as a graphic image, use the `getOutputStream` method, which returns a binary stream, specifically a `ServletOutputStream` object.

You should 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 4-14 on page 4-26 shows only a small portion of the API for the `HttpServletResponse` interface. There are more descriptions of the methods in the API throughout this course.

---

The following code highlights demonstrate how you can use header information to disable browser and proxy caching.

```
response.setContentType("text/html");
response.setHeader("Cache-Control", "no-cache");
response.setHeader("Cache-Control", "private");
```

When the `Cache-Control` header is set to `no-cache`, this is an instruction for the browser to not reuse the response for a subsequent request without revalidation with the origin server. The `private` setting for `Cache-Control` indicates that the response is intended for a single user and should not be stored in a shared cache (for example, the cache in a proxy server).

## Handling Errors in Servlet Processing

If a servlet is unable to complete processing a request, you have two choices. You can create an HTML response that describes the problem to the user, in terms that are appropriate to the user, or you can have the servlet throw an exception.

In general, you should not allow exceptions to reach the user. There are two reasons for this; first, the user has no interest in the exception, it might be frightening to them and is unlikely to help them resolve the problem. Second, exceptions often include detailed technical information that might be used by malicious persons to find weaknesses in the security of your systems. This observation suggests that controlled selection of a page that reports the semantic nature of the problem is commonly preferable to throwing an exception.

One situation in which throwing an exception might be appropriate is when processing page fragments, that is, either servlets or JSPs that are used by inclusion into a larger compound page. In this situation, the fragment is unlikely to be able to decide what to do about the error, but the host page can make a sensible decision. The call to include the page will simply throw the exception that was issued by the included page fragment, allowing the host to use a `try/catch` block to intercept and handle the problem.

If, for any reason, you decide to throw an exception from a `doXXX()` servlet method, you should wrap the exception in either a `ServletException`, or an `UnavailableException`. The `ServletException` is used to indicate that this particular request has failed. `UnavailableException` indicates that the problem is environmental, rather than specific to this request. For example, if the servlet finds that it cannot access a needed database, it is likely that subsequent requests will also fail. The `UnavailableException` can indicate a duration of failure, in which case the server is permitted to resume sending requests to that servlet after the delay. However, the server is permitted to abandon and destroy the servlet entirely should it choose to do so.

# The HTTP Protocol and Client Sessions

HTTP is a stateless protocol. That means that every request is, from the perspective of HTTP, entirely separate from every other. There is no concept of an ongoing conversation. This provides great potential for scalability and redundancy in servers, because any request can go to any one of a collection of identically equipped servers.

The stateless nature of HTTP creates a problem for many web applications, however, in that nothing ties a request to put a book into a shopping cart with a request from the same browser to check out. Clearly this must be resolved if practical web applications are to be written.

Consider that this problem is rather like sending postcards to a popular friend. The postcard mechanism, as provided by the post office, does not of itself identify the sender. However, you might put a sender label in the corner, or you might sign the card "best wishes, from John." In any case, the fact that every postcard carries information identifying you allows your friend to separate your postcards from those of his other friends.

Web servers in a Java EE environment are required to provide mechanisms for identifying clients and associating each client with a map that can be used to store client specific data. That map is implemented using the `HttpSession` class.

# The HttpSession API

The Servlet specification provides an HttpSession interface that lets you store session data, referred to as *attributes*. You can store, retrieve, and remove attributes from the session object. The servlet has access to the session object through two getSession methods in the HttpServletRequest object. Figure 4-15 shows this API.

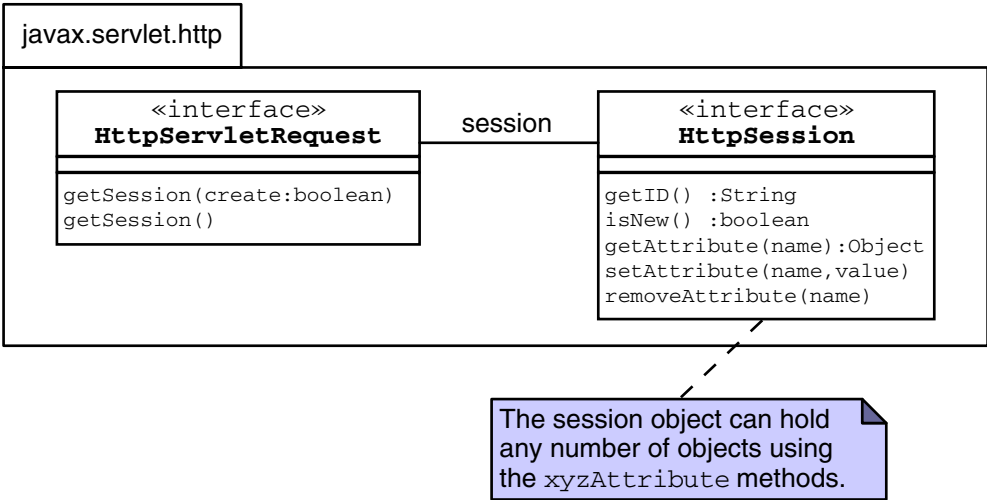


Figure 4-15 The HttpSession API

The first time a browser connects to a particular web application, there will be no session associated with the connection. Calling getSession() will create it automatically, or retrieve the existing one. The getSession(boolean) method, by contrast, can be used to control the creation of a new session object. If you call this method with a true argument, then a new session object is created if it did not already exist. If you call this method with a false argument, then null is returned if a session did not already exist. Calling the getSession() method is equivalent to calling getSession(true).

## Storing Session Attributes

To store a value in a session object, use the setAttribute method. For example, if a servlet has prepared an object referred to by a variable called league, this could be placed in the session using the following code:

```
HttpSession session = request.getSession();
session.setAttribute("league", league);
```



The `getSession` method returns, or perhaps creates, the session associated with this client. 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 returns the same object.

Code that has a handle on the session object stores data using the `setAttribute` method. Notice that the session is a map, so a key must be used to identify this particular object when it is later retrieved.



---

**Note** – Only one session object is created for a given client within a single web application. However, several use cases in the same web application can share the session object. Therefore, attribute names should be unique to avoid ambiguity within the complete web application.

---

## Retrieving Session Attributes

You can use the `getAttribute` method to retrieve data previously stored in the session object. For example, to retrieve the league object stored in the earlier example, you could use the following code:

```
HttpSession session = request.getSession();  
League theLeague =  
    (League)session.getAttribute("league");
```

Notice that the return type of `getAttribute` is `Object`, so a cast must be used before assigning the returned value to a reference variable.

The `getAttribute` method returns `null` if there is no attribute in the session with the name specified in the argument to the `getAttribute` method.

Session attributes can be read in EL, and you will see how to achieve this in a later module, but as mentioned before for parameters, it might be better design to restrict the view to reading data from the model, so that the view avoids dependencies on anything except the model.

## Closing the Session

When the web application has finished with a session, such as if the user logs out, you should call the `invalidate` method on the `HttpSession` object. If you fail to invalidate the session, you might clutter up the memory of the server with lots of unnecessary objects. Such memory clutter is detrimental to performance and scalability.



**Note** – In a secured environment, the `invalidate` method is likely to delete authentication information forcing, the user to login again.

### Additional Session Methods

To avoid excessive memory use, the application server can also invalidate the session after a period of inactivity. The default timeout depends on your web container, but is configurable using the `setMaxInactiveInterval` method. This and other methods, shown in Figure 4-16 below, provide more information about the session object.

<interface> <b>HttpSession</b>
<code>invalidate()</code> <code>getCreationTime() :long</code> <code>getLastAccessedTime() :long</code> <code>getMaxInactiveInterval() :int</code> <code>setMaxInactiveInterval(int)</code>

**Figure 4-16** Other Session API Methods

### Destroying the Session Issues

Destroying a session is a design decision, and you should consider the implications. 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 action classes handling these other use cases might lose data stored in the destroyed session. One choice is to remove only the attributes used in the current use case. The action class would then have to be designed to check that its attributes are `null` before starting the use case.



**Note** – If an application is redeployed, the session is, naturally, destroyed. However, the client browser will continue to send the session ID, which might cause some confusion. This is only a concern in the development environment, and can be overcome if the browser is restarted.

## Architectural Consequences of Sessions

If HttpSession objects are created for every client, and are allowed to exist for a long time even after the client is no longer using the site actively, it is very easy to end up with a great many sessions in memory at one time. In extreme cases this can cause severe performance problems. Because of this, you should avoid creating sessions where they are not necessary, and you should take care to clean up sessions that are no longer in use as soon as you can confidently do so.



**Note** – JSPs create a session by default for every user. To disable this behavior, be sure to include this page directive:

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

## Session Configuration

Sessions can be maintained by the container using cookies, URL rewriting, or an SSL encrypted communications channel to the browser. The preferred mechanism may be specified in the web.xml file.

Session default timeout can also be specified in the web.xml file. Below is an example of a web.xml file specifying a default timeout of 30 minutes, and requesting SSL tracking of sessions.

```
<web-app ...>
  <session-config>
    <session-timeout>30</session-timeout>
    <tracking-mode>SSL</tracking-mode>
  </session-config>
</web-app>
```

## Using Cookies for Client-Specific Storage

Cookies allow a web server to store information on the client machine. Data are stored as key-value pairs in the browser and are specific to the individual client.

- 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 life span and are flushed by the client browser at the end of that life span.
- Cookies can be labelled "HTTP-Only" which means that they are not available to scripting code such as JavaScript. This enhances security.

# Cookie API

You can create cookies using the `Cookie` class. You can add cookies to the response object using the `addCookie` method. This sends the cookie information over the HTTP response stream. 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.

Figure 4-17 shows key aspects of the Cookie API.

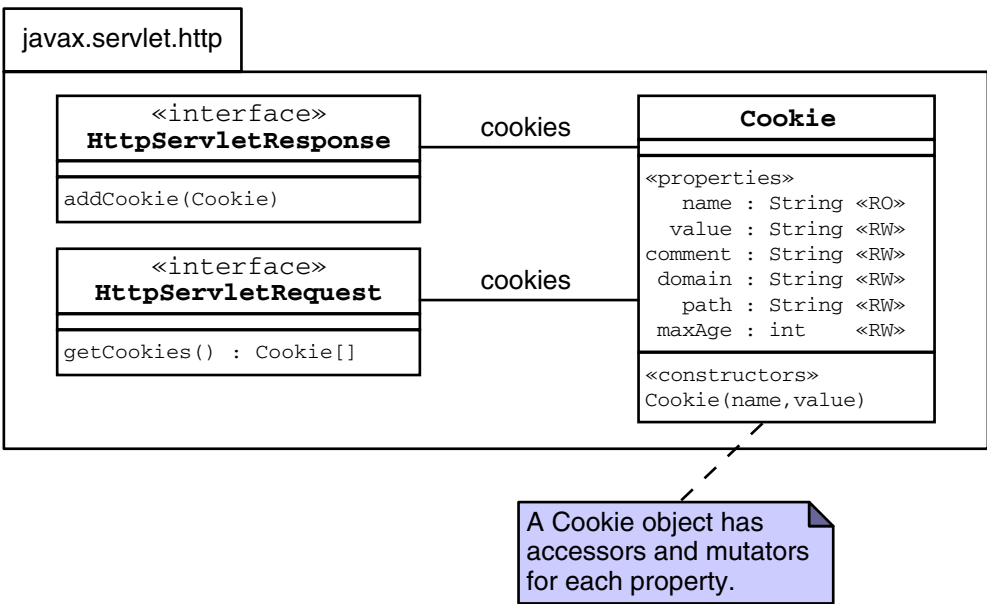


Figure 4-17 Aspects of the Cookie API



**Note** – The Servlet 3.0 specification added an additional property `httpOnly`, with corresponding methods `setHttpOnly(boolean)` and `boolean isHttpOnly()`.

## Using Cookies Example

In this example, there is a visitor to your web site and you want to store the user's name so that on the next visit to the site, the screen is personalized for the user. In your servlet, you could use the following code to store that cookie:

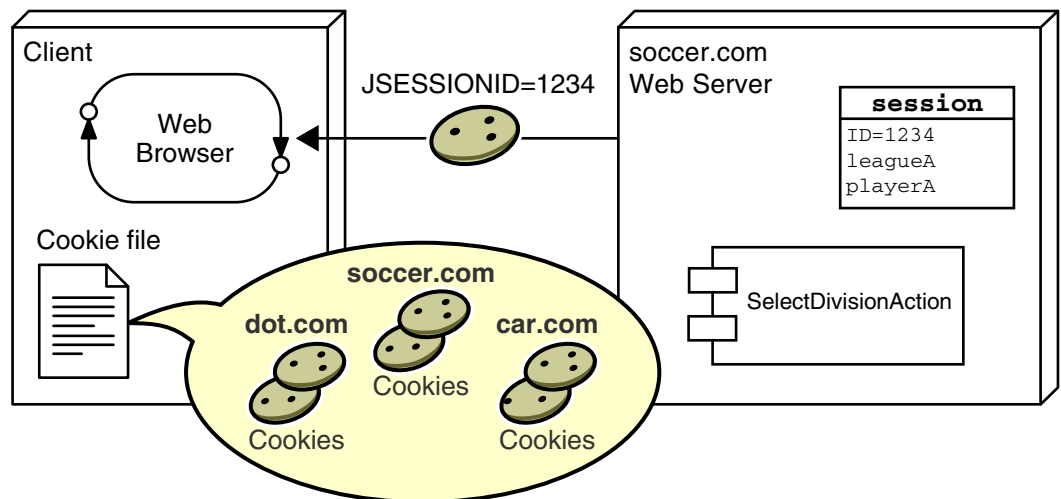
```
String name = request.getParameter("firstName");
Cookie c = new Cookie("yourname", name);
response.addCookie(c);
```

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();
    }
}
```

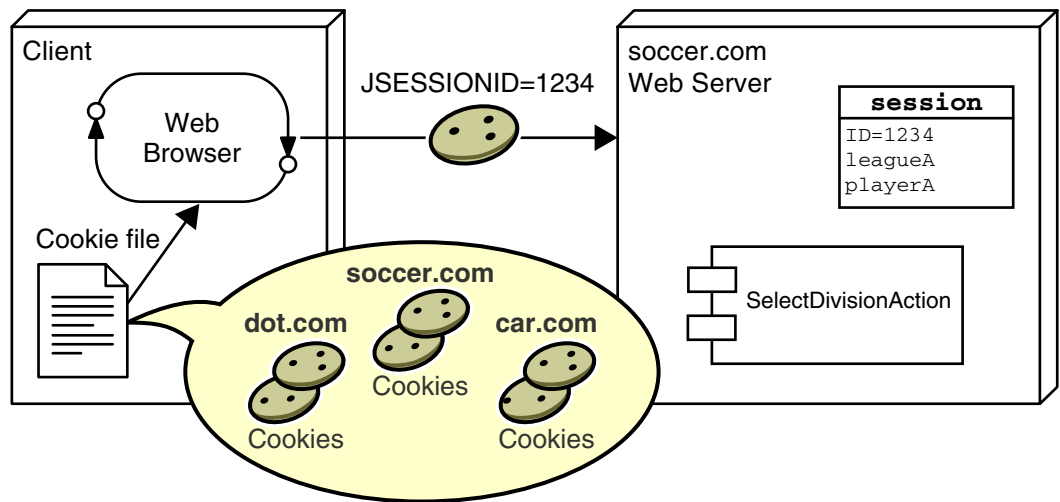
## Performing Session Management Using Cookies

You can use HTTP cookies to perform session management. The web container stores the session ID on the client machine, as shown in Figure 4-18.



**Figure 4-18** Web Container Session ID Cookie Storage

Figure 4-19 shows that every HTTP request from the client includes the session ID cookie that was stored on the client's machine.



**Figure 4-19** Web Container Session ID Cookie Retrieval

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

The cookie mechanism is the default `HttpSession` strategy. You do not need to code anything special in your servlets to use 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.

You can detect if cookies are being used for session management using the `isRequestedSessionIdFromCookie` method in the `HttpServletRequest` interface. This returns `true` if cookies are used. A similar method, `isRequestedSessionIdFromURL` in the `HttpServletRequest` interface, returns `true` if URL-rewriting is used for session management.

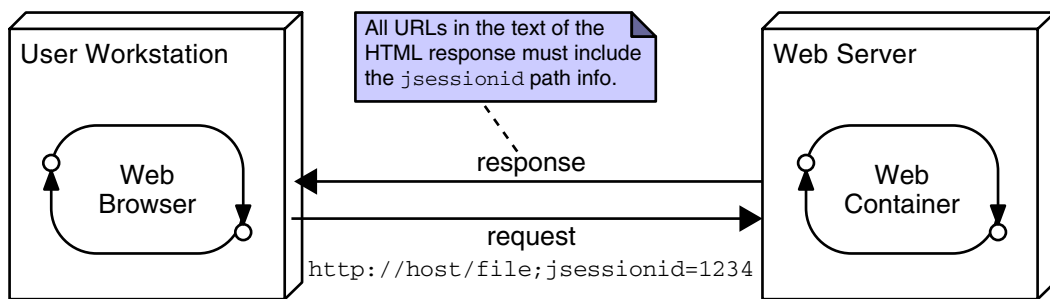
**Note** – The Servlet specification mandates that the name of the session ID cookie be `JSESSIONID`.



# Using URL-Rewriting for Session Management

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. If that fails (that is, the user has cookies turned off in the browser), then the web container tries 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, as shown in Figure 4-20.



**Figure 4-20** URL-Rewriting Session Management



## URL-Rewriting Implications

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 `SelectLeagueAction` 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, as shown in Code 4-4.

### Code 4-4 URL Encoding

```
120
121 // Present the form
122 out.println("<form action='"
123           + response.encodeURL("enter_player.do")
124           + "' method='POST'>");
125
```

The process of URL-rewriting can be tedious. A static HTML page or form now must 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 uses cookies, then you must consider that the web container might need to use URL-rewriting.

## Summary

This module introduced these concepts:

- The environment in which the servlet runs
- HTTP headers and their function
- HTML forms and the transfer of data from users to a servlet
- How the web container transfers a request to the servlet
- HttpSession objects

## Module 5

---

# Container Facilities for Servlets and JSPs

---

## Objectives

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

- Describe the purpose of deployment descriptors
- Determine the context root for a web application
- Create servlet mappings to allow invocation of a servlet
- Create and access context and init parameters
- Use the `@WebServlet` and `@WebInitParam` annotations to configure your servlets

## Relevance

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

- How can you write an application without knowing everything about the environment in which it will be used?
- How can you configure a web application if you don't have access to the command line that launched it?

## 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 Platform, Enterprise Edition 5 API Specification. Available:  
<http://java.sun.com/javaee/6/docs/api>
- Java Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- The Common Gateway Interface. [Online]. Available:  
<http://hoohoo.ncsa.uiuc.edu/cgi/>

## Packaging Web Applications

Most web applications are composed of many web pages and a considerable amount of functionality. This implies that many files of many types—HTML files, image files, JSPs, Servlet classes, supporting Java class files, and so on—must be placed in the correct place in the container to ensure a correctly functioning application.

Historically, Java has used a Java Archive (JAR) to package multiple class and resource files into a single file. This technique is used to support deployment of all kinds of libraries and applications, and the same is true of web applications that are deployed in a web container according to the Java EE specification.

The JAR file that is used to package a web application has a specific set of structural and content rules that make it a little different from other, more general, archives. Because of this, the file is given the extension `.war`, indicating that it is a web archive, rather than `.jar`, which would not distinguish it from archives created for any other purpose. The file is often referred to as a WAR file, or simply a WAR.

## The Default Context Root

Java EE web containers commonly support multiple web applications, so each application is given a *context root*. The context root is the base URL that is used as a prefix when locating pages and servlets within the application. The most basic default for the context root is that it uses the name of the war file from which the application was deployed.

For example, if a war file called `AnApplication.war` contains a file `startingPoint.html` in the root of the archive, then when deployed to a local server, the URL used to reach the `startingPoint.html` file might be:

```
http://localhost:8080/AnApplication/startingPoint.html
```

In this way, the context root is `/AnApplication`.



---

**Note** – This context root may be overridden, for example if the WAR is embedded in an Enterprise Archive (EAR) file as part of a larger enterprise application.

---

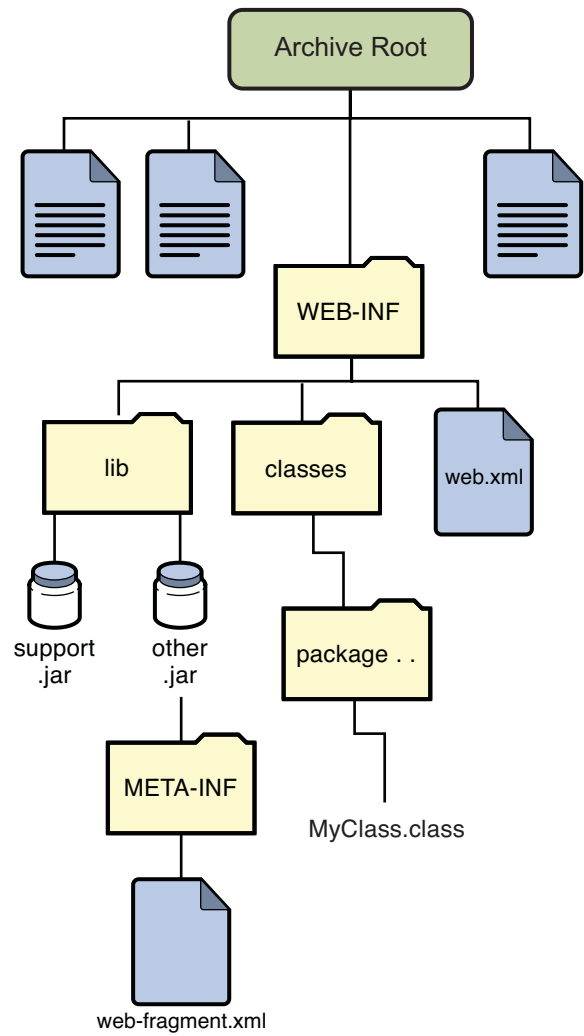
## Essential Structure of a WAR file

Figure 5-1 below shows the essential structure of a WAR file. Notice that the root of the archive, and arbitrary directories below the root are all proper places to place HTML files, JSPs, and images and other resources. Content in these locations will be directly accessible from the application's context root (this is not the same as the root of the server). A special directory, called `WEB-INF` is also placed in the root of the archive. If resources are placed there, the container must be given special instructions to make those resources available to a client.

Inside `WEB-INF` is a file called `web.xml`. This is called the deployment descriptor. As the name indicates, it is an XML structured file. Its purpose is to provide configuration information for the application and its components. This is the place where instructions can be given so that resources under `WEB-INF` become available.

Java classes can be placed under the path `WEB-INF/classes`. This path effectively becomes part of the classpath for the web application. Library JARs can be placed under the path `WEB-INF/lib`; these two are added to the effective classpath for the web application. Resources, such as HTML pages and images, located in the `META-INF/resources` directory of JAR files that are placed in `WEB-INF/lib` directories are also available in the application.

In Java EE 5 and older releases, it was critical to be intimately familiar with this file before anything significant could be achieved with servlets. In Java EE 6, however, the file might be omitted completely, as many of the elements of the deployment descriptor can be inferred by means of annotations applied to the Java source files.



**Figure 5-1**    Structure of a WAR File



# Deployment Descriptors

Configuration information is a problematic issue when the time comes for maintenance. Good object-oriented (OO) practice suggests keeping related things together, but also suggests keeping things that change independently separate. Unfortunately, configuration information fits both these criteria, suggesting it should be separate from the source code, but also kept with it. Clearly this is an unworkable contradiction.

In versions of Java EE prior to EE 6, deployment configuration was supported using the `web.xml` file. This file was required and is generally known as the deployment descriptor (DD). This type of external deployment descriptor facilitates reuse of code in differing environments without problems arising from hard-coded values needing to change, and supports the notion of keeping separate things that change separately.

However, in many cases, particularly during development, the need to have configuration information maintained in a separate file is inconvenient. This reflects the OO guidance that suggests keeping related things together. So, since Java EE 6, many configuration items can be specified using annotations embedded in the source file.

Partly because of the advent of annotations in this context, the `web.xml` file is optional since Java EE 6.

In addition to allowing configuration to be done using annotations, another method of specifying deployment information is also supported. This is the use of `web-fragment.xml` files in libraries.

## The `web-fragment.xml` Files

The `web-fragment.xml` file provides a mechanism to include deployment information that is specific to a library, without having to edit the main deployment descriptor for the whole web application. This reflects the OO principle of keeping unrelated things apart.

A `web-fragment.xml` file, located under the `META-INF` directory of a JAR file placed within the `WEB-INF/lib` directory of a WAR archive, will be read and the configuration contained within will be integrated into the configuration of the web application as a whole.

The following sections address several commonly used features of deployment. Because you should expect to find many projects using the `web.xml` deployment descriptor for a long time after the release of Java EE 6, you will be shown how to achieve each specified effect using both annotations and the `web.xml` file. Later modules, when introducing more advanced aspects of deployment, will take the same approach.



---

**Note** – Not all configurations can be achieved using annotations.

---

## Controlling Configuration

With three elements capable of providing configuration information, `web.xml`, `web-fragment.xml`, and annotations, it is possible for these to provide conflicting information. To address this, some rules and controls are provided. Three are particularly significant, and these are:

- Precedence
- `<metadata-complete>` tag
- Ordering

Annotations will be overridden by conflicting requirements in an XML file.

The `<metadata-complete>` tag may be used to indicate that an XML file specifies everything related to the current item being configured. For example, in a `web.xml` file, this will prevent any `web-fragment.xml` files or annotations being used to configure the application. In a `web-fragment.xml` file, the tag indicates that no annotations or other `web-fragment.xml` files should be applied to the configuration of this library.

If desired, you can apply a `<name>` tag to an XML file, and using that name, specify either an `<absolute-ordering>` or an `<ordering>` tag to indicate the order in which configuration elements should be applied.



---

**Note** – These rules are only outlined here. In most situations you will not need to be concerned with this degree of detail. However, the certification exam does expect you to know these elements, and how to use them.

---

## Dynamic Configuration of Web Applications

Java EE 6 adds features that allow the programmatic configuration of servlets, filters, and related components. This feature is only available as the web application is being loaded, specifically during the time that the servlet context is being initialized. It is perhaps most useful for framework creators.

## Servlet Mapping

HTML and JSP pages that exist outside of `WEB-INF` are located using a URL that maps through the context root and down to the specific file that is being requested. However, the same is not true for servlets. To access a servlet, a mapping must be created.

A mapping is information that specifies that a particular URL, or group of URLs, refer to a particular named servlet. The name is not the class name, but is an internal reference within the web container to a particular servlet class configured in a particular way.

The mapping in the `web.xml` file therefore consists of two parts, and is illustrated in Code 5-1 below:

```
1  <web-app [...] >
2      <servlet>
3          <servlet-name>MyServletName</servlet-name>
4          <servlet-class>SL314.package.MyServlet</servlet-class>
5      </servlet>
6      <servlet-mapping>
7          <servlet-name>MyServletName</servlet-name>
8          <url-pattern>/MyServlet</url-pattern>
9      </servlet-mapping>
10 </web-app>
```

### Code 5-1 Servlet Mapping in `web.xml`

Notice how the first part, lines 2 through 5, set the name of the servlet and the class from which it is instantiated.

In the second part, lines 6 through 9, the URL that should trigger the servlet is specified.

## Multiple and Wildcard URL Patterns

It is quite common to find a single servlet being used to respond to a variety of URLs. This is supported in two ways: First, the `<url-pattern>` XML element may be repeated. Second, the pattern specified by a URL may be a wildcard, ending with an asterisk.

For example, given the mapping shown in Code 5-2 below:

```
1 <servlet-mapping>
2     <servlet-name>MyServlet</servlet-name>
3     <url-pattern>/MyServlet</url-pattern>
4     <url-pattern>/YourServlet</url-pattern>
5     <url-pattern>/HisServlet/*</url-pattern>
6 </servlet-mapping>
```

### Code 5-2 Servlet Mapping to Multiple URLs

The servlet would respond to any of these URLs:

```
http://localhost:8080/MyAppRoot/MyServlet
http://localhost:8080/MyAppRoot/YourServlet
http://localhost:8080/MyAppRoot/HisServlet
http://localhost:8080/MyAppRoot/HisServlet/one
http://localhost:8080/MyAppRoot/HisServlet/twenty
```

## Mapping Using Annotations

Since Java EE 6, servlet mappings can be achieved using annotations. A servlet should be given the annotation:

```
javax.servlet.annotation.WebServlet
```

This annotation supports various optional elements, but a common form is that which has been used in examples in previous modules:

```
@WebServlet(name="MyServletName",
urlPatterns={"/MyServlet", "/YourServlet",
"/HisServlet/*"})
```

## Invoking a Servlet by Name

A servlet can be invoked directly by its name through a `RequestDispatcher`, in a `forward` or `include` call. The necessary dispatcher is obtained using the servlet context which offers a method `getNamedDispatcher(String)`.

# Servlet Context Information

The deployment descriptor can include configuration information for the servlet. Two configuration elements of frequent utility are *context parameters* and *init parameters*.

## Context Parameters

The term “servlet context” essentially refers to the web application and the container in which the servlet runs. An interface `javax.servlet.ServletContext` provides access to several aspects of this environment, but in the context of this discussion, the methods of interest are two methods that allow read-only access to a map of context parameters. These methods are:

```
String getInitParameter(String name)
Enumeration<String> getInitParameterNames()
```

Notice that these methods refer to `InitParameter`, but because they are called on the `ServletContext`, the initialization data they return are actually referred to as context parameters.

## Supplying Context Parameters

Context Parameters cannot be supplied using annotations, because they relate to the entire web-application, rather than to a single element of it.

To specify a context parameter using a `web.xml`, or `web-fragment.xml` file, the specification is quite simple. Here is an example.

```
1 <web-fragment [...]>
2   <context-param>
3     <param-name>fragmentContext</param-name>
4     <param-value>fragment Context value</param-value>
5   </context-param>
6 </web-fragment>
```

### **Code 5-3** A `web-fragment.xml` File Specifying a Context Parameter

If this is done using a `web.xml`, then line 1 would read `<web. . .`, rather than `<web-fragment. . .`

## Reading Context Parameters

Context parameters can be read in Java code using statements of the following form:

```
String paramValue =  
    this.getServletContext()  
        .getInitParameter("paramName");
```

Notice that the servlet (`this`) gives access to an object of type `ServletContext`, which describes the configuration of the application as a whole, from which the parameter may be read.



# Servlet Initialization Parameters

In addition to configuration parameters for the entire application, initialization parameters can be associated with the servlet individually. These are typically known simply as *init parameters*. This association may be achieved using the deployment descriptor file like this:

```
1  <servlet>
2      <servlet-name>MyServlet</servlet-name>
3      <servlet-class>SL314.package.MyServlet</servlet-class>
4      <init-param>
5          <param-name>name</param-name>
6          <param-value>Fred Jones</param-value>
7      </init-param>
8      <init-param>
9  </servlet>
```

Notice that this XML fragment declares the initialization parameter called `name` with a value of `Fred Jones` for the servlet named `MyServlet`.

## Using Annotations

Because `init` parameters are associated directly with the servlet, they can be specified using the annotations mechanism. The annotation that would be equivalent to the preceeding declaration would be:

```
@WebServlet (
    name="MyServlet",
    urlPatterns=({"/MyServ"}),
    initParams =
        {@WebInitParam(name="name", value="Fred Jones")}
)
```

### Code 5-4 Annotation Declaring a Servlet Initialization Parameter

## Reading Servlet Initialization Parameters

The servlet itself provides direct access to servlet initialization parameters using the method `getInitParameter(String)`, and another method of the same signature is provided in `ServletConfig` object. So, these two lines of code, executed in the servlet, are equivalent:

```
name = this.getInitParameter("name");
```

```
name = getServletConfig().getInitParameter("name");
```

### **Code 5-5** Two Ways to Read Servlet Initialization Parameters

## Other Configuration Elements

The deployment descriptor can contain elements of many other types beyond those discussed in this module. Similarly, several other annotations exist for configuration purposes.

While these are topics for later modules, you should know that the deployment descriptor is used to configure security, and connections to other aspects of a Java EE system, such as Enterprise JavaBeans™ (EJB™), message queues, and databases.

## Summary

This module introduced these concepts:

- Deployment descriptors
- The context root of an application
- Servlet mappings
- Context and init parameters
- Configuration of servlets using annotations

## Module 6

---

# More View Facilities

---

## Objectives

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

- Understand and use the four scopes
- Understand and use the EL
- Recognize and use the EL implicit objects

## Relevance

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

- What lifetimes and accessibilities are necessary for variables in a web application?
- How can the EL be best used, and what facilities should it offer that have not been introduced yet?

## 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 Platform, Enterprise Edition 5 API Specification. Available:  
<http://java.sun.com/javaee/6/docs/api>
- Java Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- The Common Gateway Interface. [Online]. Available:  
<http://hoohoo.ncsa.uiuc.edu/cgi/>

## Scopes

You saw in several earlier modules that servlets can store and access application and parameter data in several places. To review, the locations discussed so far are:

*The request object* – Contains a map of attributes and is shared across forward calls. Its scope is bounded by a request from a single client.

*The session object* – Contains a map of attributes shared across all calls from a single browser within a single session. Session duration may be bounded by time limits or by a login/logout type process.

*The servlet's initialization parameters* – Is a map of read-only configuration data available to this servlet from the deployment descriptor.

*The application's initialization parameters* – Is a map of read-only configuration data from the deployment descriptor and available to any servlet in the application.

*The browser's cookies* – Is a map of cookie values set in the browser by the application.

There is another place that data can be stored for sharing between different parts of the application. The servlet context object itself provides a read-write map of attributes that are shared throughout the entire application.

Three of these locations are referred to as scopes. These are the request scope, the session scope, and the application scope. The application scope is the name given to the map in the servlet context.

Notice that these scopes all provide useful communication between different parts of the application.

For variables within the execution of the servlet's `doXXX` methods, you can simply use local variables. However, for some techniques, variables equivalent to method locals are required within the JSP itself. These are provided using a fourth scope called page scope.



---

**Note** – Using variables in page scope suggests that the separation of concerns represented by MVC has not been implemented cleanly. Such approaches are often found in older JSP programs.

---



The four scopes may then be described as follows:

**Table 6-1** EL Implicit Objects

Scope Name	Communication
page	Between local variables within a JSP only. Equivalent to local variables in a <code>doXXX</code> servlet method.
request	Between components cooperating in the execution of a single request from a single browser. Typically used to carry data from a controller servlet to the view JSP.
session	Between servlets and JSPs used during a single user session, across the different requests being made.
application	Between all components of a single application.

## More Details About the Expression Language (EL)

EL was introduced in “Using Data in the JSP” on page 3-6 in Module 3, “ModTitle”. As you might expect, EL is considerably more powerful than was indicated at that point.

### Syntax Overview

The syntax of EL in a JSP page is as follows:

```
${expr}
```

In the previous syntax, `expr` indicates a valid EL expression. This expression can be mixed with static text and can be combined with other expressions to form larger expressions.



**Note** – If you need to present the special characters in a page literally, you can escape EL characters on a JSP page using the backslash (\) character like this: `\${expr}`

When escaped, the EL expression is not evaluated.

### EL and Scopes

In “Using Data in the JSP” on page 3-6 in Module 3, “ModTitle” The use of the expression language was introduced showing how it could read values from a bean in the request scope.

In fact, EL scans all four scopes looking for a bean with the name given in the expression. Clearly if two beans with the same name exist in different scopes, confusion might arise. However, the scopes are scanned in a fixed order: page, request, session, and application.

You can override the search order and explicitly specify the scope containing your bean. To do this, the bean name is preceded by an implicit scope object. The following example retrieves the `firstName` property from the bean `cust` located in the session scope:

```
${sessionScope.cust.firstName}
```

## EL Implicit Objects

The variables used to explicitly name a scope containing a bean are called *implicit objects*. Implicit objects are pre-defined variable names known to EL that provide access to bean-like data. Several implicit objects exist beyond those used for scope access. All but one of these (the `pageContext` object) are maps. Table 6-2 shows the implicit objects available with EL.

**Table 6-2** EL Implicit Objects

Implicit Object	Description
<code>pageContext</code>	The <code>PageContext</code> object
<code>pageScope</code>	A map containing page-scoped attributes and their values
<code>requestScope</code>	A map containing request-scoped attributes and their values
<code>sessionScope</code>	A map containing session-scoped attributes and their values
<code>applicationScope</code>	A map containing application-scoped attributes and their values
<code>param</code>	A map containing request parameters and single string values
<code>paramValues</code>	A map containing request parameters and their corresponding string arrays
<code>header</code>	A map containing header names and single string values
<code>headerValues</code>	A map containing header names and their corresponding string arrays
<code>cookie</code>	A map containing cookie names and their values
<code>initParam</code>	A map of the servlet's init parameters

Using these implicit objects, for example, a page can access a request parameter called `username` using the following EL expression:

```
${param.username}
```



---

**Note** – Recall that accessing data from places other than the model might be a weak design approach.

---

## The Dot Operator In EL

It is probably obvious by now that the EL syntax allows cascading of the dot operator to select elements from within elements. This is exactly equivalent to calling methods on the return values of methods in Java itself. So, if the request scope contains an object with the attribute name `pet` which is a `Dog` JavaBean, and that `Dog` bean has a `getOwner()` method, and that method returns a `Human` object, which in turn has a `getName()` method that returns a `String`, then the following EL syntax would be legitimate and would access the name of the dog's owner:

```
${requestScope.pet.owner.name}
```

This syntax can be used to extract data from either a Java Bean or from a map. The EL engine will determine which type of object it is accessing, and behave appropriately without intervention by the programmer.

The name elements used with the dot operator must follow the naming conventions for Java identifiers. This means that some legitimate keys that might be used to identify elements stored in a map cannot be used with the dot operator—consider what would happen if the key used to store a data item in a map contained periods. The EL engine would treat those periods as dot operators, and the lookup will fail.

## Array Access Syntax With EL

If a bean returns an array, an element in the array can be accessed by providing its index:

```
${paramValues.fruit[2]}
```

As with Java, the dot operator may be used to access elements of an object after it has been selected from an array. So, if the `Dog` bean mentioned above happened to return an array of two `Human` objects from a method `getOwnerFamilyMembers()` then this would display the name of the first owner:

```
${requestScope.pet.ownerFamilyMembers[0].name}
```

In addition to accessing elements of arrays in this way, the EL array syntax notation behaves as an associative array. So, an attribute name may be used as the array subscript if preferred:

```
${requestScope["pet"].owner["name"]}
```

In most situations, the two forms are interchangeable. However, this second form can be used in a number of situations not supported by the dot operator. These are:

- Access to map elements with keys that do not conform to the naming rules, such as those containing periods
- Access to arrays, by numeric subscript, or a String containing parsable numbers, for example:  

```
${pet.ownerFamilyMembers["2.0"]}
```
- Elements of a List, accessed by index number as with an array

The array access form also allows access to a computed element. For example if the request scope contains an attribute subscript with the value 2, then this expression:

```
${pet.ownerFamilyMembers[subscript]}
```

would have the same effect as the preceeding example, selecting the third element of the array returned by the pet's `getOwnerFamilyMembers()` method.




---

**Note** – Array access must use the array access form. You cannot put a literal numeric subscript after a dot operator.

---

Avoid classes that are both Collections and JavaBeans. If this occurs, Java Beans methods will be ignored in favor of the map, even if the map does not contain the requested element.

## EL and Errors

EL silently hides errors that arise from null references or not-found variables. That is, such expressions translate to empty strings. This is deliberate, as it ensures that errors do not mess up the pages that the user sees. Of course, the information on the page might be incomplete.



**Note** – If an EL expression refers to a non-existent attribute, such as `${exists.notFound}` then EL will cause an exception, and an unhelpful, and potentially insecure, stack trace will be shown to the user.

## EL Arithmetic Operators

Table 6-3 shows the five arithmetic operators that are available in EL.

**Table 6-3** EL Arithmetic Operators

Arithmetic Operation	Operator
Addition	+
Subtraction	-
Multiplication	*
Division	/ and div
Remainder	% and mod

The arithmetic operators are designed to operate on integer (`BigInteger` and `Long`) and floating point (`BigDecimal` and `Double`) values. Table 6-4 shows some example operations.

**Table 6-4** Some Example EL Arithmetic Operations

EL Expression	Result
<code>\${3 div 4}</code>	0.75
<code>\${1 + 2 * 4}</code>	9
<code>\${(1 + 2) * 4}</code>	12
<code>\${32 mod 10}</code>	2

The EL engine attempts to perform data conversion on terms when necessary. For example, given the expression:

`${param.miles * 1.609}`

The request parameter `miles` will be retrieved and the EL engine will attempt to convert it into a number. If the `miles` parameter does not exist in the request, the `null` value will be converted to 0, and the expression will evaluate to 0. If the `miles` parameter exists, the string is converted to a number which is then multiplied by 1.609. If the string cannot be converted (if it is character data), then an exception occurs.

## Comparisons and Logical Operators

Table 6-5 presents the six comparison operators available in EL.

**Table 6-5** EL Comparison Operators

Comparison	Operator
Equals	<code>==</code> and <code>eq</code>
Not equals	<code>!=</code> and <code>ne</code>
Less than	<code>&lt;</code> and <code>lt</code>
Greater than	<code>&gt;</code> and <code>gt</code>
Less than or equal	<code>&lt;=</code> and <code>le</code>
Greater than or equal	<code>&gt;=</code> and <code>ge</code>

Comparison operations return a boolean: `true` or `false`. If used within template text, the strings `true` or `false` actually appear in the output. Typically, these expressions are used in EL expressions that are used as values to attributes of custom tags or actions.

Table 6-6 lists useful unary operators in EL.

**Table 6-6** EL Logical Operators

Operation	Operator
and	<code>&amp;&amp;</code> and <code>and</code>
or	<code>  </code> and <code>or</code>
not	<code>!</code> and <code>not</code>
Test if an array or list is empty	<code>empty</code>

## More Details About the Expression Language (EL)

---

The logical operations for and, or, and not, behave according to the usual conventions for these operations. The unary test operator `empty` allows determination of whether a list or array has zero elements within it. This can be particularly valuable in presentation programming, which will be discussed later in this module.



## Configuring the JSP Environment

This section outlines the deployment descriptor configuration for the JSP environment. You can modify the behavior of JSP pages in the application using the `jsp-config` tag in the `web.xml` deployment descriptor.

Code 6-1 shows the JSP environment configuration.

```
13 <jsp-config>
14   <jsp-property-group>
15     <url-pattern>/scripting_off/*</url-pattern>
16     <scripting-invalid>true</scripting-invalid>
17   </jsp-property-group>
18
19   <jsp-property-group>
20     <url-pattern>/EL_off/*</url-pattern>
21     <el-ignored>true</el-ignored>
22   </jsp-property-group>
23 </jsp-config>
24
25 </web-app>
```

### Code 6-1 JSP Environment Configuration

Within the `jsp-config` tag, you can declare zero or more `jsp-property-group` elements. Within a `jsp-property-group`, you declare a `url-pattern` that specifies the files that belong to this group.

On line 6, the `scripting-invalid` tag is used to turn off scripting within the JSP pages in the group. If set to `true`, the JSP pages in the group are prohibited from containing scripting code. If a JSP page contains scripting, an error will result.

Shown on line 21, the `el-ignored` tag is used to disable EL in the JSP pages in the group. By default, `el-ignored` has the value `false`, indicating that EL is executed. If set to `true`, any EL on the JSP pages in the group is ignored. This may be useful when pages are used from an older project and those pages use EL-like text that is intended for literal presentation.

## Presentation Programming

In “Task – Create the Controller Servlet” on page Lab 4-2 you created HTML inside a servlet. It was noted that this was very poor design, but there was a problem that at the time you were not equipped to solve. Specifically, the proper approach required creating a variable number of `<li>` elements depending on the number of HTTP headers found in the request. A similar problem arises if the presentation needs to contain conditional elements. In effect, presentation might have to solve issues of iteration or selection, in much the same way as a regular programming language.

This kind of presentation-specific programming does properly belong in the view, not in either the model or the controller, but the presentation issue cannot be solved unless some form of programming facility is available. To this point in this course, design efforts have focussed on keeping “coding” out of the view. In particular, we’ve been trying very hard to avoid putting Java code in the view.

A good solution to this particular problem is provided by an extensive toolkit called the Java Standard Tags Library (JSTL). This library provides many features that will be presented in a later module, but for now, the specific features that provide for iteration and selection will be introduced.

## Standard Custom Tags

When the tag library mechanism was introduced, tags were called “custom tags,” as each developer was expected to create their own. However, a popular set of tags was rapidly adopted as the “standard tags” (or JSTL). For this reason, you’ll often here tags referred to as both custom, and standard, sometimes in the same sentence. Strictly speaking, there is now a set of standard tags, and others are custom tags. Both use the same custom tags mechanism for their implementation and execution.

## Tag Example

Here is an example of a custom tag. Line 1 is the declaration of the tag library. This specifies what library is to be used, and the namespace prefix (conventionally “c” for the core library) that will be used. Line 4 tests to see if the array referred to in EL by `${errorMsgs}` is empty, and if not, allows the rest of the error code to be expanded. On lines 8 and 10, the “forEach” tag is used to iterate over the elements of that array of errors, creating an `<li>` element for each one.

```
1  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2  [... More of the page]
3  [...]
4  <c:if test="${not empty errorMsgs}">
5      <p>
6          <font color='red'>Please correct the following errors:
7          <ul>
8              <c:forEach var="message" items="${errorMsgs}">
9                  <li>${message}</li>
10             </c:forEach>
11         </ul>
12     </font>
13 </p>
14 </c:if>
```

## Developing JSP Pages Using Custom Tags

Follow these guidelines when creating JSP pages using an existing tag library:

- Use the tag namespace prefix consistently.
- Code the custom tags carefully; the syntax follows XML rules, and nesting must be correct.
- Take care with case; the tags are entirely case sensitive.
- Declare the tag library in the JSP page and in the web application deployment descriptor.

## Key View Programming Tags

Two tags in particular provide great value for view programming. Good design suggests you should take care to use them to perform the programmatic operations that relate specifically to the presentation. The tags presented in this section are the `if` and `forEach` tags shown in the example above. More tags, and more supporting details about them and their use, are presented in a later module.

### JSTL `if` Tag

The `if` tag is a conditional tag in the core library. You use this tag to conditionally evaluate what is contained in the tag's body or to set a variable based on the evaluation of a test.

When a body is supplied, the body is only evaluated if the test expression is true:

```
<c:if test="expression" var="varName"
    [scope="{page|request|session|application}"] >
    body evaluated if expression evaluates to true
</c:if>
```

The `var` attribute specifies the variable name into which the results of the test (true or false) will be stored. The `scope` attribute specifies where the variable will be stored.

Code 6-2 below demonstrates how the `if` tag can be used in a JSP page.

```
54 <%-- Report any errors (if any) --%>
55 <c:if test="${not empty errorMsgs}">
56     <p>
57         <font color='red'>Please correct the following errors:
58         <ul>
59             <c:forEach var="message" items="${errorMsgs}">
60                 <li>${message}</li>
61             </c:forEach>
62         </ul>
63         </font>
64     </p>
65 </c:if>
```

#### Code 6-2 The `if` Tag

The code from lines 42 to 50 is only evaluated if the expression for the `test` attribute on line 41 evaluates to `true`.

## JSTL `forEach` Tag

The `forEach` tag provides a mechanism for iteration over the body of the tag within a JSP page. The attributes of the tag are `items`, `var`, `varStatus`, `begin`, `end`, and `step`. You use this tag to iterate over an existing collection or for a specified number of iterations.

To iterate over a collection, the `items` attribute is used to supply the name of a collection.

```
<c:forEach items="collection" [var="varName"]  
           [varStatus="varStatusName"]  
           [begin="begin"] [end="end"] [step="step"]>  
    body content  
</c:forEach>
```

The collection supplied to the `items` attribute can be an implementation of `java.util.Collection`, `java.util.Iterator`, `java.util.Map`, `java.util.Enumeration`, an array, or a comma-delimited string.

Table 6-7 demonstrates the remaining `forEach` tag attributes and their corresponding uses.

**Table 6-7** Remaining `forEach` Tag Attributes and Uses

Attribute	Use
<code>var</code>	Specifies the name of the nested-visibility variable, which will contain the current element in the iteration.
<code>varStatus</code>	Specifies the name of the nested-visibility variable, which will contain the status of the current cycle in the iteration. <code>varStatus</code> is itself a structured type and has elements <code>index</code> and <code>count</code> , that track the progress through the loop
<code>begin</code>	Specifies the index of the first item in the iteration.
<code>end</code>	Specifies the last item in the iteration.
<code>step</code>	Skips over elements in the iteration.

## Key View Programming Tags

Code 6-3 shows how the `forEach` tag can be used in a JSP page to iterate over a collection.

```

1    <font color='red'>Please correct the following errors:
2    <ul>
3    <c:forEach var="message" items="${errorMsgs}">
4        <li>${message}</li>
5    </c:forEach>
6    </ul>
7    </font>

```

### Code 6-3 The `forEach` Tag

The attribute `items` on line 3 has as its value an EL expression, which evaluates to a collection. Each item in the iteration will be stored in a variable called `message` which will be available in the body of the `forEach` tag. Within the body, an EL expression is used to display the contents of the `message` variable.

If the `items` attribute is not specified, then the `begin` and `end` attributes must be provided.

When used with this syntax, the tag performs similar to the Java technology `for` loop:

```

<c:forEach [var="varName"] [varStatus="varStatusName"]
           begin="begin" end="end" [step="step"]>
    body content
</c:forEach>

```

For example, if you wanted to iterate over a section of template text ten times, you could use the `forEach` tag:

```

<c:forEach begin="1" end="10">
    <!-- the repeated HTML code here -->

</c:forEach>

```

# Summary

This module introduced the following concepts:

- Four variable scopes
- More EL features
- EL implicit objects
- JSTL `if` and `forEach` tags





## Module 7

---

# Developing JSP Pages

---

## Objectives

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

- Describe JSP page technology
- Write JSP code using scripting elements
- Write JSP code using the page directive
- Write JSP code using standard tags
- Write JSP code using the EL
- Configure the JSP page environment in the `web.xml` file

## 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:

- Sun Microsystems, Inc., “JSR-000245 JavaServer Pages™ 2.1,”  
[<http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html>], accessed 02 December 2006.
- Sun Microsystems, Inc., “Java EE 5 Tutorial, Chapter 4: JavaServer Pages Technology,”  
[<http://java.sun.com/javaee/5/docs/tutorial/doc/JSPIntro.html>], accessed 02 December 2006.
- Hall, Marty. *Core Servlets and JavaServer Pages*. Upper Saddle River: Prentice Hall PTR, 2000.

## JavaServer Pages Technology

To this point, this course has focused on building web applications based on the model-view-controller separation of concerns. Considerable thought has gone into providing technologies that avoid mixing Java computational code with HTML-like presentation code. However, this separation of concerns was not present in the first form of JSPs. Indeed, in their original form, JSPs served as a means of embedding Java code into an HTML page, rather than of embedding HTML code in a Java method. The benefit was simply that the resulting source file could be edited in a WYSIWYG HTML editor.

Of course, it is inevitable that many such pages still remain, even though this approach is considered less than elegant today. This module introduces the tools and techniques of those original JSPs, so that you will be equipped to work on such pages should you be called upon to do so.

Despite the warning about careful separation of concerns, it must be acknowledged that sometimes, for example during prototyping, it might serve your needs better to hack up a quick-and-dirty solution to even a new problem, so in that sense, these tools might still have merit for an initial system. Provided you are familiar with them, you will have the choice of what to do, and when.

This section compares a servlet with an equivalent JSP. This servlet retrieves the name of the user, which becomes part of the dynamic response. Code 7-1 shows the Hello World servlet code.

```
1  public class HelloServlet extends HttpServlet {
2
3      private static final String DEFAULT_NAME = "World";
4
5      public void doGet(HttpServletRequest request,
6                          HttpServletResponse response)
7          throws IOException {
8          generateResponse(request, response);
9      }
10
11     public void doPost(HttpServletRequest request,
12                         HttpServletResponse response)
13         throws IOException {
14         generateResponse(request, response);
15     }
16
17     public void generateResponse(HttpServletRequest request,
18                                 HttpServletResponse response)
19         throws IOException {
20
21         String name = request.getParameter("name");
22         if ( (name == null) || (name.length() == 0) ) {
23             name = DEFAULT_NAME;
24         }
25
26         response.setContentType("text/html");
27         PrintWriter out = response.getWriter();
28
29         out.println("<HTML>");
30         out.println("<HEAD>");
31         out.println("<TITLE>Hello Servlet</TITLE>");
32         out.println("</HEAD>");
33         out.println("<BODY BGCOLOR='white'>");
34         out.println("<B>Hello, " + name + "</B>");
35         out.println("</BODY>");
36         out.println("</HTML>");
37
38         out.close();
39     }
40 }
```

**Code 7-1** Hello World Servlet

Code 7-2 shows the equivalent JSP page. Most of the JSP page is HTML template text. The `<%` and `%>` tags on lines 10 and 15 instruct the web container to execute the embedded Java technology code at runtime. Similarly, the `<%= name %>` element on Line 19 instructs the web container to place the string value of the name variable into the HTTP response at runtime.

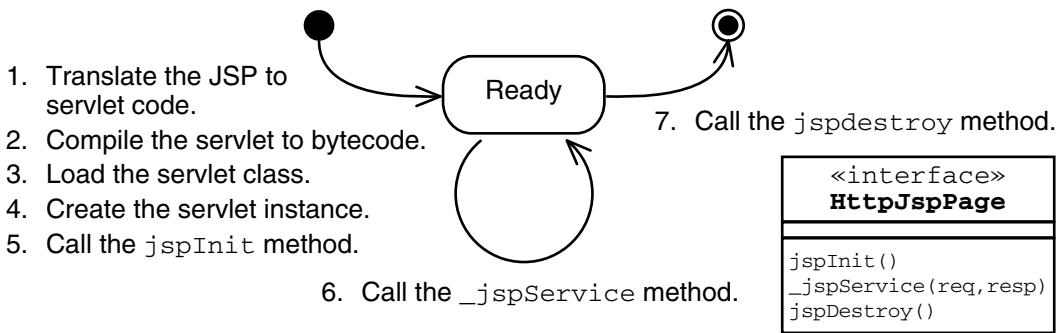
```
1  <%! private static final String DEFAULT_NAME = "World"; %>
2
3  <html>
4
5  <head>
6  <title>Hello JavaServer Page</title>
7  </head>
8
9  <%-- Determine the specified name (or use default) --%>
10 <%
11     String name = request.getParameter("name");
12     if ( (name == null) || (name.length() == 0) ) {
13         name = DEFAULT_NAME;
14     }
15 %>
16
17 <body bgcolor='white'>
18
19 <b>Hello, <%= name %></b>
20
21 </body>
22
23 </html>
```

**Code 7-2** The `hello.jsp` Page

This module describes JSP scripting elements and how the web container processes a JSP page.

# How a JSP Page Is Processed

A JSP page is essentially source code, and must be converted into a servlet before it can service requests. Figure 7-1 shows the steps of JSP page processing.

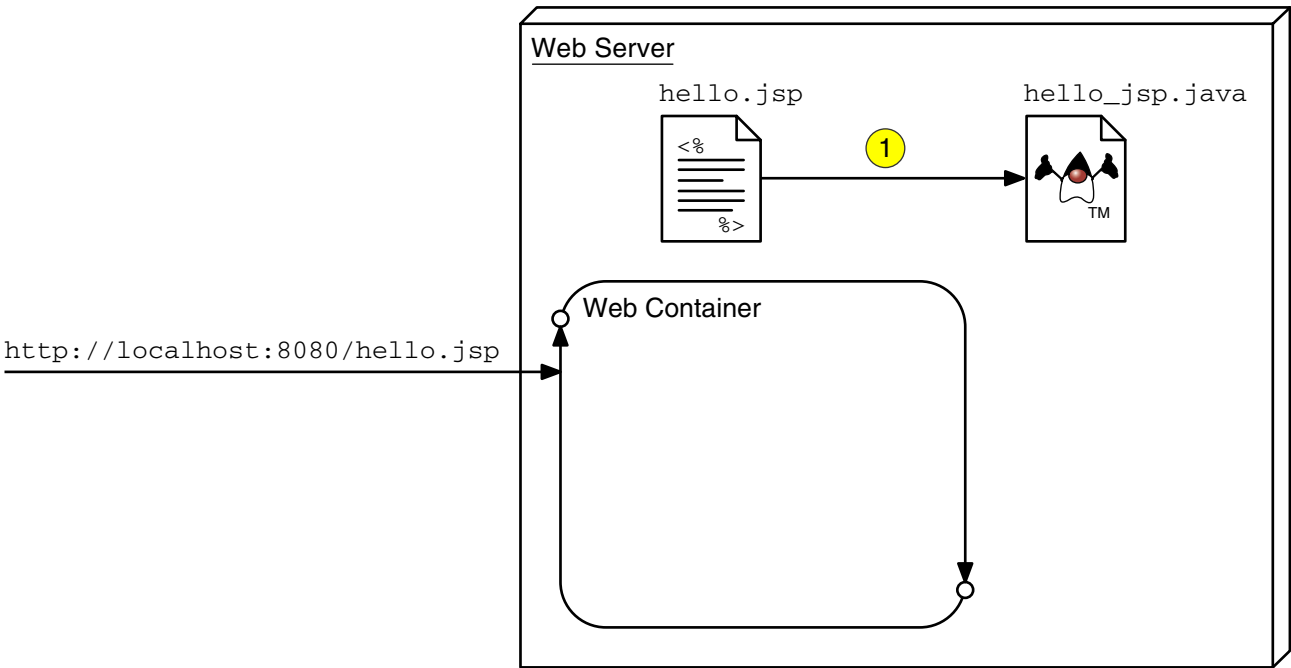


**Figure 7-1** Steps of JSP Page Processing

## JSP Page Translation

In the first step, the web container translates the JSP file into a Java source file that contains a servlet class definition.

Figure 7-2 shows JSP page translation.

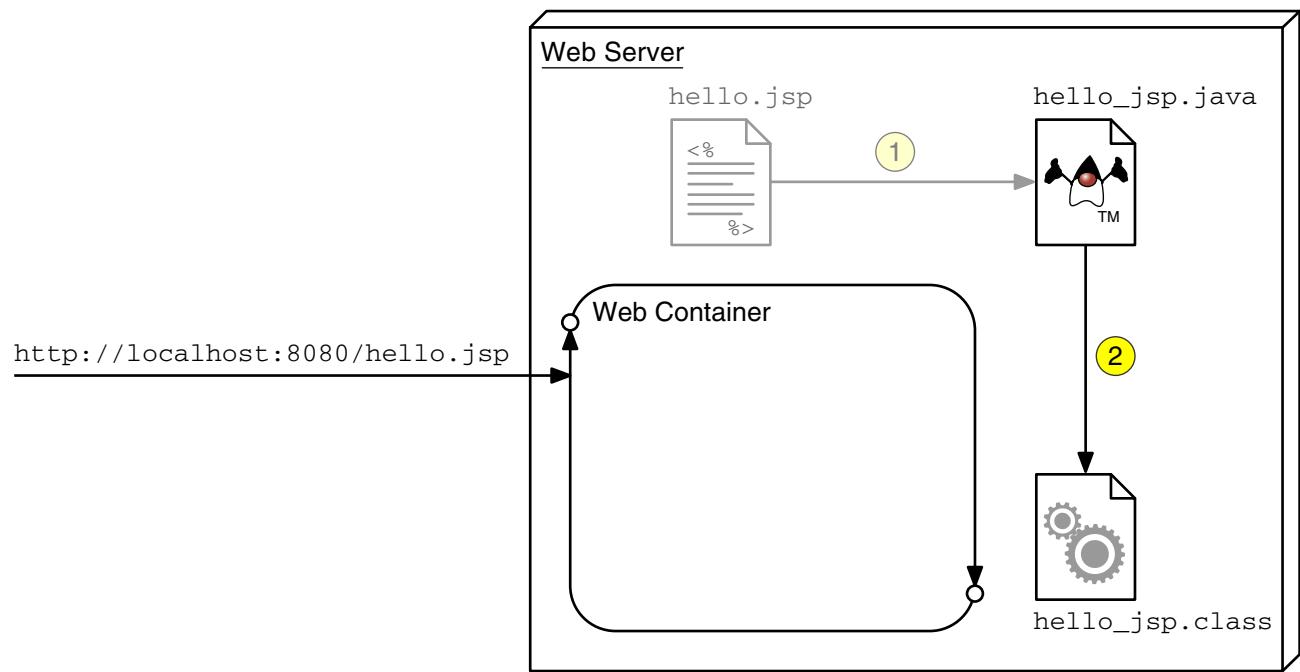


**Figure 7-2** JSP Page Processing: Translation Step

### JSP Page Compilation

In the second step, the web container compiles the servlet source code into a Java class file.

Figure 7-3 shows JSP page compilation.



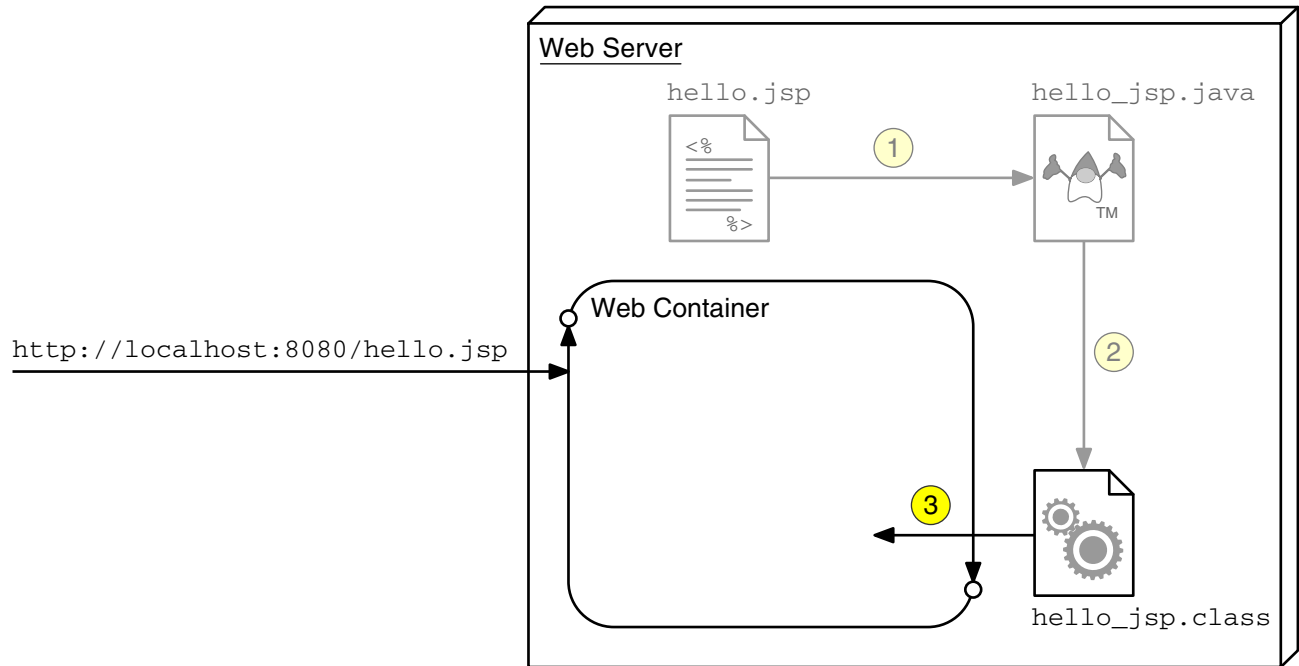
**Figure 7-3** JSP Page Processing: Compilation Step



## JSP Page Class Loading

In the third step, the servlet class bytecode is loaded into the web container's JVM software using a classloader.

Figure 7-4 shows JSP page class loading.

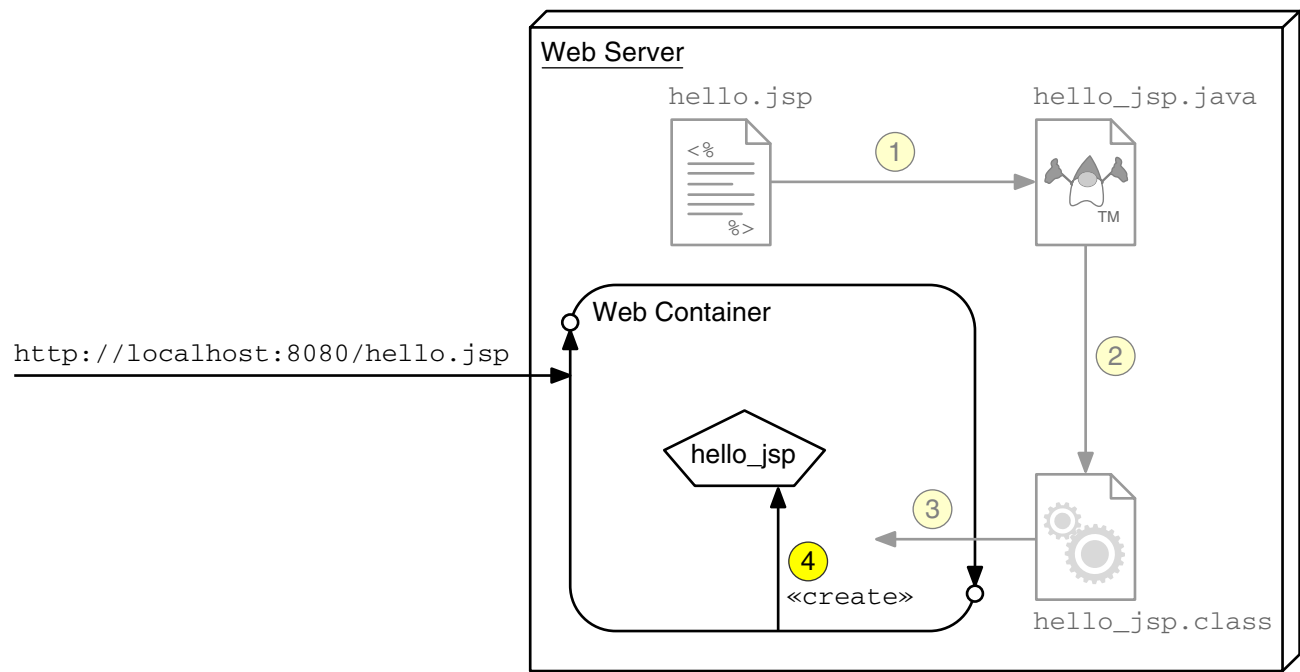


**Figure 7-4** JSP Page Processing: Load Class Step

### JSP Page Servlet Instance

In the fourth step, the web container creates an instance of the servlet class.

Figure 7-5 shows JSP page servlet instance creation.

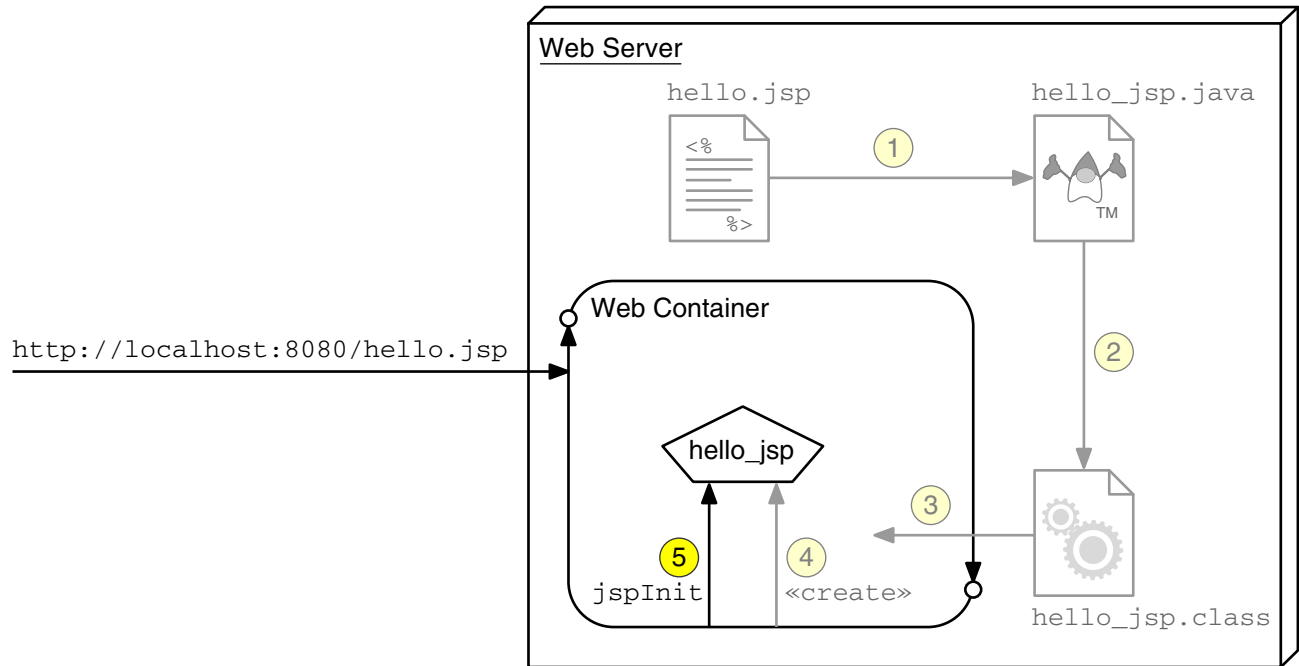


**Figure 7-5** JSP Page Processing: Instance Creation Step

## JSP Page Initialization

In the fifth step, the web container initializes the servlet by calling the `jspInit` method.

Figure 7-6 shows JSP page initialization.



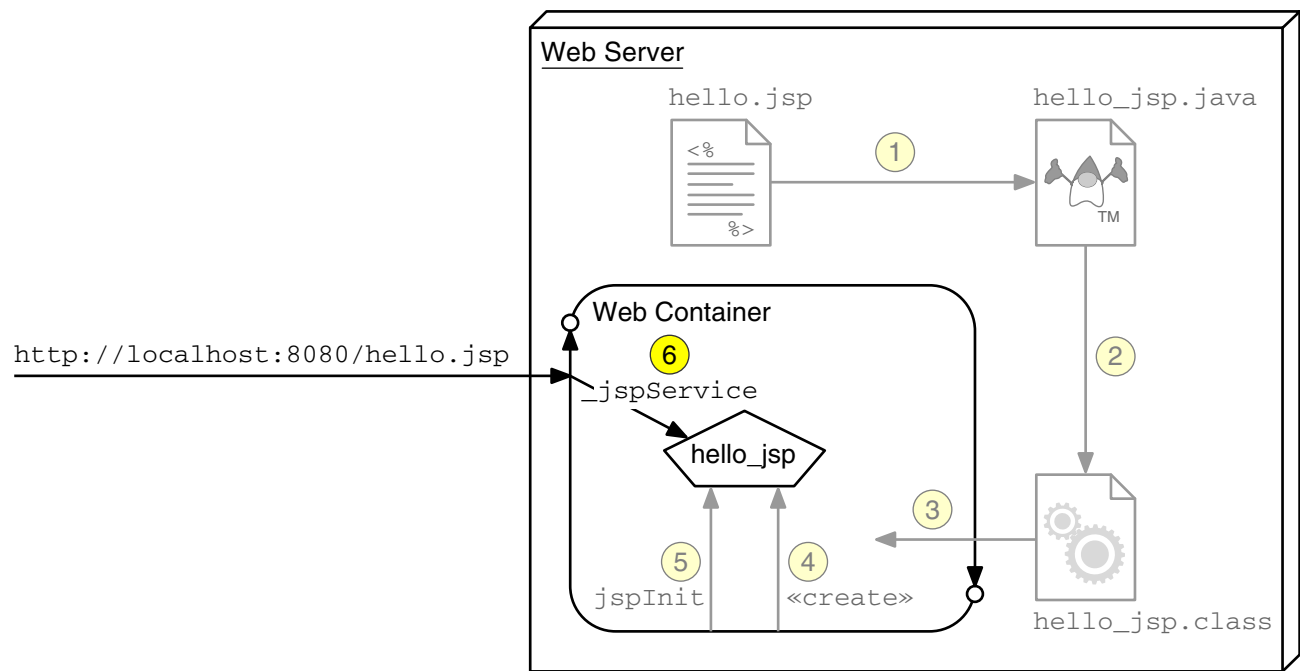
**Figure 7-6** JSP Page Processing: Initialization Step

Actually, the container calls the `init(SC)` method, but your JSP page servlet inherits an `init` method that forwards the call to the `jspInit` method.

### JSP Page Service

The initialized servlet can now service requests. With each request, the web container can call the `_jspService` method for the converted JSP page.

Figure 7-7 shows the JSP page request handling.

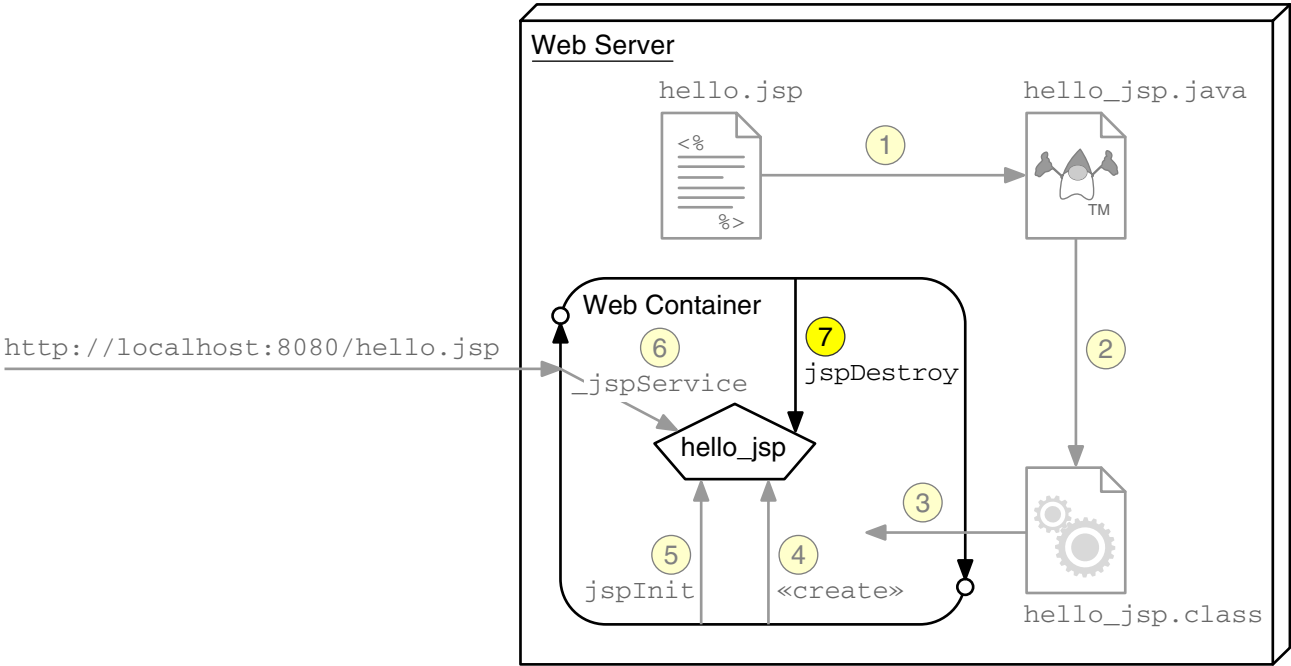


**Figure 7-7** JSP Page Processing: Request Handling Step

# JSP Page Destroyed

When the web container removes the JSP servlet instance from service, it first calls the `jspDestroy` method to allow the JSP page to perform any required clean up.

Figure 7-8 shows the JSP page destruction.



**Figure 7-8** JSP Page Processing: Destroy Step



**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.

## Writing 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>
```

Table 7-1 lists the five types of scripting elements.

**Table 7-1** Scripting Elements

Scripting Element	Example
Comment	<code>&lt;%-- comment --%&gt;</code>
Directive	<code>&lt;%@ directive %&gt;</code>
Declaration	<code>&lt;%! declaration %&gt;</code>
Scriptlet	<code>&lt;% code %&gt;</code>
Expression	<code>&lt;%= expression %&gt;</code>



**Note** – Scripting is discouraged in JSP pages. By using standard tags, custom tags, and EL elements, you can eliminate virtually all scripting in a JSP page. This can help make your JSP pages modular, reusable, and maintainable.

## 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 technology comments

Java technology 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 the Javadoc™ tool comments in declaration tags. For example, `/** @author Jane Doe */`.

---

## Directive Tag

A directive tag provides information that affects the overall translation of the JSP page. The syntax for a directive tag is as follows:

```
<%@ DirectiveName [attr="value"] * %>
```

Three types of directives are used in JSP pages: page, include, and taglib. The following are some examples:

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

```
<%@ include file="incl/copyright.html" %>
```

The page and include directives are described later in this module. The taglib directive is described in Module 8, "Developing JSP Pages Using Custom Tags."

## Declaration Tag

A declaration tag lets you include members in the JSP servlet class, either attributes or methods. The syntax for a declaration tag is as follows:

```
<%! JavaClassDeclaration %>
```

The following are some examples of the declaration tag:

```
<%! public static final String DEFAULT_NAME = "World"; %>
```

```
<%! public String getName(HttpServletRequest request) {  
    return request.getParameter("name");  
}  
%>
```

```
<%! int counter = 0; %>
```

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 such as instance variables, instance methods, class (or static) variables, class methods, and inner classes. Be careful with using instance and class variables because of concurrency issues. A JSP page is multithreaded similar to a servlet.



You use a declaration tag to override the `jspInit` and `jspDestroy` 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** – You are not permitted to override the `_jspService` method that is created by the JSP engine during translation.

---

## Scriptlet Tag

A scriptlet tag lets the JSP page developer include arbitrary Java technology code in the `_jspService` method. The syntax for a scriptlet tag is as follows:

```
<% JavaCode %>
```

The following is an examples of a scriptlet:

```
<% 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 ) {
```

## Writing JSP Scripting Elements

---

```
%>
i is a big number.
<%
    } else {
%>
i is a small number
<%
    }
%>
```

The following 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 an expression tag is as follows:

```
<%= 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** – An expression tag can hold any Java language expression that can be used as an argument to the `System.out.print` method.

---

The following are some examples:

The following 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.

```
<B>Ten is <%= (2 * 5) %></B>
```

The following 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.

```
Thank you, <I><%= name %></I>, for registering for the soccer league.
```

The following 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, and that string is included in the HTTP response stream.

```
The current day and time is: <%= new java.util.Date() %>
```

## Implicit Variables

The JSP engine gives you access to the following implicit variables, also called *implicit objects*, in scriptlet and expression tags. These variables represent commonly used objects for servlets that JSP page developers might need to use. For example, you can retrieve HTML form parameter data by using the request variable, which represents the `HttpServletRequest` object. Table 7-2 shows all implicit variables.

**Table 7-2** JSP Pages Implicit Variables

Variable Name	Description
request	The <code>HttpServletRequest</code> object associated with the request.
response	The <code>HttpServletResponse</code> object associated with the response that is sent back to the browser.
out	The <code>JspWriter</code> object associated with the output stream of the response.
session	The <code>HttpSession</code> object associated with the session for the given user of the request. This variable does not exist if the JSP is not participating in sessions.
application	The <code>ServletContext</code> object for the web application.
config	The <code>ServletConfig</code> object associated with the servlet for this JSP page.
pageContext	The <code>pageContext</code> object that encapsulates the environment of a single request for this JSP page.
page	The <code>page</code> variable is equivalent to the <code>this</code> variable in the Java programming language.
exception	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.

## Using the page Directive

You use the page directive 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. This applies to all attributes except for the `import` attribute. You can place a page directive anywhere in the JSP file but it is a good practice to place the page directive at the beginning.

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 the JSP page.



**Caution** – Do *not* use the `extends` attribute. Changing the superclass of your JSP pages might make your web application non-portable.

- The `buffer` attribute defines the size of the buffer used in the output stream (a `JspWriter` object). The value is either `none` or `Mkb`. 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 `session` attribute defines whether the JSP page is participating in an HTTP session. The value is either `true` (the default) or `false`.
- 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 `info` attribute defines an informational string about the JSP page.

## Using the `page` Directive

---

- 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.
- The `isELIgnored` attribute specifies whether EL elements are ignored on the page. The value is either `true` or `false` (default). If set to `true`, EL on the page is not evaluated.
- 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 `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).

## Including JSP Page Segments

There are two standard approaches to including presentation segments in your JSP pages:

- The `include` directive
- The `jsp:include` standard action

### Using the `include` Directive

The `include` directive lets you include a segment in the text of the main JSP page at translation time. The syntax of this JSP technology directive is as follows:

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

Code 7-3 shows an example use of the `include` directive.

```
99      <!-- START of copyright notice -->
100      <td align='right' width='480'>
101          <%@ include file="/WEB-INF/view/common/copyright.jsp" %>
102      </td>
103      <!-- END of copyright notice -->
```

#### **Code 7-3** The `include` Directive Example

The content of the segment is embedded into the text of the main JSP page while the main page is being translated into a servlet class. If the included segment also includes other segments, then all of the segments are embedded recursively until all of the `include` directives are eliminated.

The URL in the `file` attribute might be a path that is relative to the directory position of the original request URL. It also might be an absolute path, denoted by a leading slash (/) character in the URL, that is relative to the web application's context root. Because presentation segment files are often stored in a hidden directory within the `WEB-INF` directory of the web application, it is a good practice to use the absolute URL notation.

## Using Standard Tags

The JSP specification provides standard tags for use within your JSP pages. Because these tags are required in the specification, every JSP container must support them. This ensures that any applications using the standard tags will work in any compliant JSP container.

Standard tags begin with the `jsp:` prefix. You use these tags to reduce or eliminate scriptlet code within your JSP page. However, the standard tags only provide limited functionality. Using the JSTL and EL reduces the need for the standard tags.

### JavaBeans Components

A JavaBeans component is a Java technology class with at minimum 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.
- The class implements the `java.io.Serializable` interface.



Code 7-4 shows a JavaBeans component called `CustomerBean` with three attributes: `name`, `email`, and `phone`.

```
1  package sl314.beans;
2
3  import java.io.Serializable;
4
5  public class CustomerBean implements Serializable {
6
7      private String name;
8      private String email;
9      private String phone;
10
11     public CustomerBean() {
12         this.name = "";
13         this.email = "";
14         this.phone = "";
15     }
16
17     public void setName(String name) {
18         this.name = name;
19     }
20     public String getName() {
21         return name;
22     }
23
24     public void setEmail(String email) {
25         this.email = email;
26     }
27     public String getEmail() {
28         return email;
29     }
30
31     public void setPhone(String phone) {
32         this.phone = phone;
33     }
34     public String getPhone() {
35         return phone;
36     }
37 }
```

**Code 7-4** The `CustomerBean` Component

## The useBean Tag

If you want to interact with a JavaBeans component using the standard tags in a JSP page, you must first declare the bean. You do this by using the useBean standard tag.

The syntax of the useBean tag is as follows:

```
<jsp:useBean id="beanName"
             scope="page | request | session | application"
             class="className" />
```

The `id` attribute specifies the attribute name of the bean. Where the bean is stored is specified by the `scope` attribute. Scope defaults to `page` if not specified. The `class` attribute specifies the fully qualified classname.

Given a useBean declaration of the following:

```
<jsp:useBean id="myBean" scope="request"
             class="sl314.beans.CustomerBean" />
```

the equivalent Java code might look as follows:

```
CustomerBean myBean =
    (CustomerBean) request.getAttribute("myBean");
if( myBean == null ) {
    myBean = new CustomerBean();
    request.setAttribute("myBean", myBean);
}
```

Code 7-5 shows the use of the useBean tag with a body. If the tag is used with a body, the content of the body is only executed if the bean is created. If the bean already exists in the named scope, the body is skipped.

```
38 <jsp:useBean id="cust" scope="request"
39     class="sl314.beans.CustomerBean">
40     <%
41         cust.setName(request.getParameter("name"));
42         cust.setEmail(request.getParameter("email"));
43         cust.setPhone(request.getParameter("phone"));
44     %>
45 </jsp:useBean>
```

### Code 7-5 JSP Page Using the useBean Tag

Given the JSP page seen in Code 7-5, the equivalent Java technology code would look similar to the following:

```
CustomerBean cust
    = (CustomerBean) request.getAttribute("cust");
if ( cust == null ) {
    cust = new CustomerBean();
    cust.setName(request.getParameter("name"));
    cust.setEmail(request.getParameter("email"));
    cust.setPhone(request.getParameter("phone"));
    request.setAttribute("cust", cust);
}
```

## The setProperty Tag

You use the `setProperty` tag to store data in the JavaBeans instance. The syntax of the `setProperty` tag is as follows:

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

The `name` attribute specifies the name of the JavaBeans instance. This must match the `id` attribute used in the `useBean` tag. The `property_expression` can look like:

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

The `property` attribute specifies the property within the bean that will be set. For example, to set the `email` property in the customer bean, you can use the following:

```
<jsp:setProperty name="cust" property="email" />
```

This action retrieves the value of the request parameter `email` and uses this value in the `set` method of the bean. The Java technology code equivalent would look similar to the following:

```
cust.setEmail(request.getParameter("email"));
```

The `param` attribute can be supplied if the name of the request parameter is different than the bean parameter name.

## Using Standard Tags

For example, if the form field for the email address of the customer had been `emailAddress`, then you can set the bean property named `email` using:

```
<jsp:setProperty name="cust" property="email"
param="emailAddress" />
```

The Java technology code equivalent of this code would look similar to the following:

```
cust.setEmail(request.getParameter("emailAddress"));
```

The `value` attribute can be used to supply the value to be used in the `set` method. For example, the value could be hard-coded in the JSP page:

```
<jsp:setProperty name="cust" property="email"
value="joe@host.com" />
```

Values for attributes can be specified using expressions, which are evaluated at runtime. For example:

```
<jsp:setProperty name="cust" property="email"
value='<%= someMethodToGetEmail() %>' />
```

The asterisk (\*) character can be used to specify all properties of the bean. The `setProperty` equivalent of the scriptlet in Code 7-5 on page 7-26 is as follows:

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

### The `getProperty` Tag

The `getProperty` tag is used to retrieve a property from a JavaBeans instance and display it in the output stream. The syntax of the `getProperty` tag is as follows:

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

The `name` attribute specifies the name of the JavaBeans instance and the `property` attribute specifies the property used for the `get` method.

Given a `getProperty` usage of the following:

```
<jsp:getProperty name="cust" property="email" />
```

the Java language equivalent would look as follows:

```
out.print(cust.getEmail());
```

The `getProperty` tag is a convenient mechanism to display the properties of a JavaBeans instance while avoiding scriptlet code. Code 7-6 shows an example of retrieving the contents of the `CustomerBean` for display.

```
72 <H2>Customer Information:</H2>
73 Name: <jsp:getProperty name="cust" property="name" /><BR>
74 Email: <jsp:getProperty name="cust" property="email" /><BR>
75 Phone: <jsp:getProperty name="cust" property="phone" /><BR>
```

### **Code 7-6** JSP Page `getProperty` Tag

## Other Standard Tags

The following tables outline the standard tags defined in the JSP specification.

Table 7-3 describes the standard actions available for including or forwarding to other documents.

**Table 7-3** Include and Forward Standard Actions

Standard Tag	Description
<code>jsp:include</code>	Performs an include dispatch to the resource provided
<code>jsp:forward</code>	Performs a forward dispatch to the resource provided
<code>jsp:param</code>	Adds parameters to the request, used within <code>jsp:include</code> and <code>jsp:forward</code> tags

Table 7-4 describes the standard actions available for embedding client-side technologies (such as Java applets) in the JSP page.

**Table 7-4** Standard Actions for Plug-ins

Standard Tag	Description
<code>jsp:plugin</code>	Generates client browser-dependent constructs (OBJECT or EMBED) for Java applets
<code>jsp:params</code>	Supplies parameters to the Java applet, used within the <code>jsp:plugin</code> tag
<code>jsp:fallback</code>	Supplies alternate text if the Java applet is unavailable on the client, used within the <code>jsp:plugin</code> tag

## Using the `jsp:include` Standard Action

The `jsp:include` standard action lets you include a segment in the text of the main JSP page at runtime. The syntax of this JSP technology action tag is as follows:

```
<jsp:include page="segmentURL" />
```



**Note** – You might have noticed that the `include` directive uses an attribute called `file` and the `jsp:include` standard action uses an attribute called `page`. These attributes represent the same concept: The URL to the segment file to be included.

Code 7-7 shows an example use of the `jsp:include` action.

```
113 <!-- START of navigation menu -->
114 <td bgcolor='#CCCCFF' width='160' align='left'>
115     <jsp:include page="/WEB-INF/view/common/navigation.jsp" />
116 </td>
117 <!-- END of navigation menu -->
```

### Code 7-7 The `jsp:include` Standard Action Example

The content of the segment is embedded into the HTTP response of the main JSP page at runtime. If the included segment also includes other segments, then all of the segments are embedded recursively until all of the `jsp:include` actions have been processed. All of this happens at runtime. The text of the segment *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 segment = request.getRequestDispatcher(page);
segment.include(request, response);
```

Semantically, the `include` directive and the `jsp:include` standard action achieve the same result: both include an HTML or JSP page segment in another JSP page. However, there are subtle differences:

- A page that uses the `include` directive *might not* be automatically updated when the segment file is updated in the web container.
- A segment 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 segment 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. Figure 7-9 shows this message added to the banner.



Figure 7-9 Thank You Page



The `jsp:include` action lets you pass additional parameters to the presentation segment 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="segmentURL">
  <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. Code 7-8 shows an example use of the `jsp:param` action.

#### Code 7-8 The param Standard Action

```
--
188 <!-- START of banner -->
189 <jsp:include page="/WEB-INF/view/common/banner.jsp">
190   <jsp:param name="subTitle" value="Registration" />
191 </jsp:include>
192 <!-- END of banner -->
```

The name-value pair specified in the `jsp:param` action is stored in the request object passed to the segment's JSP page. These name-value pairs are stored as if they were HTML form parameters. Therefore, the segment page uses the EL code `${param.subTitle}` to retrieve these values as shown in lines 17 and 20 in Code 7-9 on page 7-34Code 7-9 overleaf.

t

t

```

1  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2
3  <!-- Determine the banner title --%>
4  <c:choose>
5      <c:when test="not empty league">
6          <c:set var="bannerTitle">${league.title}</c:set>
7      </c:when>
8      <c:otherwise>
9          <c:set var="bannerTitle">Duke's Soccer League</c:set>
10     </c:otherwise>
11 </c:choose>
12
13 <font size='5' face='Helvetica, san-serif'>
14     ${bannerTitle}
15 </font>
16
17 <c:if test="${not empty param.subTitle}">
18     <br/><br/>
19     <font size='4' face='Helvetica, san-serif'>
20         ${param.subTitle}
21     </font>
22 </c:if>

```

**Code 7-9** The banner.jsp Segment

## Summary

This module reviews the following information:

- JSP pages are dynamic HTML pages that execute on the server.
- JSP pages are converted to raw servlets at runtime.
- You can use scripting elements to embed Java technology code to perform dynamic content generation.
- You can also use standard actions and the Expression Language to reduce the amount of Java technology code.
- The ultimate goal of JSP technology is to allow non-programmers to create dynamic HTML.



## Module 8

---

# Developing JSP Pages Using Custom Tags

---

## Objectives

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

- Describe the Java EE job roles involved in web application development
- Design a web application using custom tags
- Use JSTL tags in a JSP page

## Relevance



**Discussion** – The following questions are relevant to understanding the use of custom tag libraries:

- 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:

- Sun Microsystems, Inc., “Java EE 5 Tutorial, Chapter 6: JavaServer Pages Standard Tag Library,”  
[<http://java.sun.com/javaee/5/docs/tutorial/doc/JSTL.html>], accessed 02 December 2006.
- Hall, Marty. *Core Servlets and JavaServer Pages*. Upper Saddle River: Prentice Hall PTR, 2000.
- Delisle, Pierre, *JavaServer Pages Standard Tag Library, Version 1.1*. Sun Microsystems, 2003.

## The JSTL

Custom tags were originally created to permit development of JSP pages without using Java technology code (scripting elements). Since then, other mechanisms, notably the MVC approach supported by EL, have been developed that might be considered to provide a cleaner separation of concerns. However, many existing web applications exist that make extensive use of tags. Because of this, tags remain the preferred tool of some organizations for creating JSP pages without embedding Java programming language scriptlets into those pages.

This module looks in more detail at the tags that are available in the JSTL.

## The Java EE Job Roles Involved in Web Application Development

One reason behind the development of tags, and the JSTL, was to support the separate job roles often found in organizations developing large web applications. These roles often include:

- *Web Designers* – 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)

The tag library concept provides a means of integrating Java program behavior into a web page, without the need for web designers to learn Java themselves.



## Designing JSP Pages With Custom Tag Libraries

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.

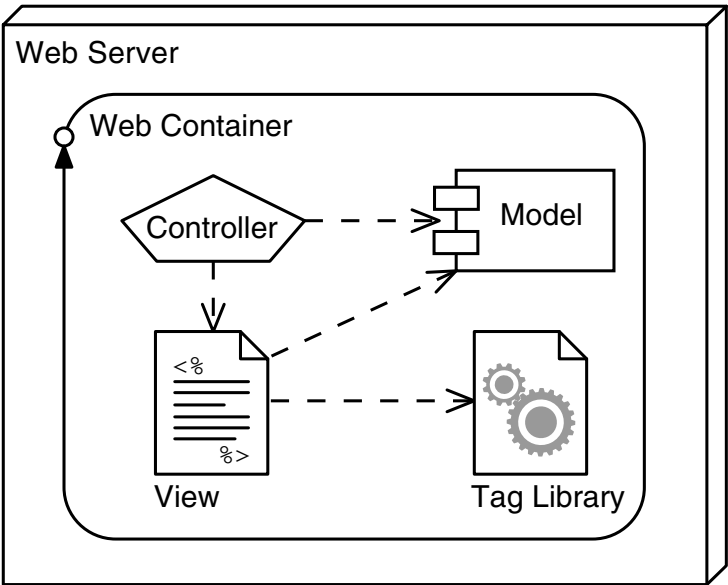
In “Presentation Programming” on page 6-14 you saw how the `<c:if`, and `<c:forEach` tags could be used to conditionally display a list of a number of errors.

Compared with coding these conditions and loops in Java scriptlets, JSTL tags offer these advantages:

- Java technology code is removed from the JSP page.  
Java technology code in a JSP page is hard to read and maintain. Java technology code in a JSP page is also difficult for some HTML editors to handle properly.
- Custom tags are reusable components.  
Scripting code is not reusable. Reusing 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. Reusing a custom tag only requires you to include the custom tag in the new JSP page.
- Standard job roles are supported.  
Web designers can use custom tags instead of scripting elements in JSP pages to facilitate dynamic behavior, without having to know the Java programming language.

# Custom Tag Library Overview

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. Figure 8-1 shows a tag library as a web component.



**Figure 8-1** Web Component Tag Library

Custom tag handlers used in a JSP page can access any object that is also accessible to the JSP page. This is accomplished by the `pageContext` object that is unique for a given JSP page and for a given request on that JSP page. The `pageContext` object provides access to all attribute scopes: page, request, session, and application. The `pageContext` object provides access to all implicit objects in the JSP page (request, response, out, and so on).

## 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:

```
<c:forEach var="message" items="${errorMsgs}">
    <li>${message}</li>
</c:forEach>
```

- Empty tag syntax must conform to the following structure:

```
<prefix:name {attribute={"value" | 'value'}}* />
```

Example:

```
<c:url value="addLeague.do" />
```

- Tag names, attributes, and prefixes are case sensitive.

For example:

- `<c:forEach>` is different from `<c:ForEach>`
- `<c:forEach>` is different from `<C:forEach>`
- `<c:forEach items="collection">` is different from `<c:forEach ITEMS="collection">`.
- Tags must follow nesting rules:

```
<tag1>
    <tag2>
    </tag2>
</tag1>
```

Here is an example of correct nesting:

```
<c:if test="${not empty errorMsgs}">
    <c:forEach var="message" items="${errorMsgs}">
        <%-- JSP code showing a single error message --%>
    </c:forEach>
</c:if>
```

Here is an example of incorrect nesting:

```
<c:if test="${not empty errorMsgs}">
  <c:forEach var="message" items="${errorMsgs}">
    <%-- JSP code showing a single error message --%>
  </c:if>
</c:forEach>
```

For more information about XML syntax, read Appendix D, “Quick Reference for XML.”

## JSTL Sample Tags

The JSTL core library contains several tags that reduce the scripting necessary in a JSP page. The `if` and `forEach` tags have been introduced already, and this section will examine more of the tags in the core library: `set`, `url`, and `out`.

### JSTL set Tag

You use the `set` tag to store a variable in a named scope, or update the property of a JavaBeans instance or Map.

There are two ways of using the `set` tag to store a variable:

```
<c:set var="varName" value="value"
[scope="{page|request|session|application}"] />
```

```
<c:set var="varName"
[scope="{page|request|session|application}"]>
value in body
</c:set>
```

The `var` attribute specifies the name of the variable, and the `scope` attribute specifies where to store the variable. The value of the variable can be provided by either the `value` attribute or the body of the tag.

For example, Code 8-1 demonstrates how the `set` tag is used to create the variable `pageTitle`, which is later used in an EL expression to specify the title of the JSP page.

```
28 <%-- Set page title --%>
29 <c:set var="pageTitle">Duke's Soccer League: Registration</c:set>
30
31 <%-- Generate the HTML response --%>
32 <html>
33 <head>
34   <title>${pageTitle}</title>
35 </head>
```

#### Code 8-1 The set Tag

There are also two ways to use the `set` tag to modify the property of a JavaBeans instance or Map:

```
<c:set target="targetName" property="propertyName"
      value="value" />
<c:set target="targetName" property="propertyName">
  value in body
</c:set>
```

The `target` attribute is used to specify the name of the existing JavaBeans instance or Map object. The `property` attribute is used to specify the JavaBeans attribute that will be set. The value of the property can be provided either using the `value` attribute or the body of the `set` tag.

For example, you could set the email property of a customer bean using the following:

```
<c:set target="cust" property="email"
      value="{param.emailAddress}" />
```

The value in the request parameter `emailAddress` is used to populate the `cust` bean's email property.

## JSTL `url` Tag

You use the `url` tag to provide a URL with appropriate rewriting for session management. The syntax of the tag is as follows:

```
<c:url value="value"
      [var="varName"]
      [scope="{page|request|session|application}"] />
```

The `value` attribute specifies the URL to which rewriting rules will be applied. The `var` attribute provides a variable name in which the rewritten URL will be stored. The `scope` attribute specifies the storage location of the variable. If `var` is not supplied, the URL is written to the current `JspWriter`.

Code 8-2 demonstrates how the `url` tag can be used in a JSP page to perform URL-rewriting.

```
156
157 <%-- Present the form --%>
158 <form action='<c:url value="enter_player.do" />' method='POST'>
```

### Code 8-2 The `url` Tag

The value attribute can also be used with absolute paths (relative to the web application's context root):

```
<form action='<c:url value="/register/enter_player.do" />'
      method='POST'>
...
</form>
```




---

**Note** – There are other variations of the url tag. You can see complete details in the JSTL specification.

---

## JSTL out Tag

You can use the out tag to evaluate an expression and write the result to the current `JspWriter`. Code 8-3 shows two forms of the tag.

```
<c:out value="value" [escapeXml="{true|false}"]
      [default="defaultValue"] />

<c:out value="value" [escapeXml="{true|false}"]>
  default value
</c:out>
```

### Code 8-3 The out Tag

The value attribute specifies the expression to be written to the `JspWriter`. The default attribute specifies the value to be written if the expression evaluates to null. If the `escapeXml` attribute is set to true, the characters (<), (>), (&), ('), and (") in the resulting string will be converted to their corresponding character entity values. For example, (<) would be converted to (&lt;). If set to false, these characters in the resulting string are not converted. The default value for the `escapeXml` attribute is true.

For example, if you want to display the value of the request parameter email or the string no\_email provided if the parameter does not exist, you could use the out tag.

```
<c:out value="${param.email}"
default="no_email provided" />
```

## Using a Custom Tag Library in JSP Pages

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.

A JSP page can use a tag library by directing the JSP technology translator to access the TLD. This is specified using the JSP technology `taglib` directive. This directive includes the TLD URI and a custom tag prefix. Code 8-4 shows this on lines 2 and 3.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="forms" uri="http://www.soccer.org/forms.tld" %>
```

### Code 8-4 JSP Page TLD Declaration

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. 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.

The URI attributes used in the `taglib` directive specify a symbolic location, not necessarily a document on the Internet. To use a tag library in a symbolic location, you can declare the TLD in the web application's deployment descriptor. Code 8-5 shows an example of declaring the TLD in the deployment descriptor.

```
<taglib>
<taglib-uri>http://www.soccer.org/forms.tld</taglib-uri>
<taglib-location>/WEB-INF/forms.tld</taglib-location>
</taglib>
```

### Code 8-5 Deployment Descriptor TLD

The TLD file is usually stored in the `WEB-INF` directory and the JAR file is stored in the `WEB-INF/lib` directory. The symbolic name can also be specified within the tag library's JAR file.



## JSTL Tags

This section contains a summary of the tags available in JSTL, based on functional category. Table 8-1 lists the five functional categories of tags in JSTL, their URI values (to be used in JSP `taglib` directives), and the typical prefix used for each.

**Table 8-1** JSTL Functional Areas

Functional Area	URI	Prefix
Core actions	<code>http://java.sun.com/jsp/jstl/core</code>	<code>c</code>
XML processing actions	<code>http://java.sun.com/jsp/jstl/xml</code>	<code>x</code>
Formatting actions	<code>http://java.sun.com/jsp/jstl/fmt</code>	<code>fmt</code>
Relational database access actions	<code>http://java.sun.com/jsp/jstl/sql</code>	<code>sql</code>
Function actions	<code>http://java.sun.com/jsp/jstl/functions</code>	<code>fn</code>

### Core Actions

Tags in the core functional area include actions to output to the response stream, perform conditional operations, and perform iterations.

Table 8-2 describes the tags available in this library.

**Table 8-2** The core Tag Library Tags

Tag	Purpose
<code>c:out</code>	Evaluates an expression and outputs the result to the current <code>JspWriter</code>
<code>c:set</code>	Sets the value of a scoped variable or property
<code>c:remove</code>	Removes a scoped variable
<code>c:catch</code>	Catches a <code>java.lang.Throwable</code> that occurs in the body of the tag
<code>c:if</code>	Evaluates the body of the tag if the expression specified by the <code>test</code> attribute is true

**Table 8-2** The core Tag Library Tags (Continued)

Tag	Purpose
c:choose	Provides a mutually exclusive conditional
c:when	Provides an alternative within a c:choose element
c:otherwise	Provides the last alternative within a c:choose element
c:forEach	Iterates over a collection of objects or for a fixed number of cycles
c:forEachTokens	Splits a string into tokens and iterates over those tokens
c:import	Imports the content of a URL resource
c:url	Rewrites relative URLs
c:redirect	Sends an HTTP redirect to the client
c:param	Adds parameters to the request (used within c:import, c:url, and c:redirect)

## XML Processing

JSTL includes tags for processing XML documents. Table 8-3 describes the tags available in this library.

**Table 8-3** The xml Tag Library Tags

Tag	Purpose
x:parse	Parses an XML document
x:out	Evaluates the XPath expression and outputs the result to the current JspWriter
x:set	Evaluates the XPath expression and stores the result in a scoped variable
x:if	Evaluates the body of the tag if the XPath expression is true
x:choose	Provides for a mutually exclusive conditional
x:when	Provides an alternative within an x:choose element
x:otherwise	Provides the final alternative within an x:choose element

**Table 8-3** The `xml` Tag Library Tags (Continued)

Tag	Purpose
<code>x:forEach</code>	Evaluates the XPath expression and repeats the body of the tag
<code>x:transform</code>	Applies an XSLT stylesheet to an XML document
<code>x:param</code>	Provides transformation parameters (used within an <code>x:transform</code> element)

## Formatting Actions

JSTL includes tags for internationalization and formatting. Table 8-4 describes these tags.

**Table 8-4** The `format` Tag Library Tags

Tag	Purpose
<code>fmt:setLocale</code>	Stores the specified locale in the locale configuration variable
<code>fmt:bundle</code>	Creates an i18n localization context that is used in the tag body
<code>fmt:setBundle</code>	Creates an i18n localization context and stores it in the scoped variable or the localization context configuration variable
<code>fmt:message</code>	Finds the localized message in the resource bundle
<code>fmt:param</code>	Supplies a parameter for replacement within a <code>fmt:message</code> element
<code>fmt:requestEncoding</code>	Sets the character encoding of the request
<code>fmt:timeZone</code>	Specifies the time zone in which time information in the body of the tag will be processed
<code>fmt:setTimeZone</code>	Stores the specified time zone in a scoped variable or the time zone configuration variable
<code>fmt:formatNumber</code>	Formats a numeric value in a locale-sensitive or customized manner as a number, currency, or percentage

**Table 8-4** The format Tag Library Tags (Continued)

Tag	Purpose
<code>fmt:parseNumber</code>	Parses the string representation of numbers, currencies, and percentages that were formatted in a locale-sensitive or customized manner
<code>fmt:formatDate</code>	Allows the formatting of dates and times in a locale-sensitive or customized manner
<code>fmt:parseDate</code>	Parses the string representation of dates and times that were formatted in a locale-sensitive or customized manner

## Relational Database Access

JSTL contains tags for database access. Table 8-5 describes these tags.

**Table 8-5** The database Tag Library Tags

Tag	Purpose
<code>sql:query</code>	Queries the database, storing the result set in a scoped variable
<code>sql:update</code>	Executes an INSERT, DELETE, UPDATE, or SQL DDL statement, storing the result in a scoped variable
<code>sql:transaction</code>	Establishes a transaction context for the <code>sql:query</code> and <code>sql:update</code> elements
<code>sql:setDataSource</code>	Exports a data source either as a scoped variable or as the data source configuration variable
<code>sql:param</code>	Sets the values for parameter markers (used with the <code>sql:query</code> and <code>sql:update</code> elements)
<code>sql:dateParam</code>	Sets the values for date parameter markers (used with the <code>sql:query</code> and <code>sql:update</code> elements)



**Note** – In an MVC architecture, do not perform database access from the view components. You probably want to avoid the JSTL database access tags.

## Functions

JSTL contains tags for functions, many of them from `java.lang.String`. Table 8-6 describes these tags.

**Table 8-6** Standard EL Functions

Tag	Purpose
<code>fn:contains</code>	Performs a case-sensitive test for a specified substring, returning true or false
<code>fn:containsIgnoreCase</code>	Performs a case-insensitive test for a specified substring, returning true or false
<code>fn:endsWith</code>	Tests if a string ends with a specified suffix, returning true or false
<code>fn:escapeXml</code>	Escapes characters that would be interpreted as XML markup
<code>fn:indexOf</code>	Returns the position of the first occurrence of a specified substring
<code>fn:join</code>	Joins the elements of an array into a string
<code>fn:length</code>	Returns the number of elements in a collection or the number of characters in a string
<code>fn:replace</code>	Returns a string after replacing any occurrences of a substring with another substring
<code>fn:split</code>	Splits a string into an array of substrings, based on a delimiter
<code>fn:startsWith</code>	Tests if a string begins with a specified prefix, returning true or false
<code>fn:substring</code>	Returns a subset of a string, defined by begin and end points

**Table 8-6** Standard EL Functions (Continued)

Tag	Purpose
<code>fn:substringAfter</code>	Returns the substring that follows the specified substring
<code>fn:substringBefore</code>	Returns the substring that precedes the specified substring
<code>fn:toLowerCase</code>	Converts the characters in a string to lowercase
<code>fn:toUpperCase</code>	Converts the characters in a string to uppercase
<code>fn:trim</code>	Removes white space from both ends of a string

## Summary

This module reviews the following information:

- Custom tags are fundamentally the same as standard tags, but you can acquire tag libraries from third parties and even build your own application-specific tags.
- JSTL provides a collection of general purpose tags.
- You can use a tag library in your JSP pages by declaring it using the `<%@ taglib %>` directive.
- Custom tags use standard XML tag syntax.
- With custom tags, standard tags, and the Expression Language, you can eliminate all scriptlet code in your JSP pages.

## Summary

---



## Module 9

---

# More Controller Facilities

---

## Objectives

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

- Understand the lifecycle of a servlet
- Understand the threading model of a servlet
- Write filters and apply them to groups of servlets or JSPs
- Handle multipart form data

## Relevance

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

- How are servlets loaded, when, and how many instances are created?
- What happens when multiple clients request service from the same servlet concurrently?
- What happens when a piece of behavior should be applied to multiple pages? Does the code have to be duplicated in multiple servlets?

## Additional Resources

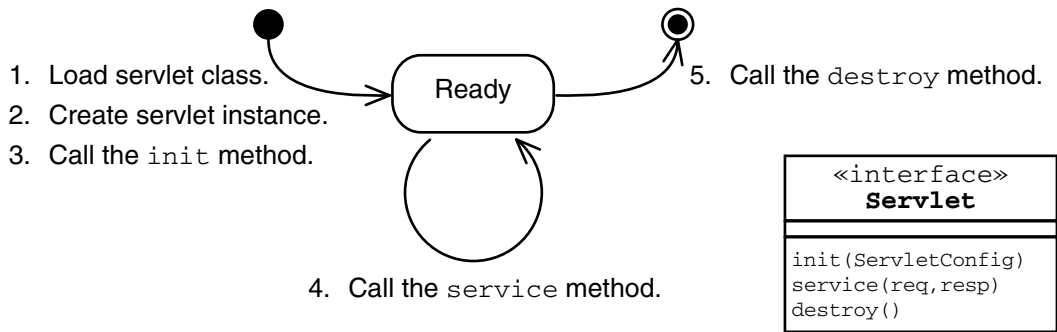


The following references provide additional details on the topics that are presented briefly in this module and described in more detail later in the course:

- Java Servlets Specification. [Online]. Available:  
<http://java.sun.com/products/servlet/>
- JavaServer Pages Specification. [Online]. Available:  
<http://java.sun.com/products/jsp/>
- Java Platform, Enterprise Edition 5 API Specification. Available:  
<http://java.sun.com/javaee/6/docs/api>
- Java Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans™ IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- The Common Gateway Interface. [Online]. Available:  
<http://hoohoo.ncsa.uiuc.edu/cgi/>

# Servlet Life Cycle Overview

In this section, you learn about the life cycle management of a servlet component. In Figure 9-1, a graphical view of the servlet life cycle is provided by an annotated UML Statechart diagram.



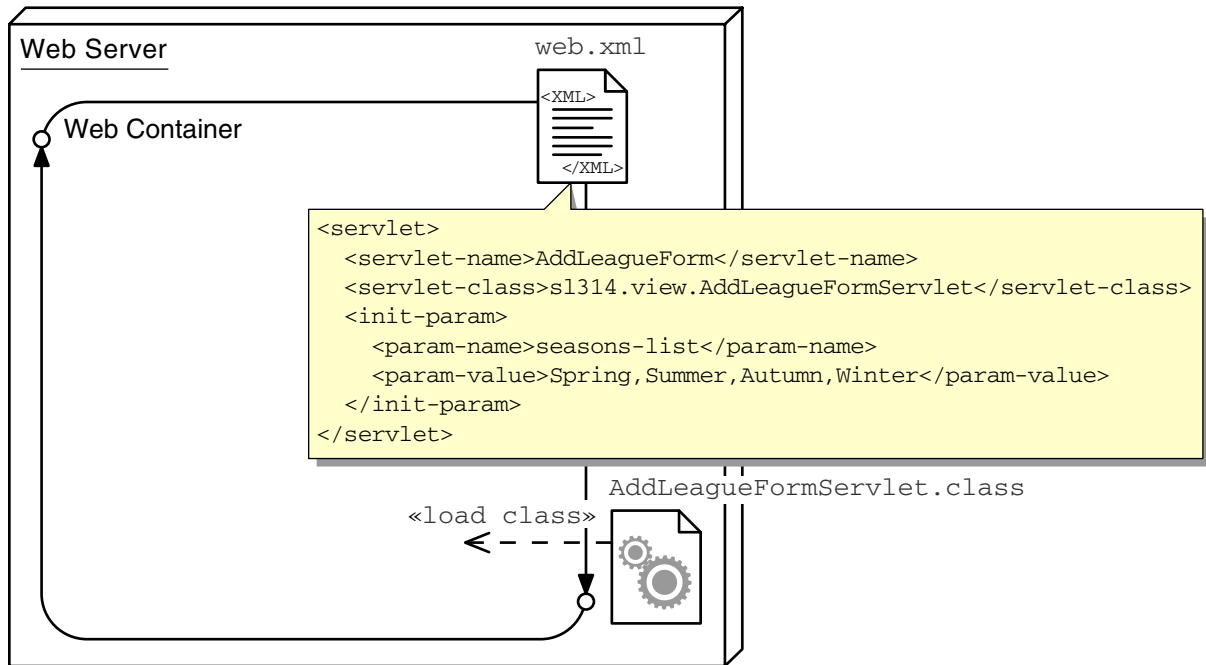
**Figure 9-1** Servlet Life Cycle

The web container manages the life cycle of a servlet instance. Steps 3, 4, and 5 include methods that are called on a servlet instance. These methods are part of the essential `Servlet` interface.

These methods should not be called by your code. These methods are called by the web container to manage the servlet.

## Servlet Class Loading

The first step in the creation of a servlet component is to load the servlet class file into the web container's JVM, as shown in Figure 9-2. The servlet definition supplies the class name for the servlet component.



**Figure 9-2** Step 1: Web Container Loads the Servlet Class

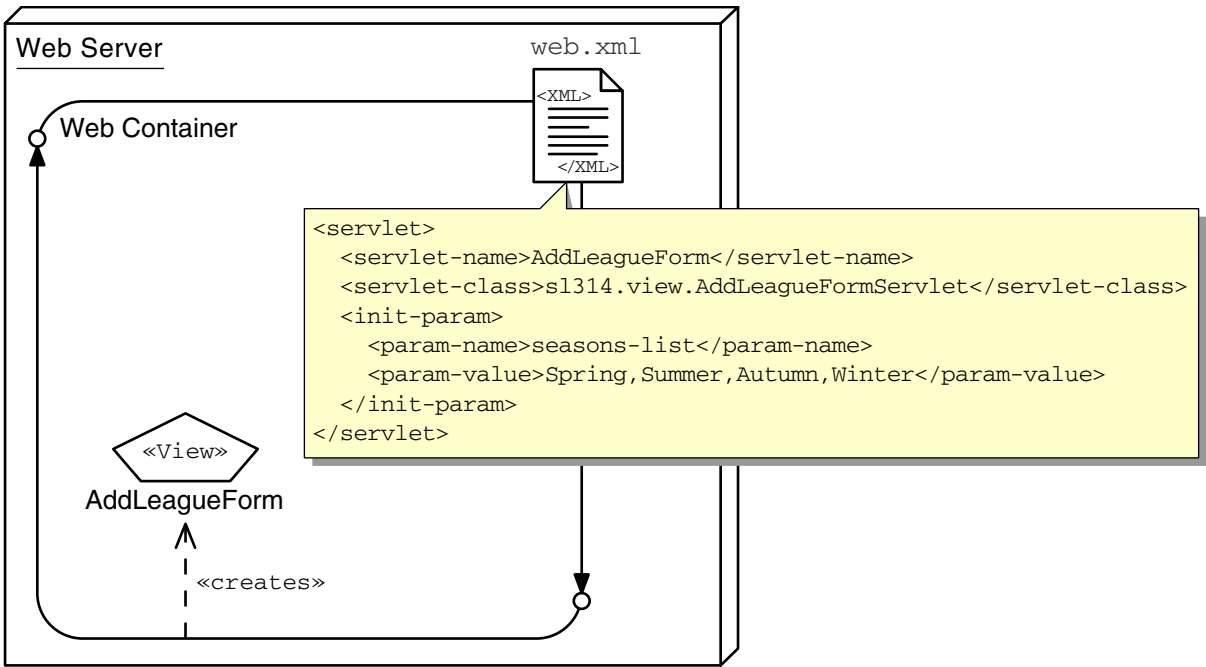
The classes for a web application must exist in well-defined locations within the web application physical hierarchy. In particular, class files can be placed under the `WEB-INF/classes/` directory. The structure of the hierarchy under this directory must match the package structure of the classes. For example, the `AddLeagueFormServlet.class` file must exist under the `/WEB-INF/classes/sl314/view/` directory.

You can also include JAR files for additional classes under the `/WEB-INF/lib/` directory. Typically, this is where you place third-party or utility JAR files, such as a driver supporting the JDBC™ API (JDBC driver) JAR files.

Your web application might also rely on standard Java classes or container-specific classes. Web container vendors are allowed to expand the *classpath* of your web application in clearly defined, but vendor-specific, ways.

# One Instance Per Servlet Definition

After the servlet class has been loaded into the JVM, the next step is to create an instance of that class, as shown in Figure 9-3. The Servlet specification declares that one, and only one, servlet instance will be created for a single servlet definition in the deployment descriptor. This is an important fact to remember when developing servlets: Requests to a given servlet definition will all be handled concurrently by the same servlet instance. Therefore, your servlet code must be thread-safe.



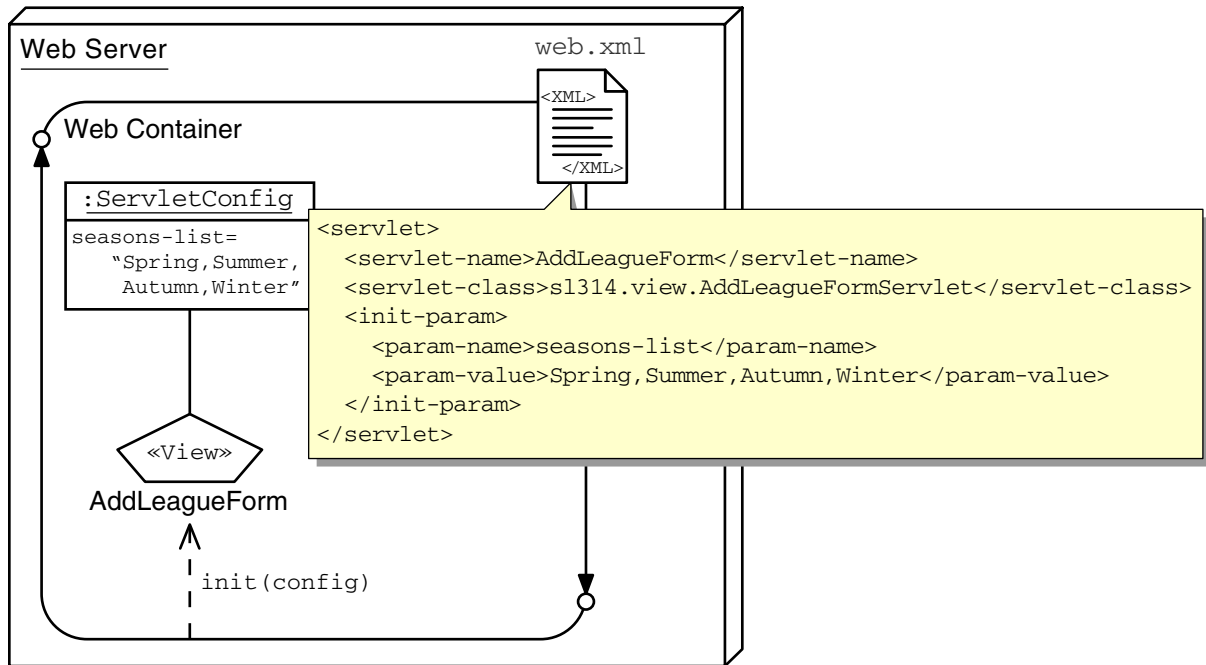
**Figure 9-3** Step 2: Web Container Creates a Servlet Class Instance



**Note** – Older versions of the servlet specification (before v2.4) did allow for web containers to create multiple servlet instances and pool them. This gave each HTTP request a unique servlet instance. This was done when a servlet class implemented a special, marker interface called `SingleThreadModel`. This interface has been deprecated with no replacement in the v2.4 specification.

## The `init` Life Cycle Method

After the servlet instance is created, the web container creates a unique `ServletConfig` object that contains any initialization parameters that were specified in the deployment descriptor, as shown in Figure 9-4.



**Figure 9-4** Step 3: Web Container Provides Config Object

Your servlet class can override the `init` method and use the `ServletConfig` object to obtain initialization and context parameters. Clearly, if the servlet needs a value from the context, for example the email address of the current webmaster, it is more efficient to load it once during initialization and then use it from a local variable than it would be to read the context information every time. This would, therefore, be a good use for initialization.

Similarly, there are times when a servlet needs a handle on some external resource, such as a database, and it would be more efficient to load it once rather than do so every time a client request is handled. This too would be a good use for the `init` method.

The first three steps (load class, create instance, call the `init` method) only occur once for a given servlet definition.

Errors during initialization

The servlet's `init` method can throw an `UnavailableException` or a `ServletException` to indicate that it failed to configure properly. In this case, the servlet will not be placed into active service and cannot be used.

The `UnavailableException` can be used to specify that the servlet will be able to initialize correctly later, with a delay period specified in seconds.

The ServletConfig API

Figure 9-5 shows the servlet API and how it relates to the `ServletConfig` interface.

Every web container vendor must implement the `ServletConfig` interface. Instances of this class are passed into the `init (ServletConfig)` method defined in the `Servlet` interface.

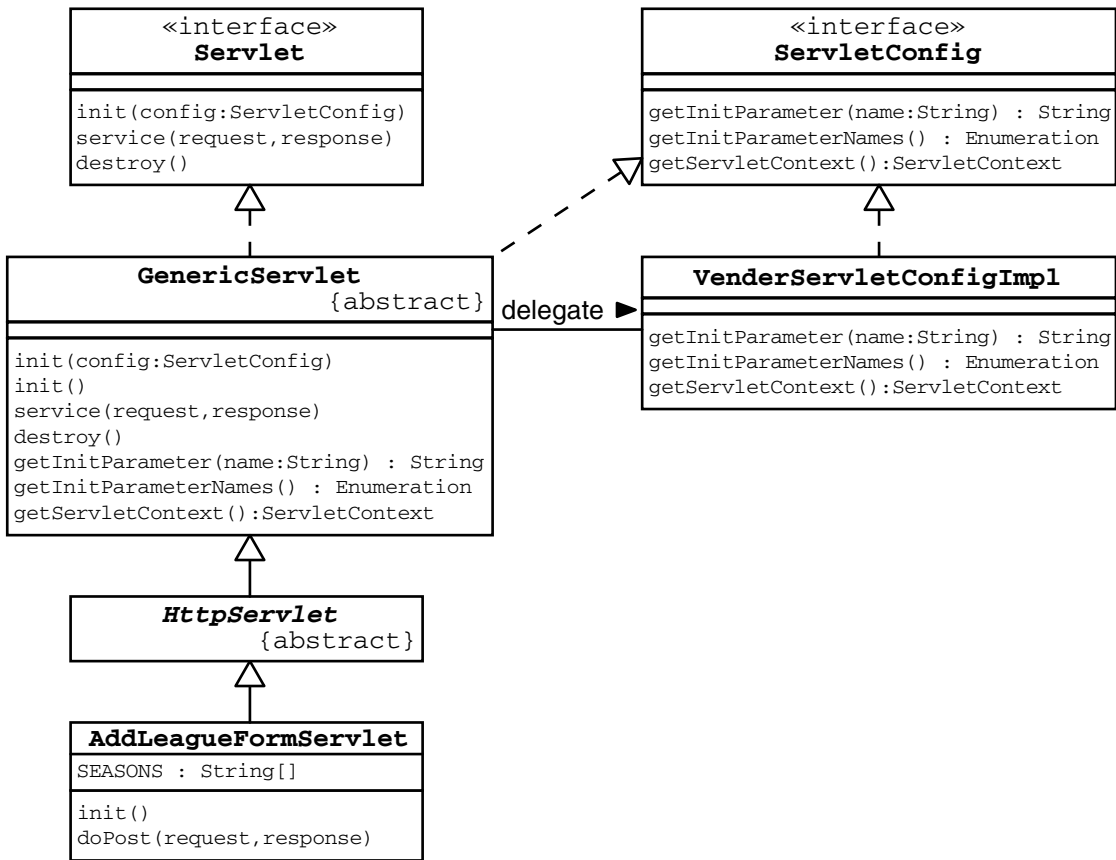


Figure 9-5 The ServletConfig API

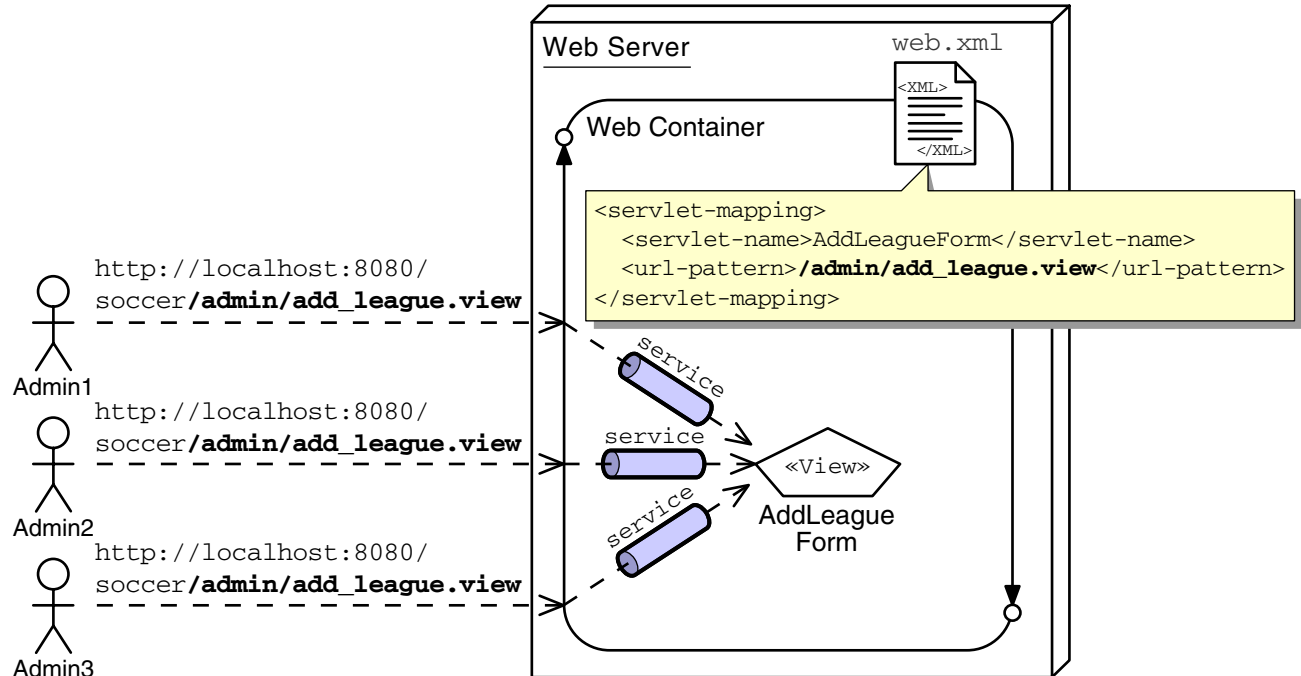


The `GenericServlet` class (supplied in the servlet API) implements the `Servlet` interface. It implements the `init(ServletConfig)` method, which stores the config object (the *delegate*) and then calls the `init()` method. It is this no-argument `init` method that you override in your servlet classes.

The `ServletConfig` interface provides a method, `getInitParameter`, which lets you retrieve your servlet's initialization parameters. As a convenience, the `GenericServlet` class also implements the `ServletConfig` interface, and those methods delegate the calls to the stored config object. This simplifies your servlet code by letting you directly call the `getInitParameter` method (without direct access to the config object).

## The service Life Cycle Method

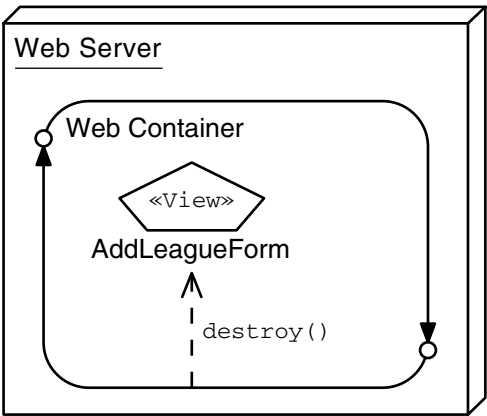
After the servlet component has been initialized, the web container can begin sending requests to that component using the `service` method. For each HTTP request mapped to a given servlet, the web container will issue a unique request and response object to the `service` method. Figure 9-6 shows that each request is also processed in a unique thread.



### Figure 9-6 Step 4: Servlet Is Ready for Service

# The destroy Life Cycle Method

When a web application is being shut down, the web container will call the `destroy` method on every servlet that has been initialized, as shown in Figure 9-7. You can override the `destroy` method to clean up any resources that your servlet might have initialized. The `destroy` method is only called once on a given servlet instance.



**Figure 9-7** Step 5: Web Container Calls the `destroy` Method

The web container can choose to destroy any servlet at any time. This might be done to free up system resources, for example. The implication is that a servlet might not exist at any given time during the life time of the web application. You cannot depend on it always being there. However, the servlet specification guarantees that a servlet instance will be created whenever an HTTP request for that component is issued against the web container.

# Servlet Lifecycle and Annotations

Java EE 5 introduced some key annotations for container managed objects. In the web container, these objects are Servlets, Filters, and many listeners.

Many of the annotations specify dependency injection. Dependency injection is a mechanism that allows the container to initialize a variable in an object without the object including any direct code to do so. In that sense, dependency injection is an idiom of the factory pattern concept.

Dependency injection is performed before the object is considered to exist from the perspective of the application. Specifically, injection is completed before any lifecycle methods are called, and before a reference to the object is made available to the application.

Dependency injection is required for classes of the specified types found under the `WEB-INF/classes` and `WEB-INF/lib` directories, although containers are permitted to process classes found elsewhere. Clearly, depending on these optional processing locations would result in a non-portable application.

A variety of annotations provide dependency injection for different resource types. Some of these are listed below:

- `@EJB`
- `@Resource`
- `@PersistenceContext`
- `@PersistenceUnit`
- `@WebServiceRef`

In addition, several plural versions of these annotations are defined so that multiple resources may be injected.

Use of each of these annotations requires some knowledge of the resource to be injected, and is therefore beyond the scope of this module.

## Lifecycle Method Annotations

The Java EE specification requires that objects that qualifies dependency injection must also be supported with two lifecycle annotations. These are `@PostConstruct` and `@PreDestroy`.

These methods are very similar in function to the `init()` and `destroy()` methods of a servlet. However, the annotations may be applied to any method that takes zero arguments, returns void, does not throw any checked exceptions, and is non-final. The method need not be public, and need not adopt any pre-defined name. This relaxes the requirements of the interface-mandated `init` and `destroy` methods.

If an `@PostConstruct` method throws any exceptions the container must abandon the object. No methods may be called on the object and the object must not be put into service.

The `@PostConstruct` method is guaranteed to be called after all injection has been completed, and before the call to the servlet's `init` method.

# Servlets and Threading

This class has mentioned in passing that servlets, and with them JSPs, run in a multi-threaded environment. Specifically, this means that if multiple clients invoke behavior that calls the same method, in the same servlet instance, at the same time, then that one servlet method (in a single servlet instance), might be concurrently executing many times under the control of many different threads—one thread for each client.

This is a great benefit to throughput, as each client request is not forced to wait for the ones before it. However, this might also lead to concurrency problems if the threads are sharing data or other resources.

## Handling Concurrency

Some data should be shared between different invocations of a servlet's `doXXX()` method. Such data are often read-only, such as configuration information. In this situation, all that need be done is to use an instance variable and set the value during servlet initialization.

Some other data should not be shared, and should instead be local to each invocation by a client. This is easy to handle too. Such data should be stored using method local (also known as automatic) variables.

Some data that are not read-only, however, must be shared; either between multiple invocations by the same client, or between multiple invocations of a servlet by different clients. This requires more thought. While the details of concurrency control are beyond the scope of this course, you should recognize the various situations that might arise, and understand how the API and environment of the web container impacts concurrency.

## Data Shared Between Invocations by a Single Client

Data that are shared between invocations by a single client are essentially session data. At first glance, this should simply involve using the `HttpSession` object to ensure that the lifetime and visibility of the data are appropriate. Unfortunately, however, it is possible for a single client to invoke servlets concurrently. This can arise if multiple browser windows or tabs are open.

It should be noted that this behavior is not normal in a web-based user interface and some browsers attempt to prevent it, but it can arise if a hacker is attempting to use poor concurrency control on your part to attack your system. Such an attacker might not even be using a browser to send the requests, consequently, it is proper to address this possibility, though it must be noted that such issues are often ignored.

## Sharing Data Between Multiple Clients

Where modifyable data are shared between multiple clients, more care must be taken, and it is not safe or reasonable to consider overlooking the issue.

Two guidelines must be observed. First, perhaps obviously, some kind of arbitration must be enforced between the threads so that data corruption cannot happen. In practice, this might be handled using transactions on the database, but data stored in memory, in the application context, must be protected manually by the programmer—that is, by you.

The second guideline is that the arbitration mechanism used by the servlets must not block other threads for longer than is essential to the correct operation of the program. Failure to observe this will create a functionally correct program that exhibits very poor scalability. It might pass unit testing, but unless load testing is properly performed, the application will probably fail to perform well in production.

The sharing of data between clients can be achieved by storing those items in instance variables in the servlet object itself. However, if you are using older web containers prior to 2.4 of the servlet specification, be aware that many used to create multiple instances of each servlet class. If this happens, writing data to the instance variable in one servlet will not reliably result in all other threads seeing that data, because some threads will execute in other servlet instances. Instead, such data should be stored in the application context.



---

**Note** – The `SingleThreadModel` interface has been deprecated. If you didn't know about it, don't worry. It didn't solve the problem for which it was intended.

---

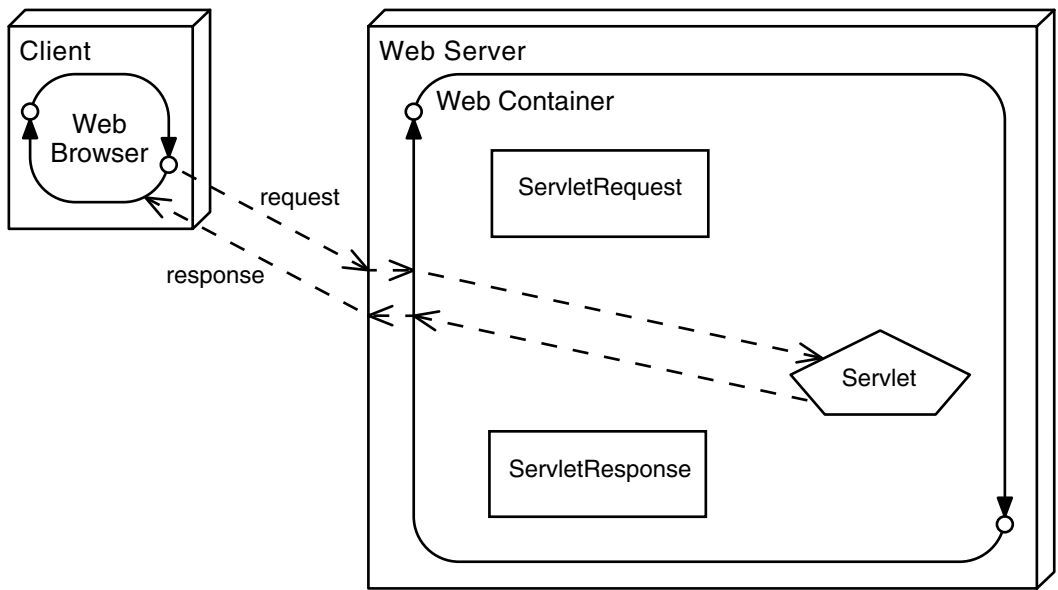
# Web Container Request Cycle

One of the benefits to the servlet framework is that the web container intercepts incoming requests before they get to your code, preprocessing the request and additional functionality (such as security). The container can also post-process the response that your servlet generates. In this module, you learn how to add components, called *filters*, to achieve this container functionality. These filters provide additional preprocessing of the request, as well as post-processing of the response.

## Web Container Request Processing

You have seen how the container creates the request and response objects when a request is sent from the client. These objects are then passed to the servlet. The servlet processes the request, and then produces a response which goes back to the client.

Figure 9-8 shows the creation of a request and a response for each request.



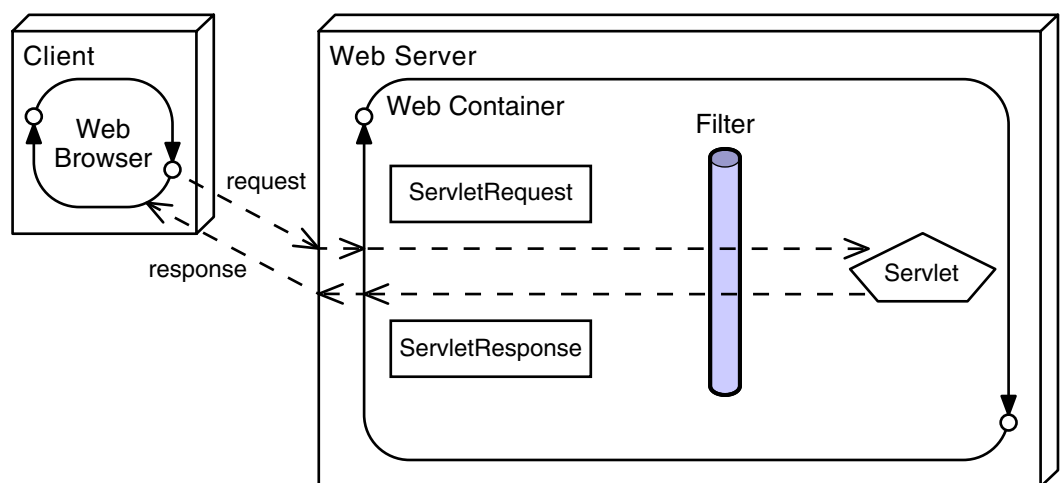
**Figure 9-8** Request and Response

## Applying Filters to an Incoming Request

Filters are components that you can write and configure to perform some additional pre-processing and post-processing tasks. When a request is received by the web container, several operations occur:

1. The web container performs its pre-processing of the request. What happens during this step is the responsibility of the container provider.
2. The web container checks if any filter has a URL pattern that matches the URL requested.
3. The web container locates the first filter with a matching URL pattern. The filter's code is executed.
4. If another filter has a matching URL pattern, its code is then executed. This continues until there are no more filters with matching URL patterns.
5. If no errors occur, the request passes to the target servlet.
6. The servlet passes the response back to its caller. The last filter that was applied to the request is the first filter applied to the response.
7. The first filter originally applied to the request passes the response to the web container. The web container might perform post-processing tasks on the response.

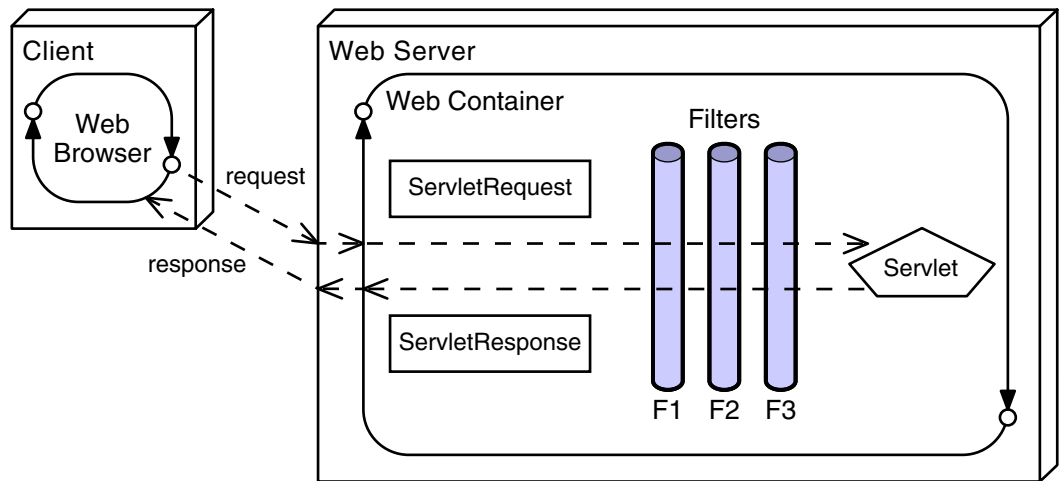
Figure 9-9 shows the cycle of a filter intercepting the request and response.



**Figure 9-9** Request and Response Filter Interception



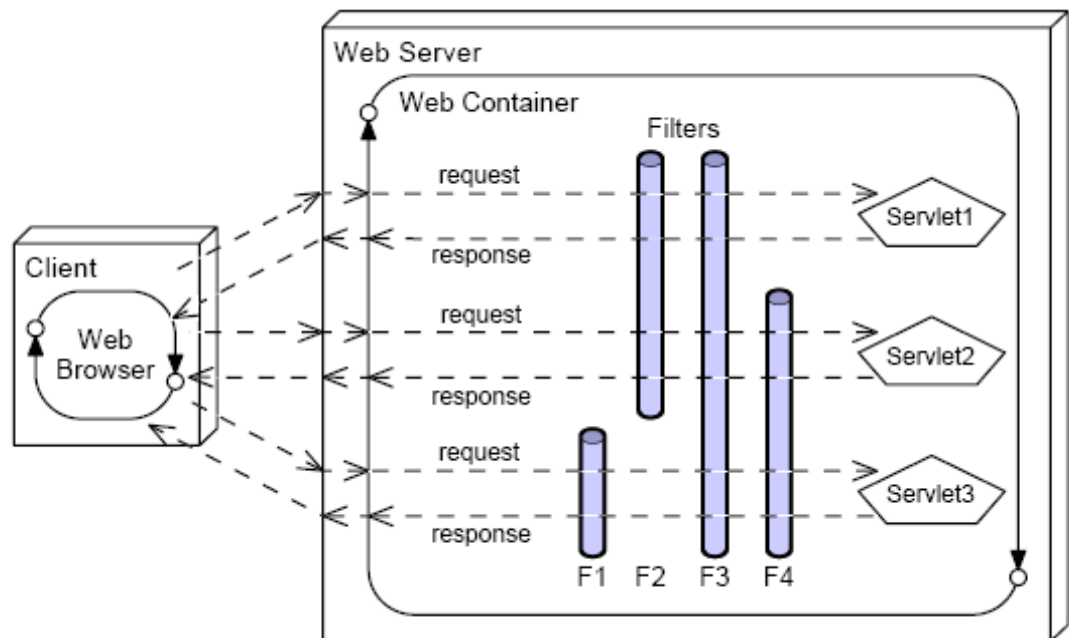
Figure 9-10 shows how you can chain several filters together to provide modular pre-processing and post-processing.



**Figure 9-10** Several Filters Used in a Chain

You should design filters to perform one task in the application, which helps ensure that they remain modular and reusable. You do not need to apply the same filters to every request.

Figure 9-11 shows modular use of filters.



**Figure 9-11** Modular Use of Filters

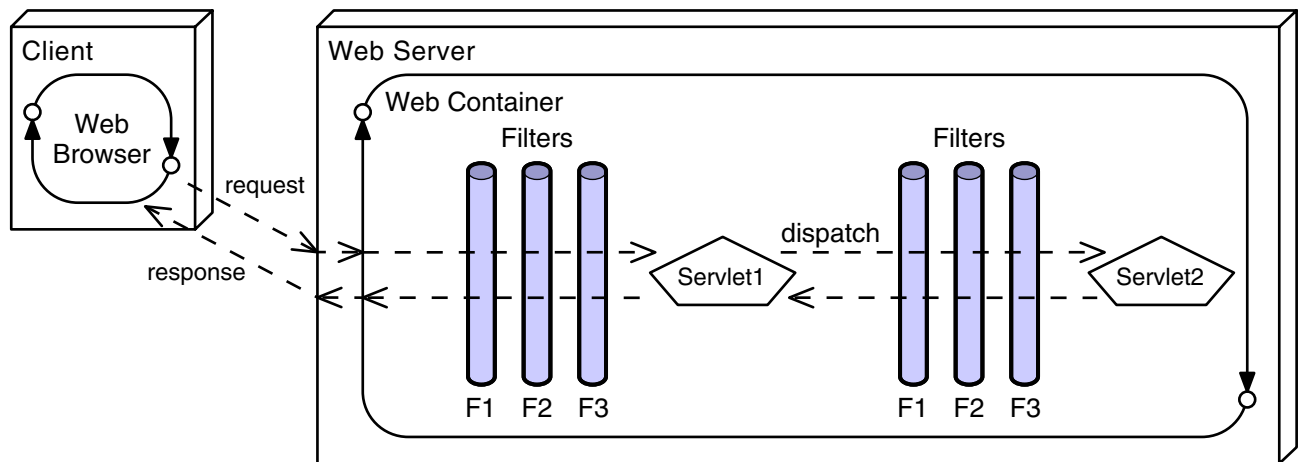
Filters can be used for operations such as:

- Blocking access to a resource based on user identity or role membership
- Auditing incoming requests
- Compressing the response data stream
- Transforming the response
- Measuring and logging servlet performance

## Applying Filters to a Dispatched Request

While filters will intercept direct requests from a client, they can also be applied to dispatches within the web container.

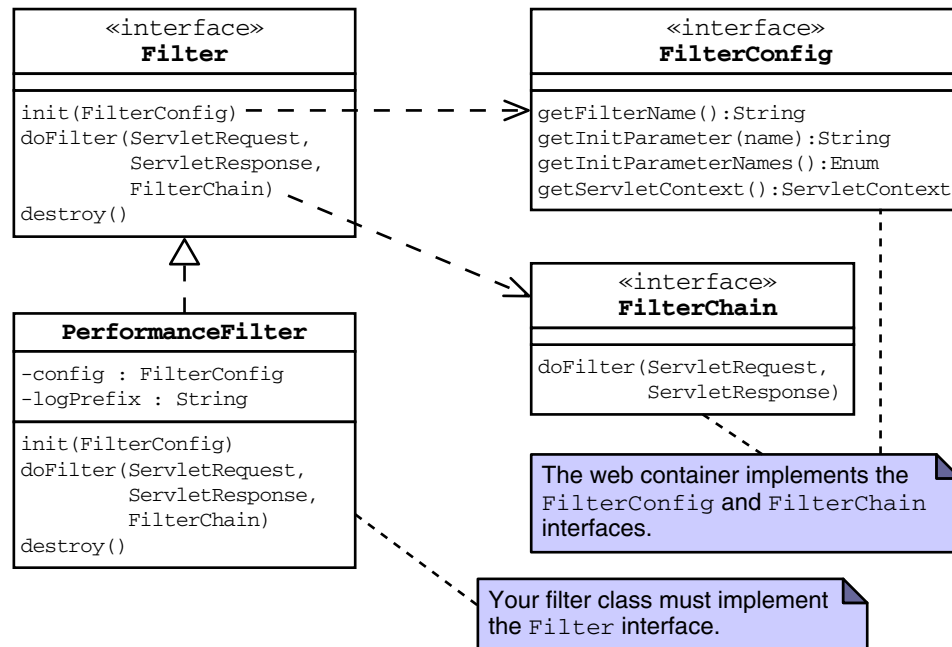
In Figure 9-12, filters are applied to the request for Servlet1. When Servlet1 dispatches to Servlet2, additional filters are applied. This mechanism works for both include and forward dispatches.



**Figure 9-12** Filters Applied to a Dispatch

# Filter API

The Filter API is part of the base servlet API. The interfaces can be found in the `javax.servlet` package. Figure 9-13 shows the three interfaces that you use when writing a filter.



**Figure 9-13** The Filter API

When you write a filter, you implement the `Filter` interface. For example, in Figure 9-13, the `PerformanceFilter` class implements the `Filter` interface. This module describes the implementation of the `PerformanceFilter` class.

The container passes your `init` method a `FilterConfig` reference. You should store the `FilterConfig` reference as an instance variable in your filter. You use the `FilterConfig` reference to obtain filter initialization parameters, the name of the filter, and the `ServletContext`.

When the filter is called, the container passes your `doFilter` method the `ServletRequest`, `ServletResponse`, and `FilterChain` references. You use the `FilterChain` reference to pass the request and response objects to the next component (filter or target servlet) in the chain.

## Developing a Filter Class

This section examines the `PerformanceFilter` class from Figure 9-13 on page 9-19.

### The `PerformanceFilter` Class

The `PerformanceFilter` class must implement the `Filter` interface, as shown in Code 9-1.

```
1  package sl314.web;
2
3  import java.io.IOException;
4
5  import javax.servlet.ServletException;
6  import javax.servlet.ServletResponse;
7  import javax.servlet.ServletException;
8  import javax.servlet.http.HttpServletRequest;
9
10 import javax.servlet.Filter;
11 import javax.servlet.FilterChain;
12 import javax.servlet.FilterConfig;
13
14 public class PerformanceFilter implements Filter {
15
16     private FilterConfig config;
17     private String logPrefix;
18 }
```

#### **Code 9-1** Declaration of the `PerformanceFilter` Class

Notice the two instance variables on lines 16 and 17. These will be initialized by the `init` method.

## The init Method

The `init` method of a filter is called once when the container instantiates a filter. This method is passed a `FilterConfig`, which is typically stored as a member variable for later use, as shown in Code 9-2. You can perform any one-time initialization tasks in the `init` method. For example, if the filter has initialization parameters, these can be read in the `init` method using the `FilterConfig` object's `getInitParameter` method.

```
73 public void init(FilterConfig config)
74     throws ServletException {
75     this.config = config;
76     logPrefix = config.getInitParameter("Log Entry Prefix");
77 }
```

**Code 9-2** The `init` Method

## The doFilter Method

The `doFilter` method is called for every request intercepted by the filter. It is the filter equivalent of a servlet's `service` method. This method is passed three arguments: `ServletRequest`, `ServletResponse`, and `FilterChain`. You use the request object to obtain information from the client, such as parameters or header information. The response object is used to send information back to the client, such as header values.

Within the `doFilter` method, you must decide to either invoke the next component in the filter chain or block the request. If the preferred behavior is to invoke the next component, then you call the `doFilter` method on the `FilterChain` reference. The next component in the chain could be another filter or a web resource, such as a servlet.

Code 9-3 shows the `doFilter` method in the `PerformanceFilter` class.

```
1
2 public void doFilter(ServletRequest request,
3     ServletResponse response, FilterChain chain)
4     throws ServletException, IOException {
5
6     long begin = System.currentTimeMillis();
7     chain.doFilter(request, response);
8     long end = System.currentTimeMillis();
9
10    StringBuffer logMessage = new StringBuffer();
```

## Developing a Filter Class

---

```
11     if (request instanceof HttpServletRequest) {
12         logMessage = ((HttpServletRequest)request).getRequestURL();
13     }
14     logMessage.append(": ");
15     logMessage.append(end - begin);
16     logMessage.append(" ms");
17
18     if(logPrefix != null) {
19         logMessage.insert(0,logPrefix);
20     }
21
22     config.getServletContext().log(logMessage.toString());
23 }
```

### Code 9-3 The doFilter Method

In Code 9-3 above, the filter obtains the current time (in milliseconds) on line 6 before sending the request and response to the next element in the chain on line 7. On line 8, the filter obtains the current time again. The difference in these time values is the time it took any other components in the filter chain (other filters, servlets, model components, and so on) to perform their operations. On line 22, this time difference is written to the log file using the ServletContext object's log method.

## The destroy Method

Before the web container removes a filter instance from service, the destroy method is called. You use this method to perform any operations that need to occur at the end of the filter's life.

Code 9-4 shows an example of releasing any resources that were allocated in the filter's init method.

```
24     public void destroy() {
25         config = null;
26         logPrefix = null;
27     }
```

### Code 9-4 The destroy Method

## Configuring the Filter

Because the web container is responsible for the life cycles of filters, filters must be configured in the web application's deployment descriptor.

### Configuring a Filter Using Annotations

Java EE 6 provides an annotation that allows declaration and mapping of a filter. The annotation is `@WebFilter`, and this allows specification of the filter name, the url patterns to which it responds, and the types of invocation for which it should be invoked. The annotation also allows the specification of init parameters. A complete example with all these features is shown below:

```
@WebFilter(filterName="PerfFilter", urlPatterns={"*.do"},
dispatcherTypes={DispatcherType.FORWARD, DispatcherType.ERROR,
DispatcherType.REQUEST, DispatcherType.INCLUDE},
initParams={@WebInitParam(name="Log Entry Prefix", value="Performance:")})
```

### Declaring a Filter in the `web.xml` File

For older Java EE versions, or for version independence, you can define and map filters using the `web.xml` deployment descriptor file.

Code 9-5 shows a filter declared within filter elements in the deployment descriptor.

```
52  <filter>
53    <filter-name>perfFilter</filter-name>
54    <filter-class>sl314.web.PerformanceFilter</filter-class>
55    <init-param>
56      <param-name>Log Entry Prefix</param-name>
57      <param-value>Performance: </param-value>
58    </init-param>
59  </filter>
```

#### Code 9-5 Filter Declaration

At minimum, a filter declaration must contain the `filter-name` and `filter-class` elements. Optional initialization parameters are placed within the filter declaration in `init-param` elements.

## Declaring a Filter Mapping in the `web.xml` File

Filters are executed when the resource to which they are mapped is requested. Code 9-6 shows this mapping specified in the deployment descriptor within a `filter-mapping` element.

```
260 <filter-mapping>
261   <filter-name>perfFilter</filter-name>
262   <url-pattern>*.do</url-pattern>
263 </filter-mapping>
```

### Code 9-6 Filter Mapping

The `url-pattern` element can specify a particular URL or use a wild card to match a set of URLs. The `url-pattern` element can be replaced with the `servlet-name` element to create a filter that applies to a particular servlet.

The same `url-pattern` and `servlet-name` values can be used in multiple filter mappings to create a chain of filters that are applied to the request. Filters are applied in the order they appear in the deployment descriptor. All the URL mapped filters are applied before the servlet mapped filters.

For example, given the following deployment descriptor declarations:

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

<filter-mapping>
  <filter-name>perfFilter</filter-name>
  <servlet-name>FrontController</servlet-name>
</filter-mapping>

<filter-mapping>
  <filter-name>auditFilter</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>transformFilter</filter-name>
  <url-pattern>*.do</url-pattern>
</filter-mapping>
```



If a request was made for the URL `/admin/add_league.do`, the filters would be applied in the order: `auditFilter`, `transformFilter`, and then `perfFilter`.

Filters are typically only applied to requests coming directly from a client. Providing a dispatcher element for the filter mapping allows filters to be applied to internal dispatches, as well as direct requests from clients.

- A dispatcher element with the value `REQUEST` indicates that the filter will be applied to requests from a client.
- A dispatcher element with the value `INCLUDE` will invoke the filter for matching `RequestDispatcher.include` calls.
- A dispatcher element with the value `FORWARD` will invoke the filter for matching `RequestDispatcher.forward` calls.
- A dispatcher element with the value `ERROR` indicates that the filter is applied when an error occurs (the dispatch is due to an error condition).

You can supply multiple dispatcher elements if you want the filter to be applied in multiple scenarios.

For example, given the following deployment descriptor segment:

```
<filter-mapping>
  <filter-name>auditFilter</filter-name>
  <url-pattern>*.do</url-pattern>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

The `auditFilter` would only be applied to internal dispatches to the URL pattern `*.do`. Any direct client requests for a URL which matches the `*.do` pattern would not have the `auditFilter` applied to them.

## Handling Multipart Forms

Java EE 6 introduced a mechanism to simplify handling of multipart form data. Three key elements make this up, which are: additional methods in the `javax.servlet.http.HttpServletRequest` object, an annotation `javax.servlet.annotation.MultipartConfig`, and an interface `javax.servlet.http.Part`.

The request now includes the methods `getParts` and `getPart`. Each part provided by a multipart form is named, in much the same way as regular forms submit named parameters. Using the name, the parts can be extracted individually using the `getPart` method. Alternatively, the method `getParts` returns a `Collection` of `Part` objects, which may then be iterated.

The `Part` interface provides several methods. In particular, the interface allows for determining the type of the content, obtaining a stream that allows reading the contents, releasing the storage used for the contents on the server and moving the written file to another location on disk.

When uploading content, the server needs to be able to control certain aspects of the interaction, such as the maximum size of data that may be uploaded, and where on the disk the data will be stored by default. The `MultipartConfig` annotation provides control of these details.

The following example illustrates a simple example of using these features to upload a data file and a text field from a multipart form.

Code 9-7 on page 9-27 shows a simple servlet that will respond to the submission of a multipart form that attempts to upload a text field and a file. The file is stored in a single location `/tmp/DATAFILE` and each new request will overwrite data from previous requests.

In this example, the file sizes are unrestricted, and the `MultipartConfig` annotation is used only to set a default location for storage of uploaded fields.

Notice that the text field is handled in the same way as a file. It appears as an `InputStream`, and must be read from that stream. This example does not make any use of the uploaded file, instead it is simply allowed to occupy disk space in the specified (`/tmp`) directory. In a full system, it would be necessary to ensure that the file was uniquely named, and presumably make some use of the data contained in the file, otherwise the upload was pointless.

```
1  package web;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import javax.servlet.RequestDispatcher;
7  import javax.servlet.ServletException;
8  import javax.servlet.annotation.MultipartConfig;
9  import javax.servlet.annotation.WebServlet;
10 import javax.servlet.http.HttpServlet;
11 import javax.servlet.http.HttpServletRequest;
12 import javax.servlet.http.HttpServletResponse;
13 import javax.servlet.http.Part;
14
15 @WebServlet(name="Upload", urlPatterns={"/Upload"})
16 @MultipartConfig(location="/tmp")
17 public class Upload extends HttpServlet {
18
19     @Override
20     protected void doPost(HttpServletRequest request,
21                           HttpServletResponse response)
22         throws ServletException, IOException {
23         Part p = request.getPart("desc");
24         BufferedReader r = new BufferedReader(
25             new InputStreamReader(p.getInputStream()));
26         String desc = r.readLine();
27         request.setAttribute("desc", desc);
28         p = request.getPart("data");
29
30         p.write("DATAFILE");
31         RequestDispatcher rd =
32             request.getRequestDispatcher("acknowledge.jsp");
33         rd.forward(request, response);
34     }
35 }
```

### Code 9-7 Simple Servlet With Multipart Handling

## Handling Multipart Forms

---

Code 9-8 below show a form that interacts with the servlet in Code 9-7 above, supplying the form fields expected by that servlet.

```
1  <form action="Upload" enctype="multipart/form-data" method="post">
2      Description: <input type="text" name="desc"/>
3      File: <input type="file" name="data"/>
4      <input type="submit" value="Upload"/>
5  </form>
```

### **Code 9-8** Simple Form Supporting the Multipart Servlet

## Summary

This module introduced these topics:

- The lifecycle of a servlet
- The threading model of a servlet
- Filters and how to apply them to groups of servlets or JSPs
- Handling multipart form data

## Summary

---

## Module 10

---

# More Options for the Model

---

## Objectives

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

- Understand the nature of the model as a macro-pattern
- Implement persistent storage for your web applications using JDBC or Java Persistence API

## Relevance

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

- What goes into a model component? Is it actually just a simple piece of the application, or more than that?
- How can you provide persistent storage for data in your web applications?



## 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 Platform, Enterprise Edition 5 API Specification. Available:  
<http://java.sun.com/javaee/6/docs/api>
- Java Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans™ IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- The Common Gateway Interface. [Online]. Available:  
<http://hoohoo.ncsa.uiuc.edu/cgi/>

## The Model as a Macro-Pattern

In the MVC architecture pattern, the model component embodies a great deal of functionality. Here are some of the responsibilities that fall to the model as it has been discussed so far:

- Present an interface to the controller
- Implement the business logic
- Present a JavaBeans compliant view of the data to the view component, in a form that is convenient for the view to use.

It should be immediately apparent that these three responsibilities might benefit from separation, according to the guidelines of OO. After all, they are different responsibilities, and are even likely to change independently.

If you consider the second item in that list “Implement the business logic,” it will also be abundantly clear that this is not likely to be a single object either. Implementing the business logic would be almost the entire program for a batch, or command-line, based system. It is likely to require implementation of many individual business processes and rules, many of which are entirely unrelated, and therefore should be in separate classes. Also, almost every real system will require data to be read from, and stored back into, a database. Some systems will require connection to other types of external systems, perhaps message-driven or built around a third party product such as SAP. Code used to interface with such systems is non-trivial, and should not properly be part of the same classes as those performing the individual business operations. This is part of the reasoning behind the creation of session, entity, and message driven beans and many other frameworks intended to simplify solving some of these issues.

So far in this class, you have been presented with the idea of a model as a single component, even a single class. However, it is important to recognize that in your daily work, it will be a component made of many other objects. Those objects will themselves be designed using other design patterns. For the most part, these issues, and the patterns that relate to them, are beyond the scope of this course

For the purpose of learning about servlets and JSP, the most important thing to consider is how the view will interface to the model. You should also consider the interface between the controller and the model, although this is arguably less important.

## The View Helper Pattern

According to OO principles, any object is designed best if designed without considering the way in which it will be used. This is because if it were to be designed specifically to support a certain usage style, and that usage style changes, then the object must be changed too. A foundational premise of OO is that good design minimizes the consequences of change.

So, it follows that the model should not be designed with consideration to the way that the view will need to use it. Of course, the view is being coded using EL, which is not a programming language. Furthermore, there might need to be specific pieces of formatting or other translation between the “pure” model and the presentation provided by the view. Again, EL is usually not a good solution to this requirement.

To address this issue, it is common to create a model that ignores the view, but then create an interface component that goes between the view and the model. This component can act as an adapter, providing a desirable interface for the view (with the data organized conveniently for the way the view will use it) and connecting to the pure model. It can also perform translations, such as might be required for the purpose of supporting different locales in an internationalized system.

This component is called a View Helper, and is the basis of the design pattern with that name.

## Database and Resource Access

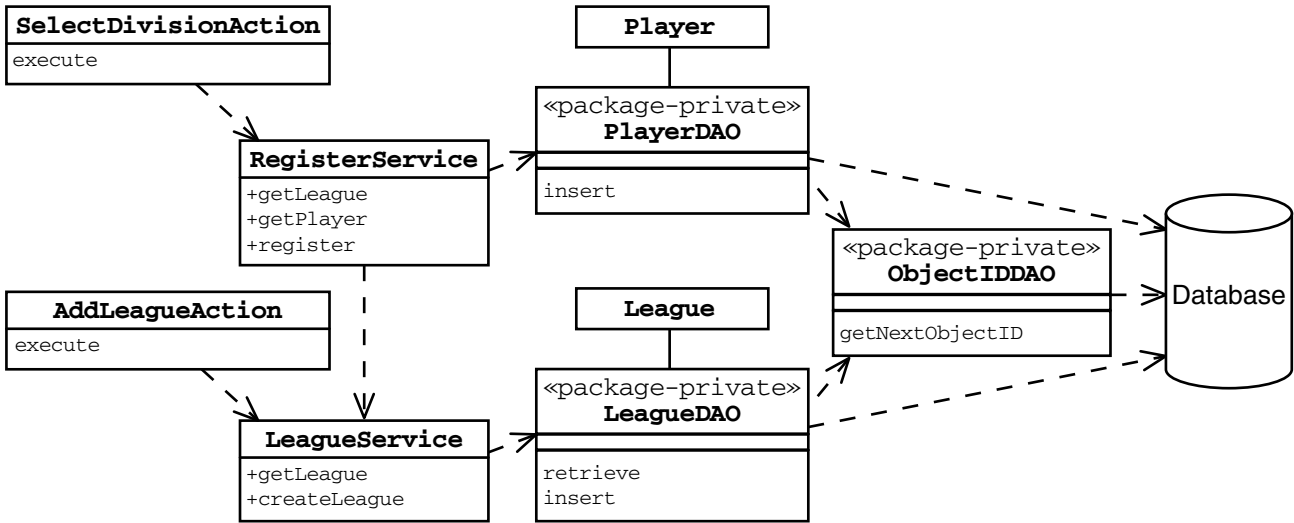
Virtually all practical web applications will require access to a database or other external support system. Good OO requires that the elements of the model that handle this should be separated from the elements of the model that represent the domain objects and the elements that perform business operations on those objects.

The desire to separate these concerns leads to the Data Access Object pattern, also known as DAO.

### Data Access Object (DAO) Pattern

The DAO pattern eases maintenance of applications that use databases by separating the business logic from the data access (data storage) logic. The data access implementation (perhaps JDBC technology calls) is encapsulated in DAO classes. The DAO pattern permits the business logic and the data access logic to change independently, increasing the flexibility of your application. For example, if the database schema changes, you only need to change the DAO methods, and not the business services or the domain objects.

The business 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* (that is, only available to classes in the same package). The business services use the DAOs to perform the data access logic, such as inserting an object or retrieving an object. Figure 10-1 exemplifies a sports league application.



**Figure 10-1** The DAO Pattern Exemplified Using a Sports League

When a registration request is processed, the `SelectDivisionAction` component 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.

## DAO Pattern Advantages

The DAO pattern has the following advantages:

- Domain objects and persistence logic are now separate.  
The domain objects do not need to know how their persistence is handled.
- The data access objects promote reuse and flexibility in changing the system.

New domain objects and business services can be constructed that reuse the data access logic in the DAO classes.

- Developers writing other clients, whether servlets or regular client code, 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 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 facilitates changes to front-end technologies.

Using the DAO pattern, you can change the web tier components easily without impacting the existing data access logic.

- This design facilitates changes to back-end technologies.

Using the DAO pattern, the resource tier can change independently of the front-end tiers. For example, you can 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.

## JDBC API

The JDBC API is the Java technology API for interacting with a database management system (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 learn techniques for designing and developing a web application in which the JDBC API 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 completing 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*. This is directly supported by the JDBC v2 API.

## Developing a Web Application Using a Database

This section describes the implementation and configuration details when developing a web application using a database.

### Traditional Approaches to Database Connections

In Java technology, the most basic mechanism for obtaining a database connection is to use the static method `java.sql.DriverManager.getConnection`. Obtaining an exclusive connection is expensive, and the total number of concurrent connections is usually limited. Because connections are needed in many places in a Java EE application, using this approach would create performance problems in a real application.

A better option is to create a pool of connections and store this pool in the servlet context. Connections are retrieved from the pool by a request, used, and then returned to the pool so they can be reused by another request. This solution avoids the constant creation and destruction of database connections and enhances performance in the application. However, because this connection pool is stored in the servlet context, the pool would only be available to web components (such as servlets and action classes) and not to business tier components that need access to the database.

### Using a DataSource and the Java Naming and Directory Interface API

A `DataSource` is a resource that contains the information needed to connect to an underlying database (typically the database URL, driver, user name, and password). All Java EE application servers are required to provide a naming service into which resources such as `DataSource` can be stored. The naming service can be queried using the Java Naming and Directory Interface™ (JNDI) APIs. Behind the scenes, the server will maintain a pool of connections. You can use the `DataSource` to retrieve a connection from this pool.

Benefits of using a `DataSource` for your database access include:

- Less work – You do not have to write a connection pool mechanism.
- Portability – Your code will work across all application servers.



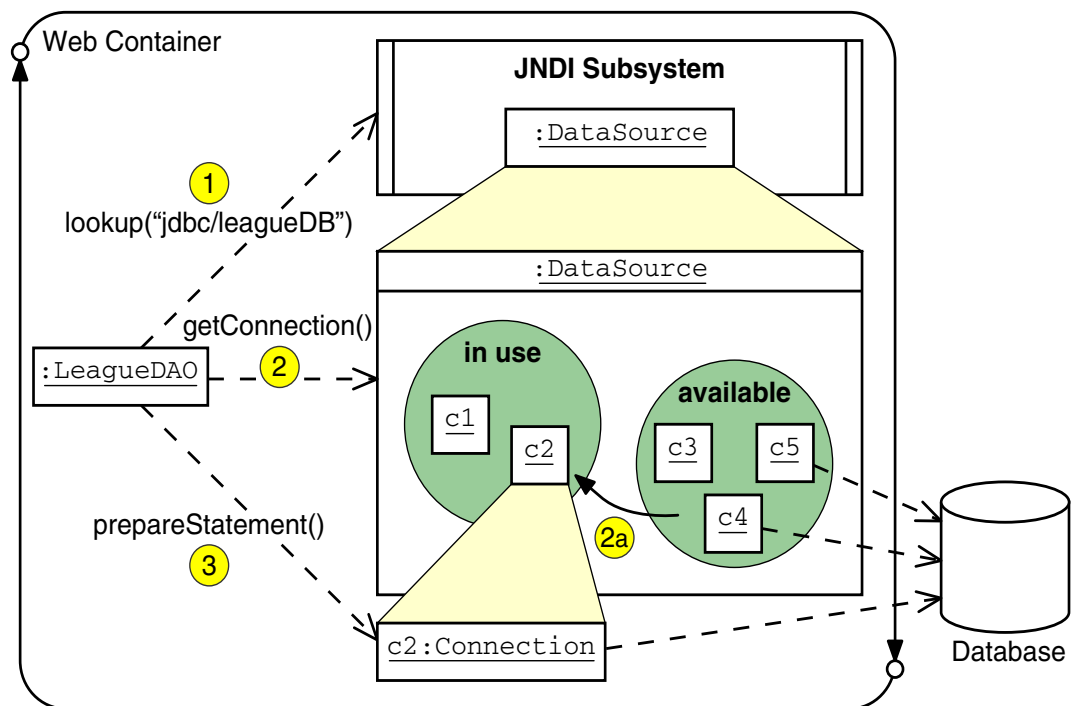
When a component needs a database connection, the following steps occur:

1. The component performs a JNDI lookup on the naming service to retrieve the `DataSource`.
2. The component calls the `getConnection` method on the `DataSource` to retrieve a database connection.
3. The component uses the JDBC API to communicate with the database by way of the connection.



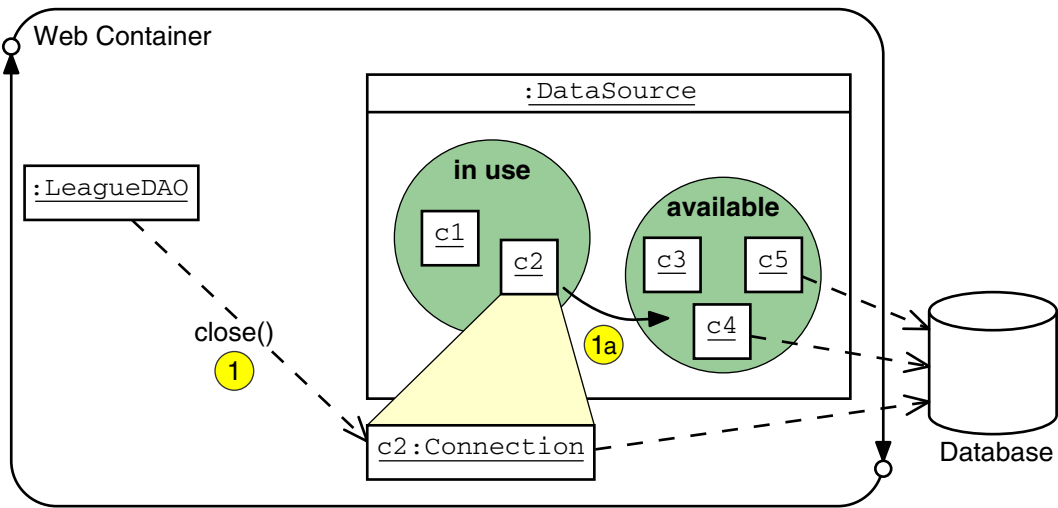
**Note** – The `DataSource` interface is defined in the `javax.sql` extension package and is not part of the standard `java.sql` package.

Figure 10-2 shows how `DataSource` is used in an application.



**Figure 10-2** Application `DataSource` Use

When the DAO component has finished using the database connection, then the DAO should call the `close` method on the connection object. The `close` method on the connection object returns the connection back to the `DataSource` object, which marks the connection as available. Figure 10-3 shows this process.



**Figure 10-3** Closing the Connection Returns it to the `DataSource` Object

The `DataSource` interface is defined in the `javax.sql` extension package and is not part of the standard `java.sql` package. Figure 10-4 shows the API.

<code>javax.sql.DataSource</code>
<code>getConnection(): java.sql.Connection</code>
<code>getConnection(username: String, password: String): java.sql.Connection</code>

**Figure 10-4** The `DataSource` API

Prior to using a `DataSource`, a component must locate the resource in the JNDI namespace. A JNDI Context is used to perform the lookup. Code 10-1 shows locating a `DataSource` using JNDI in the `LeagueDAO` class.

```
52 Context ctx = new InitialContext();
53 if ( ctx == null ) {
54     throw new RuntimeException("JNDI Context could not be found.");
55 }
56 ds = (DataSource)ctx.lookup("java:comp/env/jdbc/leagueDB");
57 if ( ds == null ) {
58     throw new RuntimeException("DataSource could not be found.");
59 }
```

### Code 10-1 Locating a `DataSource` Using JNDI

On line 52, the JNDI context is created using the container's default information. On line 56, the JNDI lookup is performed for the `DataSource` JNDI name.



**Note** – You can use the JNDI API to store other resources used by applications, such as environment entries, Java Message Service (JMS) resources, JavaMail™ API resources, and so on. The JNDI API is the preferred method for configuring parameters, as opposed to context configuration parameters, because information configured in the JNDI namespace is available to every component in the application. For example, a helper class does not need access to the `ServletContext` object to locate information stored in JNDI namespace.

## Executing SQL

The `DataSource` provides a method that is used to get another object called a connection. The connection provides a method that is used to prepare another object that embodies a template SQL statement. Into that statement, you can then inject variable data if applicable. Then, using the completed statement along with its data, you can execute the statement and obtain a result set.

The result set allows you to iterate over the rows of the result, and extract individual fields for use in the program.

Sample code might look like this:

```

ds = (DataSource)ctx.lookup(
    "java:comp/env/jdbc/leagueDB");
if ( ds == null ) {
    throw new RuntimeException("DataSource not found.");
}
connection = ds.getConnection();
stmt = connection.prepareStatement(
    "SELECT * FROM LEAGUE_TABLE WHERE 'year'=?1 and
    'season'=?2");
stmt.setInt(1, year);
stmt.setString(2, season);
results = stmt.executeQuery();
while ( results.next() ) {
    int objectID = results.getInt("LID");
    num_of_rows++;
    if ( num_of_rows > 1 ) {
        throw new SQLException("Too many rows.");
    }
    int theYear = results.getInt("year"),
    String theSeason = results.getString("season"),
}

```



**Note** – It is important to avoid constructing an SQL statement using simple concatenation of strings if the arguments include any data provided directly or indirectly by the user. Such an approach leads to SQL injection vulnerabilities. Use of the PreparedStatement, ensures that the SQL system can distinguish data from instructions.

## Configuring a DataSource and the JNDI API

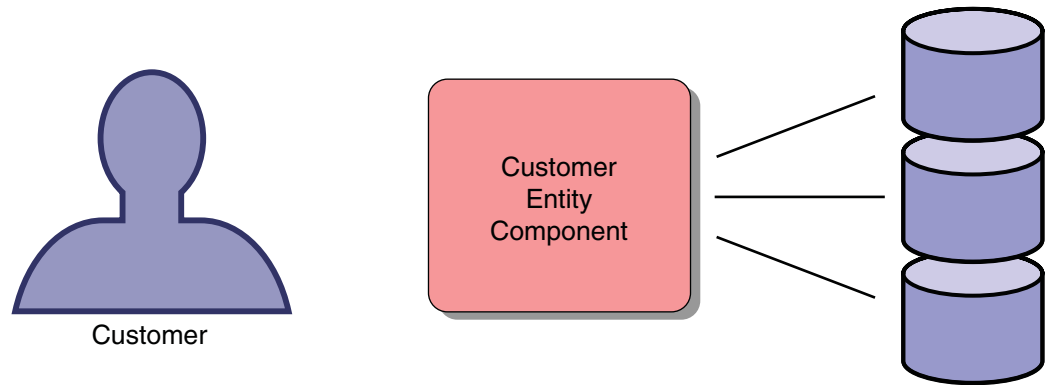
The DataSource reference must be created in the application server, and published by name in the JNDI lookup service. In principle, three things must be configured:

- The database and the tables/schema for the application
- A connection pool
- An entry in the JNDI service that references the connection pool

The method of achieving these steps varies between application servers, and the lab exercise for this module will guide you through that process for the GlassFish™/NetBeans combination.

# Object Relational Mapping Software

The accepted standard persistence mechanism used in most enterprise applications is a relational database. OO program design and relational database table structure might not organize data in the exact same structure. A Java business domain object can encompass partial data from a single database table or include data from multiple tables depending on the normalization of the relational database.



**Figure 10-5** Business Concept-to-Object Oriented Domain Object and Relational Database Mapping

Writing code to translate from an object-oriented domain scheme to a relational database scheme can be a time consuming endeavor. Object Relational Mapping (ORM) software attempts to provide this mapping to OO software developers without requiring much or any coding. Often just configuration information in the form of code annotations or XML configuration files are supplied to ORM software. Examples of existing ORM software are EclipseLink and Hibernate.

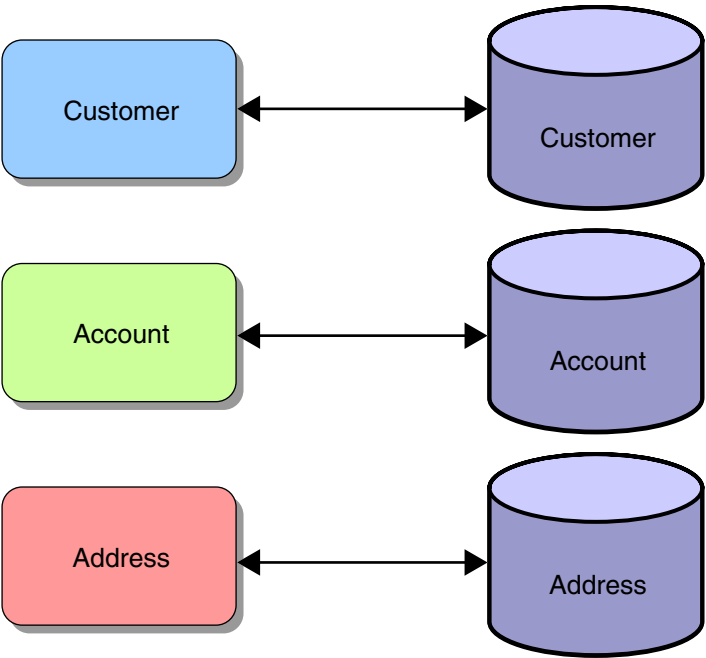


---

**Note** – EclipseLink, <http://www.eclipse.org/eclipselink/>, is the continuation of the open source version of Toplink. Oracle donated the source code for Toplink, a Java Persistence API 1.0 provider, to the Eclipse Foundation. EclipseLink is the reference implementation for the Java Persistence API 2.0 specification.

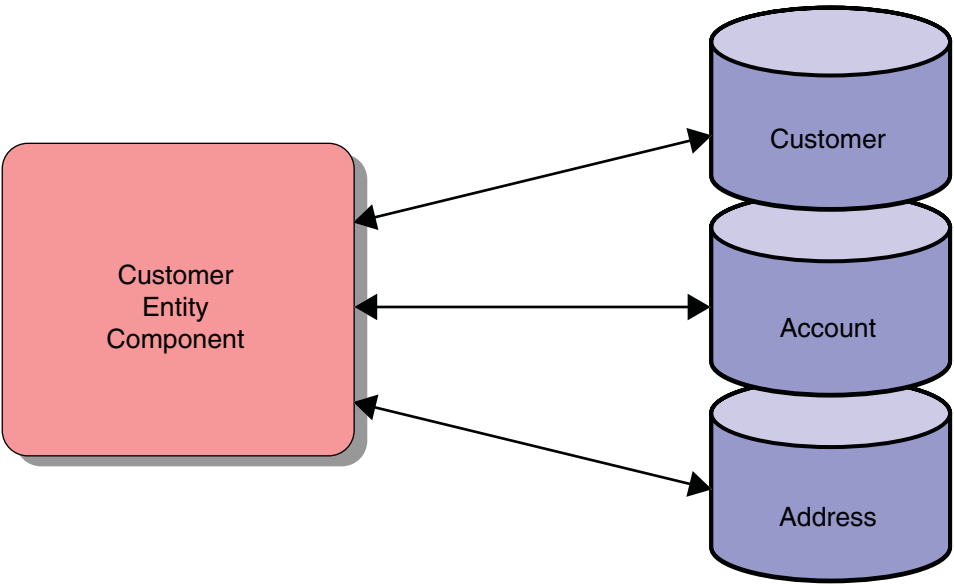
---

The most basic ORM software supports a simple mapping of Java objects to database tables, as shown in Figure 10-6.



**Figure 10-6** One-to-One Mapping of Java Objects to Relational Tables

Modern ORM software has the ability to map Java objects to database table structures that do not have a simple one-to-one mapping, as shown in Figure 10-7.



**Figure 10-7** One-to-Many Mapping of a Java Object to Relational Tables

Java EE application servers provide EJB 2.1 Container Managed Persistence (CMP) EJBs as the standard ORM technique. Other ORM methods, such as Java Data Objects (JDO), also exist. The Java Persistence API attempts to combine the best features from existing ORM technologies while removing the application server requirement of EJB 2.1 CMP EJBs.

The Java Persistence API specification is just a specification for an API and life cycle behavior; it is not usable ORM software. A Java Persistence API provider is required to use the Java Persistence API. Two examples of a Java Persistence API provider are EclipseLink and Hibernate.

## Java Persistence API Structure

The Java Persistence API splits the problem of persistence into a number of key elements, each of which requires configuration if the system is to work. These elements are:

- An entity class, defined by the programmer, that presents the object-oriented view of the business entity.
- A mapping between the fields of the entity class and the schema of the persistent storage. This can be provided using annotations on fields within the entity class along with default mapping rules, and may be modified where necessary by additional annotations.
- Mapping that specifies the persistence implementation (recall that Java Persistence API is simply a specification), the JNDI `DataSource`, and other configuration elements of the implementation. This is provided using a file called `persistence.xml`.
- A JNDI lookup service entry that maps the published name of the `DataSource` to an actual `DataSource`. This is provided in a container-specific manner as discussed earlier.
- A `DataSource` that is connected to a connection pool. This has also been discussed earlier, and is also provided in a container-specific way.
- A connection pool that is connected to the database. As before, this is container specific and was introduced earlier.
- The actual database, along with the necessary tables and schema to support the application. Again, this is container specific.



# Entity Class Requirements

This section provides an overview of the tasks required to define a Java Persistence API entity class. To define an entity class, you are required to perform the following tasks:

- Declare the entity class.
- Verify and override the default mapping.

The following subsections describe these tasks in more detail.

## Declaring the Entity Class

The following steps outline a process you can use to declare an entity class.

1. Collect information required to declare the entity class.  
This step involves identifying the application domain object (identified in the object oriented analysis and design phase of application development) that you want to persist.
  - Use the domain object name as the class name.
  - Use the domain object field names and data types as the field names and types of entity classes.
2. Declare a public Java technology class.  
The class must not be final, and no methods of the entity class can be final. The class can be concrete or abstract. The class can not be an inner class.
3. If an entity instance is to be passed by value as a detached object through a remote interface, then ensure the entity class implements the `Serializable` interface.
4. Annotate the class with the `javax.persistence.Entity` annotation.
5. Declare the attributes of the entity class.  
Attributes must not have public visibility. They can have private, protected, or package visibility. Clients must not attempt to access an entity classes' attributes directly.
6. You can optionally declare a set of public getter and setter methods for every attribute declared in the previous step.

## Entity Class Requirements

7. Annotate the primary key field or the getter method that corresponds to the primary key column with the `Id` annotation.
8. Declare a public or protected no-arg constructor that takes no parameters. The container uses this constructor to create instances of the entity class. The class can have additional constructors.

Code 10-2 below shows an example of an entity class.

```

1  package entity;
2
3  import java.io.Serializable;
4  import javax.persistence.Column;
5  import javax.persistence.Entity;
6  import javax.persistence.GeneratedValue;
7  import javax.persistence.GenerationType;
8  import javax.persistence.Id;
9
10 @Entity
11 public class Airplane implements Serializable {
12     private static final long serialVersionUID = 1L;
13     @Id
14     @GeneratedValue(strategy = GenerationType.AUTO)
15     private String id;
16     @Column private String type;
17     @Column private int engines;
18
19     public String getId() { return id; }
20
21     public void setId(String id) { this.id = id; }
22
23     public String getType(){ return type; }
24
25     public void setType(String type){ this.type = type; }
26
27     public int getEngines() { return engines; }
28
29     public void setEngines(int engines){ this.engines = engines; }
30
31 }

```

**Code 10-2** Entity Class Code Example

## Verifying and Overriding the Default Mapping

The following table outlines the default standard relationship between entity classes and relational databases.

**Table 10-1** Mapping Object Tier Elements to Database Tier Elements

Object Tier Element	Database Tier Element
Entity class	Database table
Field of entity class	Database table column
Entity instance	Database table record

The default database table name can be overridden using the `Table` annotation. The Java Persistence API assigns the name of the entity class as the default name to the database table. This is exemplified below in Code 10-3:

```
@Entity
@Table(name = "Cust")    //Cust is the name of the database table
public class Client {
    //...
}
```

### Code 10-3 Table Mapping

The Java Persistence API assigns the name of the entity class property (or field) name as the default name to the database table column. Use the `Column` annotation to override the default setting, as shown below in Code 10-4:

```
@Entity
@Table(name = "Cust")
public class Client {
    @Column(name = "cname")
    private String clientName;
    //...
}
```

### Code 10-4 Column Mapping

You have several options available for generating the values of primary keys. The simplest of these is to use the autogeneration feature of the container, as shown in the following example in Code 10-5:

## Entity Class Requirements

---

```
@Entity
@Table(name = "Cust")
public class Client {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int clientReference;
    //...
}
```

### Code 10-5 Primary Key Generation

When using `@GeneratedValue(strategy = GenerationType.AUTO)`, Toplink creates a sequence table in the relational database. Other Java Persistence API providers can use different techniques to create unique IDs.



---

**Note** – This discussion examined the optional task of overriding the default settings. For the vast majority of situations, using the defaults should be sufficient if no previous database tables exist.

---

# Life Cycle and Operational Characteristics of Entity Components

Besides entity classes, there are several key concepts that must be understood to begin leveraging the Java Persistence API. They are:

- **Persistence Unit**  
Configuration information in the form of an XML file and a bundle of classes that are controlled by the Java Persistence API provider.
- **Entity manager**  
An entity manager is the service object that manages the entity life cycle instances. This is the core object that a Java developer uses to create, retrieve, update, and delete data from a database.
- **Persistence context**  
Every entity manager is associated with a persistence context. A persistence context is a set of entity instances in which, for any persistent entity identity, there is a unique entity instance.
- **Persistent identity**  
The persistent identity is a unique value used by the container to map the entity instance to the corresponding record in the database. Persistent identity should not be confused with Java object identity.

## Persistence Units

A persistence unit is a collection of entity classes stored in a EJB-JAR, WAR, or JAR archive along with a `persistence.xml` file. The key concept to understand at this point is that a persistence unit defines what entity classes are controlled by an entity manager. A persistence unit is limited to a single `DataSource`.

## The `persistence.xml` file

Configuration of a persistent unit is controlled by an XML configuration file named `persistence.xml` which does the following:

- Configures which classes make up a persistence unit
- Defines the base of a persistence unit by its location
- Specifies the `DataSource` used

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="BrokerTool-ejb" transaction-type="JTA">
    <jta-data-source>StockMarket</jta-data-source>
    <jar-file>BrokerLibrary.jar</jar-file>
    <properties/>
  </persistence-unit>
</persistence>
```

## The Persistence Context

A persistence context can be thought of as a working copy of a persistence unit. Several persistence contexts using the same persistence entity can be active at the same time. A persistence context has the following characteristics:

- Typically lasts the duration of a transaction
- Limits entity instances to a single instance per persistent identity
- Has a management API known as the entity manager

## The EntityManager

An entity manager provides methods to control events of a persistence context and the life cycle of entity instances in a persistence context. An entity manager has the following characteristics:

- Provides operations, such as `flush()`, `find()`, and `createQuery()`, to control a persistence context
- Replaces some of the functionality of home interfaces in EJB 2.1 entity beans

An entity manager can be obtained using dependency injection in managed classes.

```
@PersistenceContext private EntityManager em;
```

## Entity Instance Management

When a reference to an entity manager has been obtained, you use its methods to marshall entity instance data into and out of the database.



**Note** – Many exceptions that can be thrown by the `EntityManager` methods are indirect subclasses of `java.lang.RuntimeException`. Integrated Development Environments (IDE)s and compilers do not require try catch blocks around lines of code that cause a `RuntimeException` subclass, such as `javax.persistence.EntityExistsException`, to be thrown.

## EntityManager Injection

The entity manager object is created by injection. That is, the variable is declared and annotated, but not initialized explicitly. Instead, the annotation results in a value being provided for the field at runtime.

To create an entity manager in your class, you can use this code:

```
import javax.persistence.PersistenceContext;

[...]

// Declaring a field in the class:
@PersistenceContext EntityManager em;
```

Following this, the value of the variable `em` at runtime should not be `null`, and should provide an entity manager instance that can be used to interact with the database.

## EntityManager Methods

The entity manager can perform typical database operations, but does so without using SQL, and by referring to objects, rather than rows and tables. So, key methods in the entity manager are:

- `find` - used to load a specific entity, identified by its primary key, into the application (Read)
- `createQuery` - used to prepare for a select-like lookup of potentially multiple objects. When the query is created it must subsequently be executed, for example using `getResultList()` (Read)
- `persist` - used to store an object in the database (Create)
- `merge` - used to write an object back into the database (Update)
- `remove` - used to remove an object from the database (Delete)

## User Transactions

Operations that update the database typically require transactions to take effect, and to provide control over the database behavior where concurrent access is occurring.

A `UserTransaction` object can be used to demarcate the boundaries of a transaction in your code. The `UserTransaction` object is created by injection as was the `EntityManager`. It can be created as a field in a class like this:

```
import javax.transaction.UserTransaction;

[...]
// Declaring a field in the class:
@Resource UserTransaction utx;
```

Then the `UserTransaction` methods `begin`, `commit`, `rollback`, and `setRollbackOnly` can be used to provide control over the transaction and its scope.



# Java Persistence API Example

Below is a code fragment from a simple servlet that, with a JSP view component, a JSP error view component, and an Airplane entity class, uses the persistence facilities to add new entries to a database of airplane types.

```

1  @WebServlet(name="AddServlet", urlPatterns={"/AddServlet"})
2  public class AddServlet extends HttpServlet {
3      @PersistenceContext EntityManager em;
4      @Resource UserTransaction utx;
5
6      protected void processRequest(HttpServletRequest request,
7          HttpServletResponse response)
8          throws ServletException, IOException {
9          try {
10             assert em != null;
11             String id = request.getParameter("id");
12             String type = request.getParameter("type");
13             int engines =
14                 Integer.parseInt(request.getParameter("engines"));
15             Airplane a = new Airplane();
16             a.setId(id);
17             a.setType(type);
18             a.setEngines(engines);
19             utx.begin();
20             em.persist(a);
21             utx.commit();
22
23             List <Airplane> l = em.createQuery(
24                 "select a from Airplane a").getResultList();
25             request.setAttribute("airplaneList", l);
26             RequestDispatcher rd = request.getRequestDispatcher(
27                 "ListView.jsp");
28             rd.forward(request, response);
29         } catch (Exception ex) {
30             request.setAttribute("exception", ex);
31             RequestDispatcher rd = request.getRequestDispatcher(
32                 "ExceptionView.jsp");
33             rd.forward(request, response);
34         }
35     }
36 }

```

## Code 10-6 Persistence API Example

## Summary

This module introduced the following topics:

- The nature of the model as a macro-pattern
- Techniques for implementing persistent storage for your web applications

## Module 11

---

# Asynchronous Servlets and Clients

---

## Objectives

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

- Use the Asynchronous Servlet mechanism
- Use JavaScript to send an HTTP request from a client
- Process an HTTP response entirely in JavaScript
- Combine these techniques to create the effect of server-push

## Relevance



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

- What happens if the response to an HTTP request cannot be made until a message is received from a third party? Must the servlet thread be blocked indefinitely?
- Suppose dozens of clients are cooperating, for example, in a chat room. Must a request from every client be blocked waiting until one client triggers an update?

## 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 Platform, Enterprise Edition 5 API Specification. Available:  
<http://java.sun.com/javaee/6/docs/api>
- Java Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans™ IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- The Common Gateway Interface. [Online]. Available:  
<http://hoohoo.ncsa.uiuc.edu/cgi/>

## Asynchronous Servlets

In some situations, the processing of a client request might take a long time. In such a situation servlet threads that are managed by the web container are bound up in the processing of the long-running servlet and are unavailable to other requests. If this occurs because the request cannot be satisfied until a separate trigger condition occurs, then those threads are simply blocked and unavailable to service other requests. This might impose an excessive restriction on the application's scalability. This discussion is not concerned with the user becoming impatient while waiting for a response, which is a valid, but different concern.

Java EE 6 adds the option to perform request processing asynchronously. If this is done, then the servlet execution thread can be freed up to service requests from other clients, and the generation of the response may be performed in another thread, for example the thread that creates the trigger condition that allows the response to be prepared.



---

**Note** – The asynchronous servlet API is unrelated to asynchronous JavaScript and XML (AJAX) techniques. In AJAX, JavaScript code makes synchronous calls to servlets, but those calls are not triggered by page requests. A brief examination of some rudiments of AJAX follows later in this module.

---

## Separating Request Receipt from Response Generation

To allow for separation of request from response generation, the servlet API provides a class called `AsyncContext`. A servlet that wishes to hand off response generation to another thread can obtain an `AsyncContext` object and pass this to another thread, perhaps using a queue.

The `AsyncContext` object provides access to the original `HttpServletRequest` and `HttpServletResponse` objects. This allows processing of the request and generation of the response. Additionally, the `AsyncContext` allows for a forwarding mechanism. This does not use the `RequestDispatcher`, is achieved using one of several dispatch methods in the `AsyncContext` itself.

## Asynchronous Servlet Example

The code in Code 11-1 below shows a simple asynchronous servlet that simply passes all work immediately to another thread via the `handler.addJob(ac)` call.

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    AsyncContext ac = request.startAsync();
    handler.addJob(ac);
}
```

### Code 11-1 Creating an AsyncContext and Passing it to Another Thread

To receive the request, the handler class mentioned in Code 11-1 must be created. This class is shown in Code 11-2 below:

```
1  package service;
2
3  import java.util.ArrayList;
4  import java.util.Iterator;
5  import java.util.List;
6  import javax.servlet.AsyncContext;
7  import javax.servlet.ServletException;
8
9  public class Handler implements Runnable {
10
11     private static final Handler self;
12     private static final String[] words = {
13         "long", "short", "big", "small", "clever", "foolish", "tidy",
14         "disorganized"
15     };
16     private Thread myThread;
17     private boolean stop = false;
18     private List<AsyncContext> queue = new
19     ArrayList<AsyncContext>(100);
20     private List<AsyncContext> inProgress = new
21     ArrayList<AsyncContext>(100);
22
23     static {
24         self = new Handler();
25     }
26 }
```

Continued...

## Asynchronous Servlets

---

```
23
24     private Handler() {
25     }
26
27     public static Handler getHandler() {
28         return self;
29     }
30
31     public void stop() {
32         stop = true;
33     }
34
35     public synchronized void addJob(AsyncContext job) {
36         queue.add(job);
37         System.out.println("Added a job, queue length is " +
queue.size());
38         if (myThread == null) {
39             System.out.println("Started handler thread");
40             myThread = new Thread(this);
41             myThread.start();
42         }
43     }
44
45     public void run() {
46         while (!stop) {
47             System.out.println("Handler loop running");
48             try {
49                 Thread.sleep(5000 + ((int) (Math.random() * 5000)));
50             } catch (InterruptedException ex) {
51                 ex.printStackTrace();
52             }
53             // check the "queue"
54             synchronized (this) {
55                 List<AsyncContext> l = inProgress;
56                 l.clear();
57                 inProgress = queue;
58                 queue = l;
59             }

```

Continued...



```

60         if (!inProgress.isEmpty()) {
61             System.out.println("Queue contains " +
inProgress.size() + " elements");
62             // Create the new information
63             String value = "The word is: " + words[(int)
(Math.random() * words.length)] + " and the number is: " + Math.random()
* 1000000;
64             Iterator<AsyncContext> iac = inProgress.iterator();
65             while (iac.hasNext()) {
66                 // generate responses
67                 AsyncContext ac = iac.next();
68                 ServletRequest req = ac.getRequest();
69                 req.setAttribute("value", value);
70                 System.out.println("Handler dispatching to a
jsp");
71                 ac.dispatch("asyncResponse.jsp");
72             }
73         }
74     }
75 }
76 System.out.println("Stopping handler thread");
77 }
78 }

```

### Code 11-2 Example Handler For Asynchronous Servlet Requests

The Handler class shown in Code 11-2 above is not the only way to process asynchronous servlet requests, but is a simple example. Notice that the class behaves as a singleton, so that all servlets can submit to the same queue. The class also maintains two Lists that are used to provide a queuing mechanism for the jobs.

The service thread runs at intervals. In this example, these intervals are deliberately created using a call to `Thread.sleep`. In a real situation, these delays would result from external factors, such as the availability of data. Each time the service thread runs, it prepares some result data and then selects in turn each of the `AsyncContext` items that have been added to a queue, and dispatches these to a simple JSP for presentation of the results.

The JSP that presents the results is entirely conventional, and simply picks up the attribute called `value` from the request attribute using EL in the normal way.

## Forwarding and Filtering

Asynchronous handlers are permitted to dispatch or forward their processing to other servlets or JSPs, including those that are not asynchronous (that is, they do not carry the `asyncSupported=true` element in the annotation). This allows a standard response page to be used as a view for both synchronous and asynchronous responses.



---

**Note** – JSPs should be used as views, and as such simply present data. Given that an asynchronous servlet can dispatch to a synchronous element, it is not necessary to mark a JSP as asynchronous.

---

Filters can be invoked on asynchronous invocations too. If the `AsyncContext.dispatch` call is used on a matching URL, the dispatcher type will be `ASYNC`.

# Asynchronous Listeners

Given an `AsyncContext` object, you can add one or more listeners to it using the `addListener(AsyncListener)` method. These listeners will be called as a consequence of particular events in the life of the `AsyncContext`. These events are: starting, and completion, of asynchronous processing, error situations, and timeout.

The `AsyncListener` interface must be implemented by the listener that will be notified when these situations arise. The interface defines four listener methods:

- `onComplete(AsyncEvent)`
- `onError(AsyncEvent)`
- `onTimeout(AsyncEvent)`
- `onStartAsync(AsyncEvent)`

Event processing and delivery is essentially the same as for any other event in any other Java API, although the order of delivery is guaranteed to be the order of registration of listeners.

## Asynchronous JavaScript Clients

A very common behavior in modern web clients is to allow a single page to use JavaScript to fetch elements of data from the server and update the displayed page without fetching a whole new page or redrawing the whole page. This is the basis of the programming approach known as AJAX.

In pure AJAX, the request and response is formatted as an XML document. However, the technique can be used quite easily with simple GET requests from the client eliciting fragments of HTML from the server.

In these Asynchronous JavaScript approaches, JavaScript code on the client is used to modify elements of the existing page based on a combination of the user's interactions, local client-side processing, and information retrieved from the server. Remember that in a normal HTTP request, an entire page is fetched, but in this approach, JavaScript code fetches a small amount of data which is then integrated into the existing page. Usually the server's response is really very small, for example, it might be just a fragment of HTML, as in the following example:

### Simple Asynchronous Client Example

```
1  <html>
2    <head>
3      <title>Async JavaScript Page</title>
4    </head>
5    <body>
6      <h1>Hello World!</h1>
7      <form>
8        <input id="inputField" type="text" onkeyup="doUpdate()">
9      </form>
10     <div id="wisdom">
11   </div>
12   <script type="text/javascript">
13     wisdomTag = document.getElementById("wisdom");
14     inputTag = document.getElementById("inputField");
15
16     function doUpdate() {
17       var req;
18       if (window.XMLHttpRequest) {
19         req = new XMLHttpRequest();
20       } else if (window.ActiveXObject) {
```

```
21         req = new ActiveXObject("Microsoft.XMLHTTP");
22     } else {
23         alert("AJAX not supported");
24     }
25
26     req.onreadystatechange = function() {
27         if (req.readyState == 4 && req.status == 200) {
28             wisdomTag.innerHTML = req.responseText;
29         }
30     }
31
32     text = inputTag.value;
33     req.open("GET", "update.jsp?text=" + text, true);
34     req.send(null);
35 }
36 </script>
37 </body>
38 </html>
```

### Code 11-3 HTML Page With Asynchronous JavaScript

In this example, the basic display consists of the heading “Hello World!” followed by an input field and a blank, but labeled, division. When a key has been entered into the input field, the JavaScript method `doUpdate` (lines 16 through 35) will be invoked (see “`onkeyup`” at line 8). That method triggers an HTTP request, and the response to that request is subsequently embedded in the labeled division without refreshing the page as a whole.

The cornerstone of sending an HTTP request in JavaScript is embodied in the `req` reference. Generally, this will refer to an object of the `XMLHttpRequest` type, however, in some older Microsoft browsers, it will instead be an Active X component. This is prepared using the code between lines 18 and 24. Notice that some older browsers cannot perform this type of behavior.

To send the request, the two calls `open` and `send` are used to prepare the HTTP request and send it. These are invoked on lines 33 and 34. Notice that line 33 constructs an HTTP GET request that will request a URL based on the same URL as the one from which this page was loaded, but with the particular page `update.jsp` and with the parameter `text` set to the value of whatever was typed into the input field.

To receive the response, a listener must be configured so that it will be triggered when data are received. This is the purpose of the assignment on line 26, which prepares an anonymous function and arranges that it will be invoked when the “state” of the connection changes. In particular, when the status is 200 and the readyState is 4, then a valid response has been received from the server. The fact that this response handling listener is triggered by the arrival of the response is what makes this approach qualify as asynchronous. The update is not triggered by the sequence of operations in the program, but by the response when it happens to arrive.

In this example, a second, very simple JSP is used to provide the server response. That JSP is shown below, and is triggered by the request for `update.jsp` from the JavaScript code.

```
1 <p>You asked ${param["text"]}</p>
2 <p>It's <%= new java.util.Date() %> </p>
```

#### **Code 11-4** JSP to Service Asynchronous Requests

Notice that the JSP that services the asynchronous request does not preset a well-formed HTML page. Instead, this example simply provides a small fragment of dynamically computed HTML. This HTML will be embedded into the original page by the JavaScript.

## Server Response Content in an AJAX System

Although it is very easy to use a simple fragment of HTML as the response to a request of this sort, you can also send structured data using XML, or JavaScript Object Notation (JSON). Details of these, and of their processing, is beyond the scope of this module which serves only as an introduction to these asynchronous ideas, but are covered in other Sun courses, including DTJ2121, *The JavaScript Language and AJAX for Java Developers*.

## Combining Asynchronous Servlets With Asynchronous JavaScript

Although each of these techniques serves a purpose on its own, the two can be used together to create something akin to a server-push environment.

Consider a page containing JavaScript code that makes an asynchronous request for update from the server, and a server-side implementation that uses the asynchronous servlet techniques. Now, if the server does not choose to respond for several minutes, the user is not inconvenienced, as the body of the page is operating normally. Similarly, because of the use of asynchronous servlet execution, the resource load on the server is minimized. Consequently, the response may be sent at a time convenient to the server, and the effect of server-push is achieved in an architecturally manageable way.

## Summary

This module introduced the following topics:

- The Asynchronous Servlet mechanism
- Using JavaScript to send an HTTP request from a client
- Processing an HTTP response entirely in JavaScript
- Combining these techniques to create the effect of server-push



## Module 12

---

# Implementing Security

---

## Objectives

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

- Describe a common failure mode in security
- Require that a user log in before accessing specific pages in your web application
- Describe the Java EE security model
- Require SSL encrypted communication for certain URLs or servlets

## Relevance

**Discussion** – The following question is relevant to understanding what technologies are available for developing web applications and the limitations of those technologies:

- If your application uses data that are private to your company or your users, how can you be sure that malicious users cannot inappropriately access or modify those data?

## 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 Platform, Enterprise Edition 5 API Specification. Available:  
<http://java.sun.com/javaee/6/docs/api>
- Java Platform, Enterprise Edition Blueprints. [Online]. Available:  
<http://java.sun.com/j2ee/blueprints/>
- Sun Microsystems software download page for the Sun Java System Application Server. [Online]. Available:  
<http://www.sun.com/download/>
- NetBeans™ IDE download page. [Online]. Available:  
<http://www.netbeans.org/downloads/>
- HTTP RFC #2616 [Online]. Available:  
<http://www.ietf.org/rfc/rfc2616.txt>
- The Common Gateway Interface. [Online]. Available:  
<http://hoohoo.ncsa.uiuc.edu/cgi/>

## Security Considerations

Every application that is accessible over the web must consider security. Your site must be protected from attack, the private data of your site's users must be kept confidential, and your site must also protect the browsers and computers used to access your site.

Unfortunately, these considerations are never complete, and security must always be viewed as a level of protection, not absolute protection. Recognizing that, the administration of the system should include monitoring, so that when an attack occurs, it will be noticed, and might be stopped or disconnected before major damage occurs.

Because of the sheer complexity, scope, and importance of the security problem, consider using specialist security experts in the creation of a web-based program. However, each team member should also have as much understanding of major issues as is practical.

This module introduces the following key points:

- Confusion of code and data'
- Encryption of data in transit over the network
- Authentication and authorization of users

## Confusion of Code and Data

A very substantial proportion of attacks against computer systems are variations on the seemingly simple idea of tricking the system into executing as code something that was passed into the system in the guise of data.

Consider that data do not control the operation of the machine, and further that the machine must allow data to be provided from outside—from the user. You can see that systems do not need to prevent data from entering the system. However, if the system's willingness to allow data in can be tricked, it might be possible to provide code in the guise of data. This will bypass the security restrictions of the system and might be catastrophic.

### SQL Injection Example

One of the simplest examples of this security failure is in a simple form of SQL injection. Consider an HTML form that submits two fields, `itemCode` and `quantity`, to your application so that the user may add items to a shopping cart. Your application lets these data in, because they are just that, data. Clearly the application will not be useful if the user cannot add items to the shopping cart.

Now consider that the application wishes to verify availability of these items. You might want to execute an SQL query to determine the quantity on hand of this particular item code. Your application might take the text of the item code, provided by the user, and paste it into an SQL statement like this:

```
SELECT count from ITEMTABLE where itemcode="XXXXXX";
```

In this case, the `XXXXXX` would be replaced using the data provided by the user in the field of the form.

This looks fine so far, but consider what happens if the user provides the following as the `itemCode` field in the form:

```
unk"; DROP TABLE ITEMTABLE;
```

Now the result of pasting this "data" into the query is this:

```
SELECT count from ITEMTABLE where itemcode="unk"; DROP  
TABLE ITEMTABLE;";
```

In this example, the effect is to cause a significant denial of service, requiring you to restore your `ITEMTABLE` from a backup. Worse things could be done, such as inserting a new administrative user into a database.

Clearly this attack required some knowledge of the tables that are in use, but in many cases, major vendors use standard tables for administrative purposes, and applications often give such information away when they show exception stack traces that often include information of this sort.

## Other Code as Data Attacks

Many other attacks exist that are dependent on the confusion of code as data. One such is “shellcoding” where an overly long data string is sent to a program so as to overflow the stack. The result is that some part of the data becomes executed as code. This attack cannot happen inside a Java program, though it might be launched against the host operating system. Also, a degree of protection is achieved by configuring the OS to prohibit execution of code on the stack, a feature that is now available in all current operating systems, though it was slow in arriving in Windows.

Another well-known example, which is used in web applications and attacks the client’s computer rather than the web server, is known as cross-site scripting. In this attack, a malicious user provides data in a URL in an email. The victim clicks on the link which passes the text of the URL to the genuine host. The host then reflects that data back to the victim’s computer, where it is processed by the victim’s browser. If that data includes JavaScript, then the victim’s browser will execute that code with the trust the victim places in your site.

## Preventing Code as Data Attacks

To minimize the threat of code as data attacks, it is important to filter *all* input, and reject anything that does not conform to a narrowly defined criterion of input that makes sense in the current context. For example, itemcodes might be formed with uppercase letters and numbers only. In this case, your system should reject any itemcode that does not match this form. Importantly, you should not send the user a rejection message saying “We’re sorry, but BLAH#%32 is not a valid item code” because to send the offending data back to the user would provide the basis of a cross-site scripting attack. Consider if the item code had contained JavaScript.



---

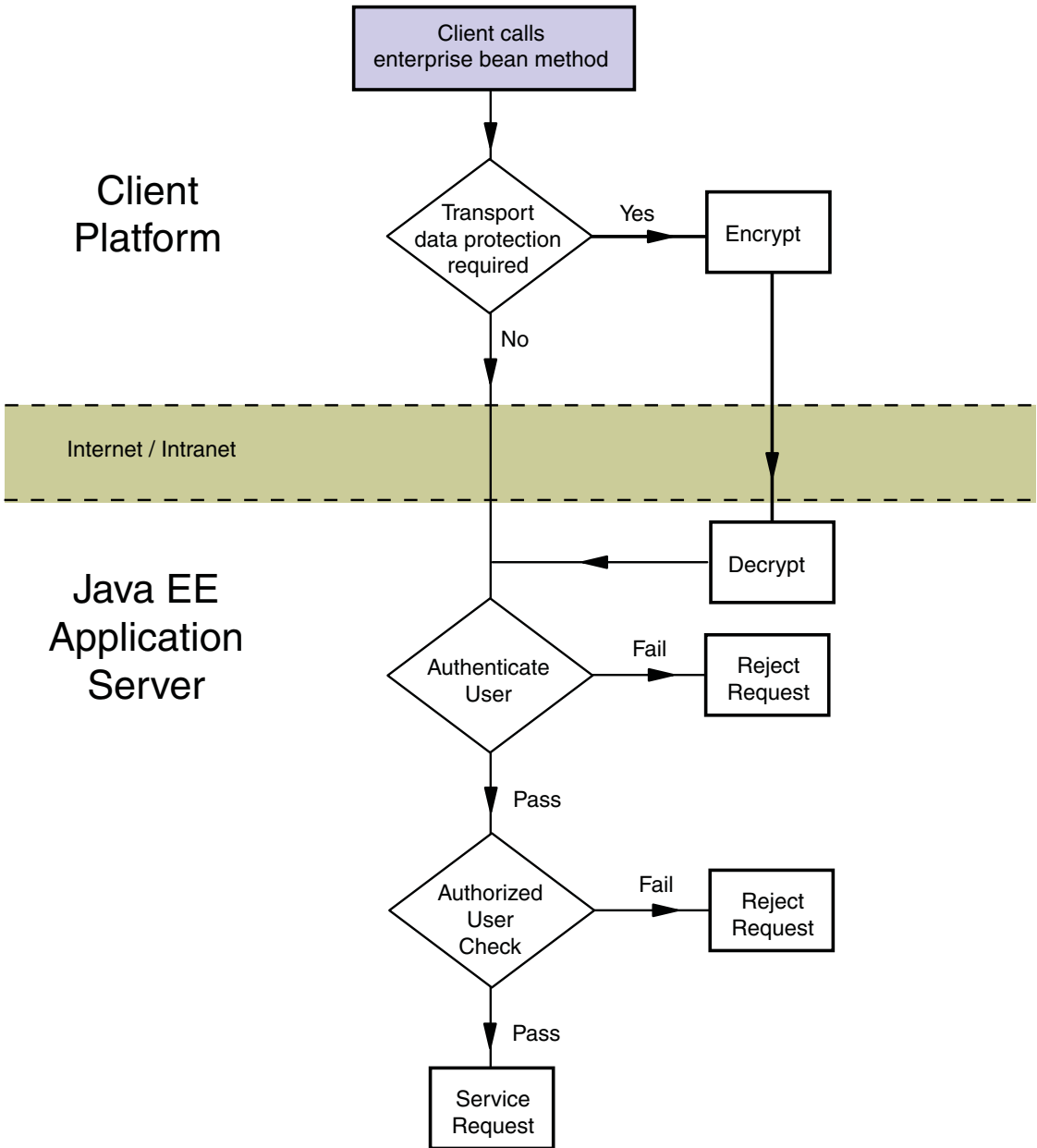
**Note** – In the specific case of SQL injection, the use of prepared statements, as outlined in “Executing SQL” on page 10-13 prevents the confusion of code and data.

---

# Authentication and Authorization

The application usually needs to be able to identify the user, decide what operations the user is allowed to perform, and maintain the confidentiality and the integrity of the data that is in transit.

Figure 12-1 illustrates the runtime security interventions performed by the Java EE infrastructure when responding to a method invocation by a client.



**Figure 12-1** Security Interventions



These security interventions can be described as follows:

- **Transport data protection**  
Transport security ensures the confidentiality and integrity of the information transported across the network. Systems use a combination of certificate exchange and encryption to achieve this security.  
EJB, web-client, and application-client containers are required to support both Secure Sockets Layer (SSL 3.0) protocol and the related Transport Layer Security (TLS 1.0) protocol for the Internet Inter-Orb Protocol (IIOP).
- **Caller identification and authentication**  
Identification and authentication is the process of determining the user's identity, and verifying that they are who they claim to be. Most web-based systems use password challenges to identify the user at the client. Certificates are common for the purpose of identifying the server host.
- **Authorization and access control implementation for resources**  
Authorization determines who is allowed to do what. Authorization is the process of allocating permissions to authenticated users. These permissions give a user access to resources and services in the native domain. Access control is the mechanism used to enforce the decisions of authorization.

A primary goal of the Java EE platform is to relieve the application developer from the details of these security mechanisms and to facilitate the secure deployment of an application in diverse environments. When you use the Java EE security model, you increase development efficiency and portability.

The following list summarizes the characteristics of Java EE security. Java EE security:

- Contains no reference to the real security infrastructure, so that the application may be portable.
- Can use declarative or programmatic authorization controls.
- Uses JAAS, the Java Authentication and Authorization Service to map the application security policy to the security domain of the deployment environment. In this way, platform independence is essentially preserved even though it becomes deployment specific. JAAS implementations can provide for simple username-password pairs stored in a database, up to more complex systems, such as Kerberos or Active Directory. In any case, the verification interface is standardized, although the means of adding new users is not.
- Provides end-to-end security, so that security credentials are gathered in one part of the system, but are available to all parts. For example, if the user is using a web browser to interact with a servlet-based application, and that servlet application calls enterprise beans, then all the components (servlets and enterprise beans) that are invoked in a particular user interaction *see* the same user.

The rest of this module focuses on the authentication and authorization aspects of the Java EE technology security model.

# Authenticating the Caller

Caller authentication is the process of verifying what the user's identity is, and consists of the following two steps.

1. Determine the identity claimed by the user
2. Verify that the user is who they claim to be (Authenticate the user)

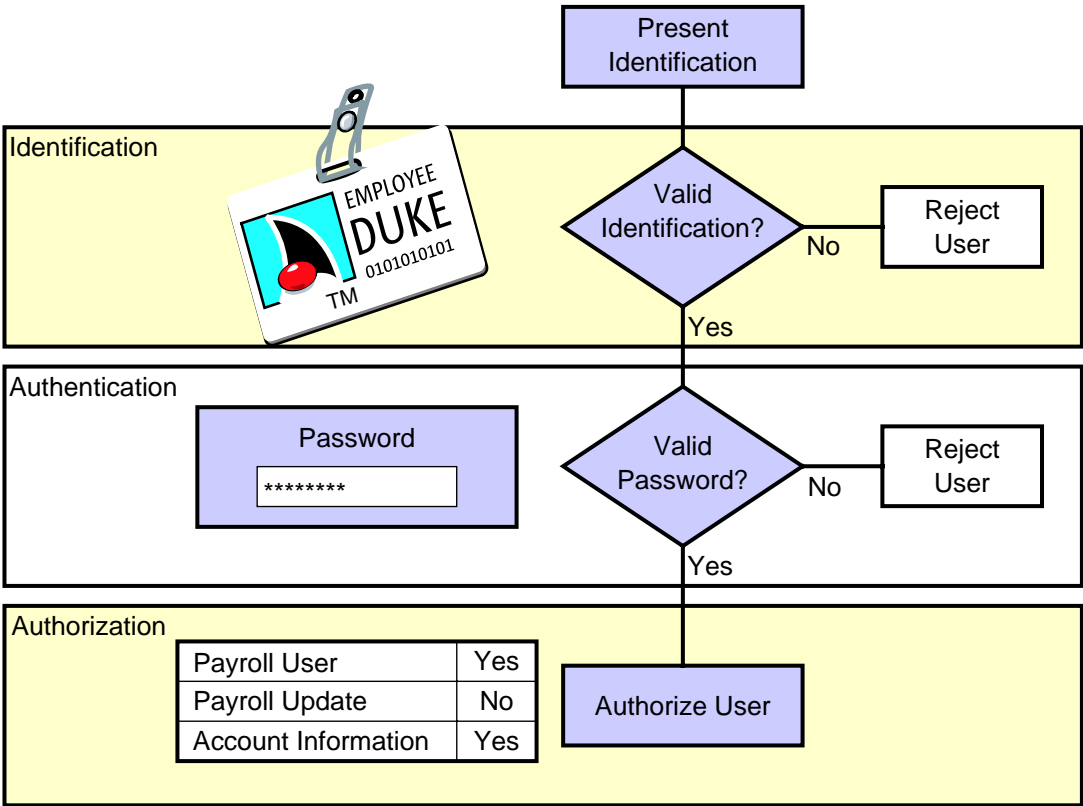
Java EE web containers can use password challenges or client certificates in any of four standardized forms to implement this task. The four approaches are:

- **BASIC** – HTTP basic authentication. This mechanism is standardized, based on a user name and password challenge, and requires the use of encryption to maintain wire security, and suffers from two significant disadvantages: No logout mechanism is supported, and the login prompt is determined by the browser, not by the application. Because of these disadvantages, this mode of authentication is not popular.
- **DIGEST** – This mechanism modifies the HTTP basic approach by providing encryption of the password in transit. However, the other disadvantages remain and it too is unpopular.
- **FORM** – This is the standard mechanism for most web applications that wish to authenticate their users. The application programmer provides a special HTML form that prompts for user name and password, and these data are provided to the web container which performs the authentication. Logout is supported because the user is identified by a session, and if the session is invalidated, the authentication is lost along with it. Critically, this method must be used over an encrypted connection if the user name and password are not to be sent over the wire unprotected.
- **CLIENT CERTIFICATE** – In this mechanism, the client is required to provide a public key certificate, signed by an authority trusted by the web container. The web container then challenges the client browser to digitally sign a piece of data. If the signature provided by the client verifies against the certificate, then the identity of the client has been authenticated. This can be a very secure means of authentication provided that the certificates and keys are properly handled, but is usually only used in business-to-business transactions.

# Caller Authentication

The authentication is performed by checking the provided data (such as username and password) against some kind of record (a “truststore” or (credentials database”). The Java EE specification leaves the choice of the caller authentication mechanism to the deployer.

At minimum, a user interface must be provided to the user to request entry of the data used to identify and authenticate the user. To assist the deployer, the application server can provide authentication mechanisms of various levels of sophistication. Figure 12-2 illustrates a user name-password based authentication scheme.



**Figure 12-2** User Name-Password Based Authentication

To allow for maximum flexibility, all Java EE web containers must interface with the Java Authentication and Authorization Service (JAAS) API. JAAS enables services to authenticate and enforce access controls upon users. JAAS implements a Java technology version of the standard Pluggable Authentication Module (PAM) framework, and extends the access control architecture of the Java platform to support user-based authorization.

## Establishing User Identities

The process of caller authentication requires that users of an application be known in advance to the security system. The Java EE specification recognizes the following two types of user identities:

- Principals
- Roles

### Principal

A principal is an authenticated user in the application security domain. That is, a principal is identifiable to, and can be authenticated by, a JAAS authentication mechanism deployed in the web container. The content and format of the principal name and the authentication data are dependent upon the JAAS implementation in use.

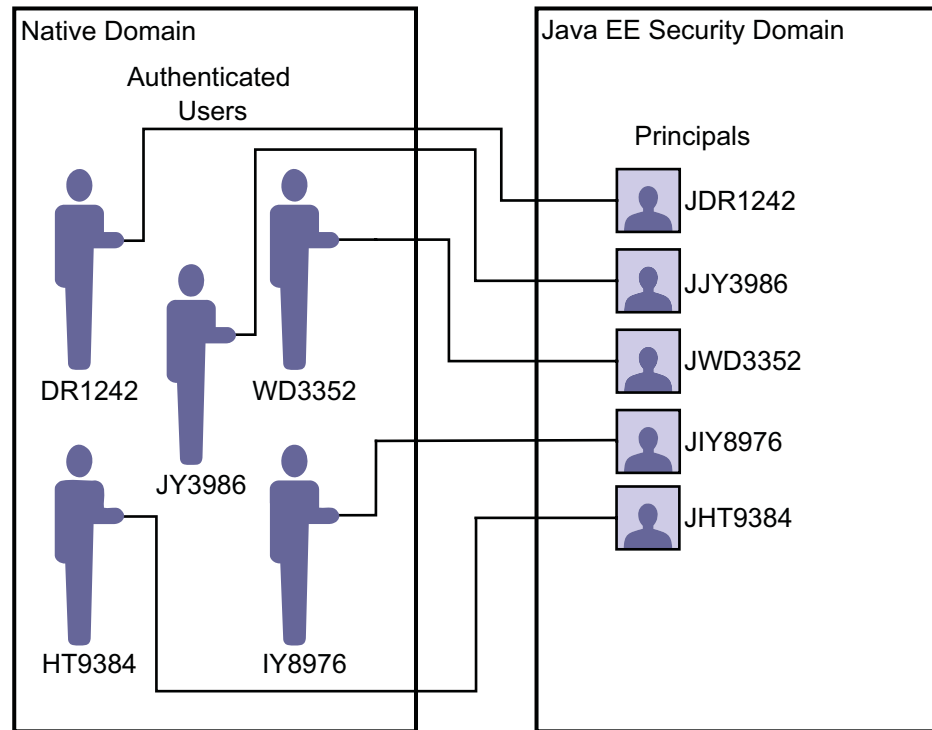


---

**Note** – It might be more appropriate to think of this the other way round. That is, the JAAS implementation is often selected to interoperate with an existing security database.

---

Once the user has logged in, the identifying information is available inside the application in the form of a Principal. Figure 12-3 shows the mapping of users in the native security domain to principals in the Java EE security domain.



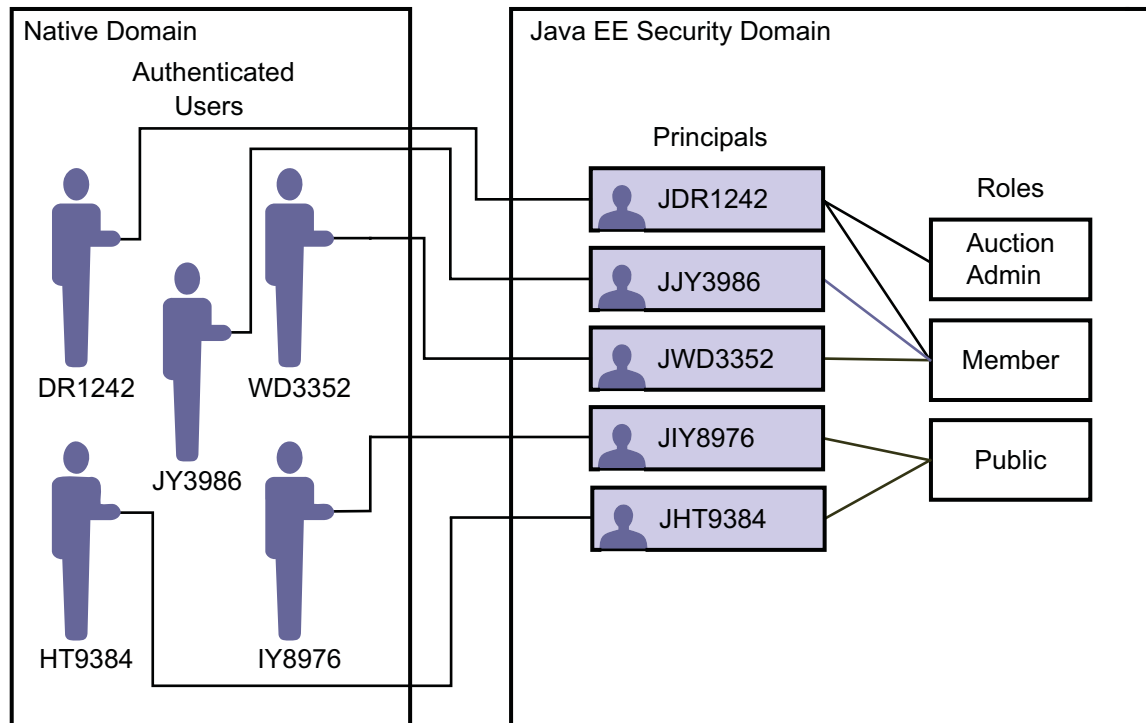
**Figure 12-3** Mapping Native Domain Users to Principals

Mapping users in the native domain to principals is the role of the JAAS implementation selected.

## Role

When writing an application, the users, and the principals to which they will map, are usually not known. Nevertheless, you must design a security model that will specify that certain categories of user will have certain rights and be denied other rights. This notion of the category of a user is called a *role*, and application security models are generally described in terms of these roles. A role is used to maintain a group of principals sharing a common set of permissions. The users that belong to a given role are defined as part of the deployment of the application, and the user-to-role mappings are administered using application server vendor-supplied tools.

Figure 12-4 shows the mapping of principals to roles.



**Figure 12-4** Mapping Principals to Roles

Each role has a set of permissions associated with it. A permission provides any member of the role with the required authorization to execute certain behavior. That behavior might be loading of a JSP or servlet, or in the larger Java EE environment, might be the execution of a method on an Enterprise Java Bean (EJB). The application assembler or the deployer can use either metadata annotations or the DD to associate permissions with a role name. For more details on method permissions, see “Declaring Permission Requirements” on page 12-20.

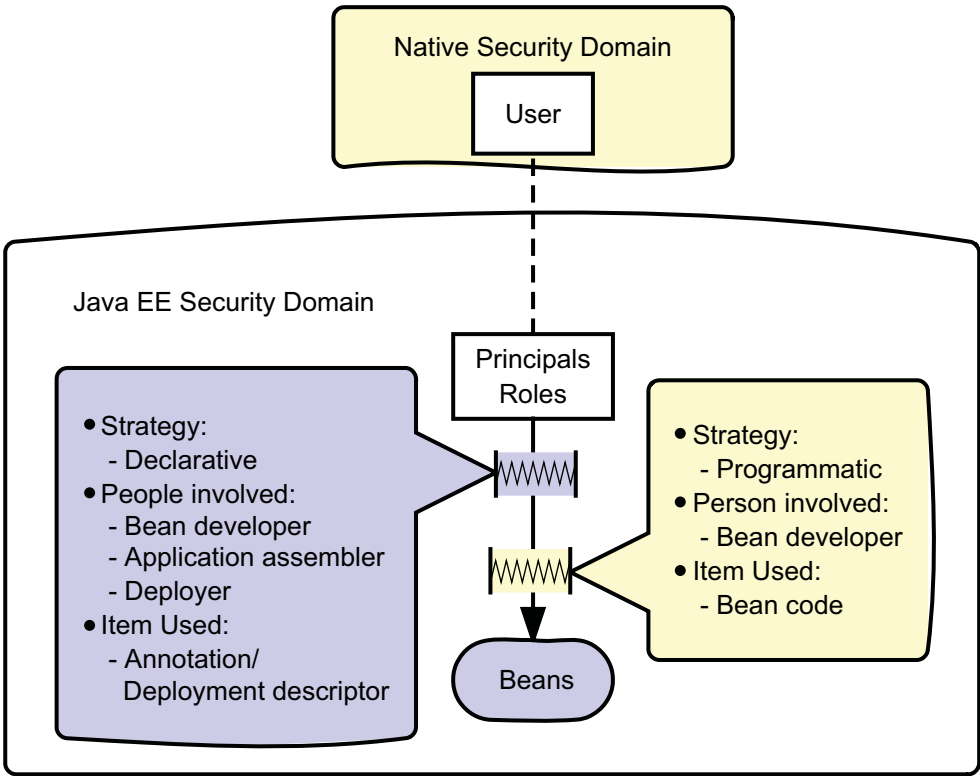
Most real application servers work with security infrastructures that have their own concept of *group* or *role*. The JAAS mechanism allows the deployer to make mappings between groups or roles in the application domain and Java EE technology roles, which simplifies the mapping process and generally reduces the chance of error.

The Java EE specification designates the task of mapping principals to roles to the deployer.

# Examining the Java EE Authorization Strategies

The primary purpose of the Java EE security model is to control access to business services and resources in the application. The Java EE security model provides two complementary strategies for access control: programmatic access control and declarative access control. Both strategies assume that the user has been authenticated by the application server, and the roles of which the user is a member can therefore be determined by the web container.

Figure 12-5 provides a high-level comparison of the Java EE authorization strategies.



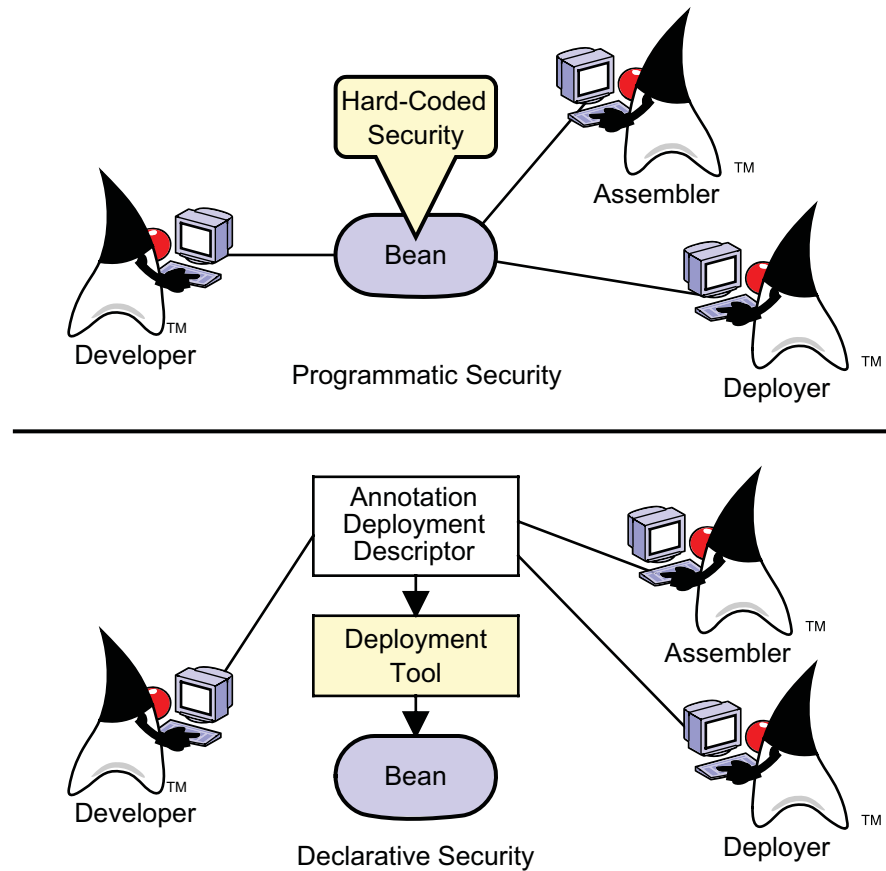
**Figure 12-5** Overview of Java EE Authorization Strategies

With programmatic access control, access control decisions are made in program code. With declarative access control, access control decisions are made by the container, based on information in annotations or deployment descriptors. Prior to the Servlet 3.0 specification, the setting of declarative security polices was restricted to the DD.



Using programmatic access control in the Java EE security model does not necessarily mean that you write code that interacts with the target (legacy) security infrastructure. Even with programmatic authorization, the application server remains responsible for authentication against a credentials database and the extraction of the roles assigned to particular principals. You have only limited programmatic control of authorization.

Figure 12-6 illustrates the differences between the programmatic authorization strategy and the declarative authorization strategy.



**Figure 12-6** Differences Between the Authorization Strategies

## Using Declarative Authorization

Declarative authorization can be configured by the programmer, assembler, or deployer. Usually the programmer sets up some initial security properties when developing or testing the web application or its components. The assembler and deployer can then modify these properties according to the needs of the application as a whole. The declarative security policy can be defined in the servlets using annotations or the DD. The use of vendor-supplied packaging and deployment tools can be helpful here, because they can provide a basic, graphical representation of the security policy.

Declarative authorization for web applications involves the following tasks:

- Collection of user credentials into a credentials database
- Declaration of roles
- Mapping users to roles
- Specification of role requirements for access to URLs

The following sections address these requirements.

### Creating a Credentials Database

Creating the collection of user credentials is entirely dependent on the web container in use. The lab for this module will show you the most basic way to achieve this in the NetBeans/GlassFish environment you are using.

### Declaring Security Roles

Security roles are declared in the `web.xml` deployment descriptor, using the `<security-role>` element. This element lives at the first level of the `web.xml` file, as a direct child of the `<web-app>` element. In many cases, it is created and edited using tools in your chosen environment, but the specification of the naked XML is simple too, as shown in Code 12-1 below, which shows a fragment of `web.xml` illustrating the declaration of the security role `auction-admin`.

```
1  [...]
2  <security-role>
```

```

3      <description>Needs to be able to create and delete arbitrary
4      auctions.
5      </description>
6      <role-name>auction-admin</role-name>
7  </security-role>
8
9  <security-role>
10     <description>...</description>
11     <role-name>...</role-name>
12 </security-role>

```

### Code 12-1 The security-role Element

Notice that lines 9-12 illustrate that the security-role tag is often repeated to declare multiple roles.

## Mapping Users to Roles

Mapping users to roles is, like the credentials database itself, dependent on the web-container in use. In the case of GlassFish, this occurs in a special deployment descriptor, `sun-web.xml`. This file contains one `<security-role-mapping>` element for each role that will exist, and this element specifies the name of the role and the names of all the users that are members of that role. Code 12-2 below shows an example of this:

```

1  <sun-web-app error-url="">
2  [... other top-level elements]
3      <security-role-mapping>
4          <role-name>Manager</role-name>
5          <principal-name>Trent</principal-name>
6          <principal-name>Bob</principal-name>
7      </security-role-mapping>
8      <security-role-mapping>
9          <role-name>Customer</role-name>
10         <principal-name>Alice</principal-name>
11         <principal-name>Maverick</principal-name>
12 </security-role-mapping>

```

### Code 12-2 Example of Mapping Users to Roles in GlassFish

Notice again that multiple roles may be defined, and each role may have multiple users as members.

## Declaring Permission Requirements

Where a servlet must be restricted to certain roles, this may be specified in the `web.xml` file.




---

**Note** – If you do not specify a permission requirement to a servlet, this results in unrestricted access being granted to that servlet.

---

Code 12-3 below shows an example of restricting a particular URL (in this case `/PrivateServlet`) to users in the role `Trusted`. Notice that the URL may be a wildcard specification, and that multiple roles may be listed. In this example, the `<security-role>` declaration is included for context and completeness.

```

1  <security-constraint>
2      <display-name>Constraint1</display-name>
3      <web-resource-collection>
4          <web-resource-name>Private</web-resource-name>
5          <description/>
6          <url-pattern>/PrivateServlet</url-pattern>
7      </web-resource-collection>
8      <auth-constraint>
9          <description/>
10         <role-name>Trusted</role-name>
11     </auth-constraint>
12 </security-constraint>
13 <security-role>
14     <description/>
15     <role-name>Trusted</role-name>
16 </security-role>

```

**Code 12-3** Example Deployment Descriptor Limiting Access to a URL Pattern to Users in the Role `Trusted`

## Declaring Method Permissions Using Metadata Annotations

You can use an annotation to specify method permissions. Annotation-based permissions differ somewhat from those specified in deployment descriptors, in that they are applied to servlets rather than to URLs listed in collections.

To require that the user be logged in and in the role “Trusted” for access to methods in the bean class, place an annotation immediately outside the servlet class like this:

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "Trusted"))
```

## Using Programmatic Authorization

Programmatic authorization is the responsibility of the bean developer. The following methods in the `HttpServletRequest` support programmatic authorization:

- `boolean isUserInRole(String role)`
- `Principal getUserPrincipal()`

Programmatic authorization is more expressive than the declarative approach, but is more cumbersome to maintain, and because of the additional complexity, more error prone. In particular, declarative authorization controls access at a role level, while programmatic authorization can selectively permit or block access to principals belonging to a role.

### Using the `isUserInRole` Method

A servlet can use the `isCallerInRole` method to verify the role of the caller. The `isUserInRole` method allows recognition of roles without identifying the specific principal.

### Specifying Role References

When you write code using the method `isUserInRole`, the application roles have generally not yet been defined. Because of this, Java EE provides another level of indirection to delay committing to the naming. In the code, you refer to the role by means of another string, called the role reference. Later, the role reference is mapped to a role in the deployed application. This allows separation of the job functions of programmer and application assembler/deployer. This is achieved on a servlet-by-servlet level using the `<security-role-ref>` tag in the `web.xml` deployment descriptor.

Code 12-4 shows that a `security-role-ref` element entry is created by the servlet developer.

```
1  <web-app>
2    <servlet>
3      <servlet-name>AuctionManager</servlet-name>
4      <servlet-class>web.AuctionManagerServlet</servlet-class>
5      <security-role-ref>
6        <description>
7          Needs to be able to create and delete arbitrary auctions
```

```
8         </description>
9         <role-name>auction-admin</role-name>
10        <role-link>Administrator</role-link>
11    </security-role-ref>
12 </servlet>
13 ...
```

**Code 12-4** The security-role-ref Element

Code 12-4 contains a declaration of a security-role-ref element for the role name auction-admin, which is the role name used throughout the application in general. At the time the servlet is written, the role-link name Administrator is used in the code.

## Enforcing Encrypted Transport

Provided the server has been configured with a public key certificate, you can require that communication between client and server be encrypted. This is done in the same security-constraint block that was used to protect a url-pattern that was illustrated in Code 12-3 earlier. In this case, an additional element, `<user-data-constraint>`, will be added, and the resulting DD will look similar to the example in Code 12-5 below.

```
1  [...]
2
3      <security-constraint>
4          <display-name>Constraint1</display-name>
5          <web-resource-collection>
6              <web-resource-name>Private</web-resource-name>
7              <description/>
8              <url-pattern>/PrivateServlet</url-pattern>
9          </web-resource-collection>
10         <auth-constraint>
11             <description/>
12             <role-name>Trusted</role-name>
13         </auth-constraint>
14         <user-data-constraint>
15             <description/>
16             <transport-guarantee>CONFIDENTIAL</transport-guarantee>
17         </user-data-constraint>
18     </security-constraint>
19
```

**Code 12-5** A `<security-constraint>` Block Including a `<user-data-constraint>` Element That Enforces Encrypted Communications

Notice that it is important that you make use of encryption across sufficient pages within your site. If you allow some pages to be browsed over an unencrypted connection while others are encrypted, you might end up exposing cookies, particularly those that identify a session, in such a way that allows an attacker to pretend to be the user that logged in properly earlier.



## Using an Annotation to Mandate Encrypted Transport

An annotation can be used to require encrypted transport. If used, the annotation applies to the servlet, rather than to a collection of URLs. The format of the annotation is:

```
@ServletSecurity(@HttpConstraint(transportGuarantee =  
TransportGuarantee.CONFIDENTIAL) )
```

## Summary

This module introduced the following topics:

- A common failure mode in security
- How to require a user log in before accessing specific pages in your web application
- The nature of the Java EE security model
- Enforcing SSL encrypted communication

## Appendix A

---

# Introducing JavaServer™ Faces Technology

---

## Objectives

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

- Explain the purpose of the JavaServer Faces technology
- Identify the components of a JavaServer Faces web-based application
- Describe the key concepts of the JavaServer Faces technology

## Relevance



**Discussion** – The following questions are relevant to understanding JavaServer Faces technology in the context of present-day business challenges:

- What are the advantages of using JavaServer Faces over custom-framework-driven applications?
- What user interface (UI) components are available in JavaServer Faces?
- What are the uses for available UI components?
- How can JavaServer Faces be leveraged in the process of rapid application development (RAD)?

## Additional Resources



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

- Sun Microsystems, Inc. JavaServer Faces technology reference documentation:  
[<http://java.sun.com/javaee/javaxserverfaces/reference/api/index.html>], accessed 20 March 2006.
- Sun Microsystems, Inc. The Java 2 Platform, Enterprise Edition (J2EE) 1.4 Tutorial:  
[<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/>], accessed 20 March 2006.
- Sun Microsystems, Inc. The Java Platform Enterprise Edition (Java EE) 5 Tutorial:  
[<http://java.sun.com/javaee/5/docs/tutorial/doc/index.html>], accessed 20 March 2006.
- The developer resource center for Sun Java Studio Creator:  
[<http://developers.sun.com/jscreator/>], accessed 6 April 2006.
- Kurniawan, Budi. “Introducing JavaServer Faces”:  
[[http://www.onjava.com/pub/a/onjava/2003/07/30/jsf\\_intro.html](http://www.onjava.com/pub/a/onjava/2003/07/30/jsf_intro.html)], accessed 14 June 2006.
- McGraw-Hill/Osborne. “Introduction to JavaServer Faces, Part 1”:  
[<http://www.devshed.com/c/a/Java/Introduction-to-JavaServer-Faces-1/>], accessed 14 June 2006.

# JavaServer Faces Technology Overview

## Introducing JavaServer Faces Technology

JavaServer Faces technology is a server-side user interface (UI) component technology for developing Java technology-based web applications. JavaServer Faces technology consists of a set of APIs and a JavaServer Pages (JSP) custom tag library.

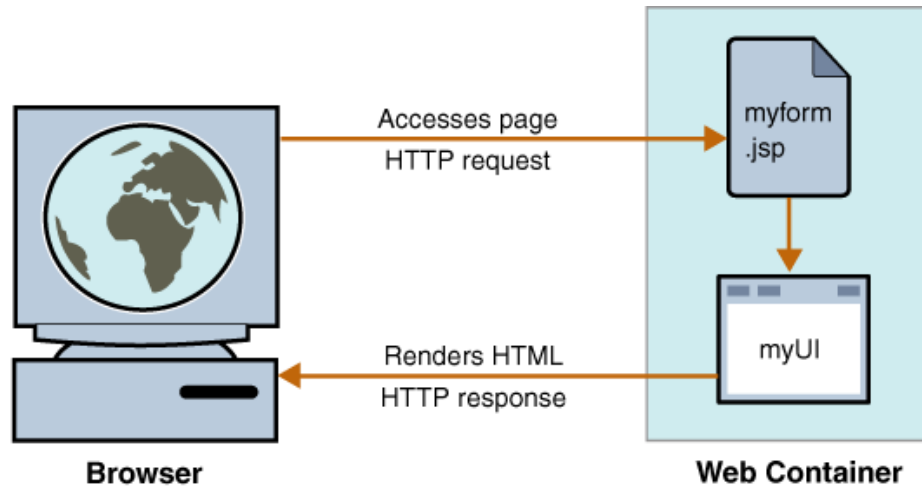
The JavaServer Faces APIs do the following:

- Represent UI components
- Manage the state of UI components
- Handle events
- Handle input validation and conversion
- Define page navigation

- Support internationalization and accessibility

The JSP custom tag library is used to express a JavaServer Faces interface within a JSP page.

Figure 12-7 illustrates the user interface of a JavaServer Faces technology-based web application that runs on the server and renders the response to the client. The JSP page uses the tags defined by the JavaServer Faces technology to express the UI components. These components form the UI of the web application.



**Figure 12-7** JavaServer Faces Web Application

JavaServer Faces technology eases the process of developing and maintaining powerful and dynamic web applications by providing a well-defined programming model and tag libraries. Using this technology, developers can easily perform these activities:

- Assemble reusable UI components in a page
- Connect UI components to application data sources
- Map client-generated events to server-side event handlers
- Manage the state of UI components

This technology enables tool vendors to provide IDEs that are based on the JavaServer Faces technology. It equips developers with capabilities like drag-and-drop methodology that can be used for rapid application development. A good example of a tool that is built to leverage JavaServer Faces technology is Sun Java Studio Creator 2.

## Components of a JavaServer Faces Technology-Based Web Application

A JavaServer Faces technology-based web application contains these components:

- Pages such as JSP pages
- Custom tag libraries
  - For rendering UI components on a page
  - For representing event handlers, validators, and other actions
- UI components represented as stateful objects on the server
- Managed beans that define the properties and functions of the UI components



---

**Note** – The terms “managed beans” and “backing beans” are used interchangeably.

---

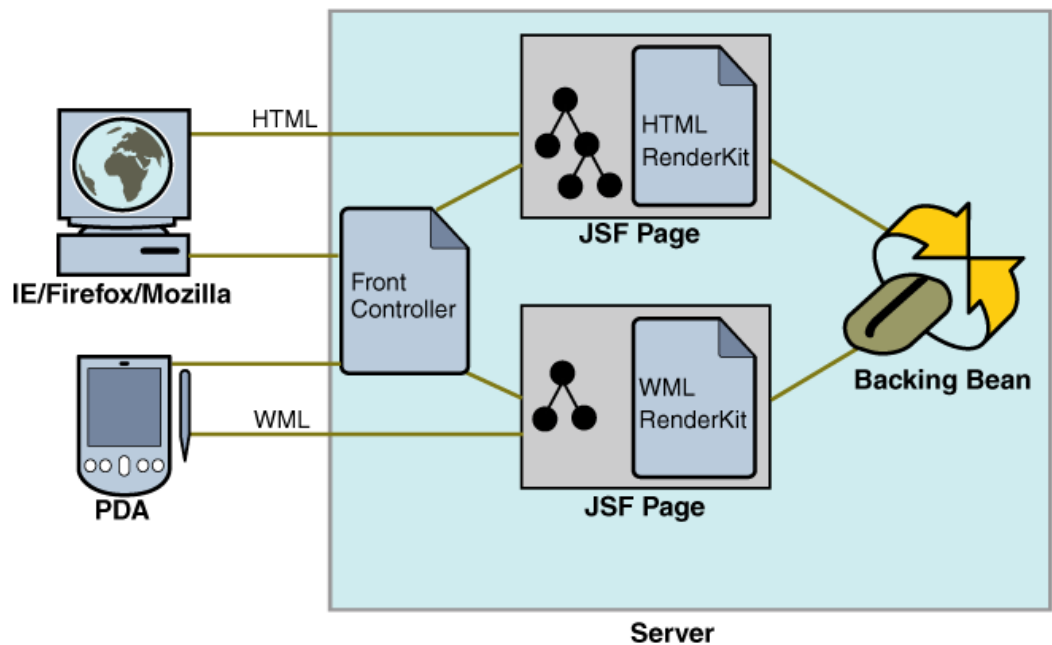
- User input-related components:
  - Validators
  - Converters
  - Event listeners
  - Event handlers
- An application configuration resource file for configuring application resources

A typical JavaServer Faces technology-based web application that uses JSP pages for rendering HTML must include a custom tag library that contains tags for representing the UI components. It must also have a custom tag library for representing other core actions, such as validation and event handling. Both of these tag libraries are provided by the JavaServer Faces implementation.

The custom tag library that represents UI components eliminates the need to hard code UI components in HTML or another markup language, resulting in completely reusable UI components. The custom tag library that represents core actions, also known as the core tag library, makes it easy to register events, validators, and other actions on the UI components.



To understand how JavaServer Faces components are related to each other, you need to understand JavaServer Faces operation. Figure 12-8 illustrates how the JavaServer Faces technology functions.



**Figure 12-8** JavaServer Faces Technology Functions

You can view JavaServer Faces technology as a Model-View-Controller (MVC)-based framework that has its own front controller (known as the Faces servlet) to control client interaction with the server. The front controller also acts as an interface between the client and the UI components on the JavaServer Faces page.

A JavaServer Faces page is made up of a tree of UI components. Whenever an event occurs on the JavaServer Faces page, these UI components access the managed beans of the JavaServer Faces application. The managed beans contain application logic and handle the processing of user data. After the managed beans process user data, the JavaServer Faces technology uses a RenderKit to render the display to the client. Different types of clients, such as a desktop client or a Portable Digital Assistant (PDA) client, require different types of RenderKits.

Like any other technology, the JavaServer Faces technology has some key concepts. Familiarity with these concepts is essential to understanding the JavaServer Faces technology.

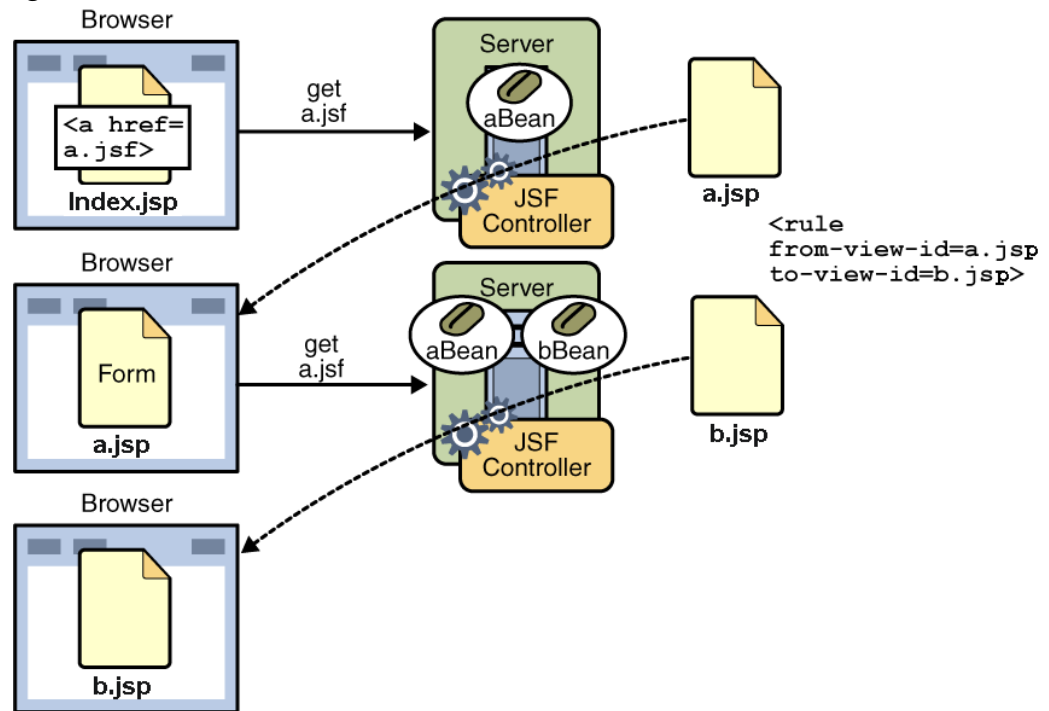
## JavaServer Faces Interaction

A JavaServer Faces application functions very much like any other servlet or JSP application. It executes in a servlet container and contains the following:

- JavaBeans components, also known as managed beans, to hold the model data and execute application-specific functionality
- Event listeners
- JSP pages with JavaServer Faces components representing the UI
- Server-side helper classes
- A custom tag library for rendering UI components
- A custom tag library for representing event handlers and validators
- UI components represented as server-side stateful objects
- Validators used to validate data on the individual components prior to server-side data updates
- Event and navigation handlers
- Application resource files specifying the JavaServer Faces Controller Servlet, managed beans, and navigation handles

A JavaServer Faces application differs from other servlet/JSP applications in that it is event driven – that is, it works by processing events (caused by user actions) triggered by the JavaServer Faces components in the pages.

Figure 12-9 shows what occurs in a JSF interaction.



**Figure 12-9** JavaServer Faces Interaction

In the diagram, the user caused an event (such as clicking a button) to occur in the `Index.jsp` page. This event notification is sent as an HTTP request (`get`) to the server. Inside the server's web container is a special servlet called the Faces servlet (shown in the diagram as the JSF Controller) which serves as the engine for all JavaServer Faces applications. Each JavaServer Faces application in the same web container employs its own controller servlet. All JavaServer Faces requests must be directed to the JavaServer Faces Controller if they are to be processed. The JavaServer Faces Controller servlet processes the request and forwards it to a `jsp` page. If necessary, using the JavaServer Faces tag libraries and the RenderKit, the `jsp` page generates the output and a response is sent back to the client which displays the result (Form).

JavaServer Faces technology provides the application with a default action listener for page navigation. Through the page navigation rule, `<rule from-view-id=a.jsp to-view-id=b.jsp>` (defined in the application's configuration file, `faces-config.xml`), the action listener for page navigation knows which page to present next.

## Comparison of JavaServer Faces Technology With Struts Framework

Listed are some of the benefits of using JavaServer Faces technology rather than the Struts framework:

- Not limited to a particular display technology – JavaServer Faces technology can be used with any display technology, which makes it suitable for a large variety of clients. In other words, JavaServer Faces components are “client device” independent and can be rendered using different markup languages through different renderers. Struts is limited to HTML.
- Easy access to beans – In JavaServer Faces technology you can assign names to managed beans, which lets you refer to them by name in the forms. Struts has a complex mechanism for accessing beans.
- Simpler controller and bean definitions – Unlike Struts, JavaServer Faces technology does not require your controller and bean classes to extend any particular parent class (for example, *Action*) or use any particular method (for example, *execute*).
- Simpler configuration file and overall structure – Comparing the configuration files of JavaServer Faces technology and Struts, the *faces-config.xml* file is much easier to use than the *struts-config.xml* file.
- Powerful tools and broad vendor support – The orientation around graphical user interface (GUI) controls and their handlers opens the possibility of simple-to-use, drag-and-drop IDEs.

Listed are the few shortcomings that JavaServer Faces technology has when compared with the Struts framework.

- Confusion in the file names – The actual pages used in the JavaServer Faces application end with the *.jsp* extension, but the URLs that are used by the deployed application end with *.faces* or *.jsf*.
- No provision for strong page layout facilities – Struts provides powerful page layout facilities, such as tiles, which JavaServer Faces does not provide.
- Much weaker automatic validation – JavaServer Faces technology comes with standard validators for missing values, length of input, and numbers in a given range. Struts, however, comes with more complex validators, such as validators for email addresses, credit card numbers, and regular expressions.

- Lack of client-side validation – Because JavaServer Faces technology is a server-side UI component framework, it does not support client-side validation. Struts, on the other hand, provides support for client-side, form-field validation based on the JavaScript language.
- Bookmarking problem – JavaServer Faces technology is POST only and does not support GET, which means you cannot bookmark the results page.

## Key JavaServer Faces Concepts

Familiarity with JavaServer Faces technology key concepts is essential for understanding the JavaServer Faces technology. This topic describes each of these concepts:

- UI component classes – Specify the state and behavior of UI components.  
A JavaServer Faces page is made up of a tree of UI components. Whenever the user submits a page, the request is sent to the server, where the parameters of the request are mapped to the appropriate component which in turn communicates with the backing beans. The backing beans are the interface to the application logic, also known as the Model logic.
- Component rendering model – Each component has a server-side representation that renders code which streams the markup of that component back to the client. The client uses that markup to draw the component on the screen. The renderer for each component can be specified either in the web application's configuration file or by runtime logic; therefore, each component can be rendered in a way that is platform independent. This means that each component has the capability to be rendered to a completely different device, such as a PDA or a cell phone, even as voice feedback. Likewise, a single component may be rendered using different presentation – for example, a boolean component could be rendered as a radio button, a check box, or a menu choice.
- Managed beans – Also known as “Backing Beans,” managed beans are JavaBean objects which serve as the interface between the presentation processing provided by the JavaServer Faces APIs and the application model logic that the web application developer provides.

- Event and listener model – The server-side processing of each component on a page can cause new behavior to be inserted into the list of events that are triggered at various phases of processing the submitted request.
- Conversion model – Each parameter arrives from the client as a string in the HTTP request. The process for handling each request includes a phase when data is converted into the appropriate Java object type.
- Validation model – Another phase in the process for handling each request includes a phase for validating that the incoming objects meet the conditions expected by the web application.
- Navigation model – After the incoming request has been processed, JavaServer Faces includes a mechanism for determining which page in the application will be displayed next.

## UI Component Model

UI components are the objects that manage interaction with a user and compose the user interface of a JavaServer Faces application. Examples include text box, panel, data grid, label, and button.

A JavaServer Faces UI component is a stateful object and is stored on the server. UI components can be either responsible for their own display or can delegate display to a renderer.

The JavaServer Faces technology UI component architecture includes the seven characteristics previously discussed.

## UI Component Classes

The JavaServer Faces technology provides a set of prebuilt UI component classes. UI component classes specify all the UI component functionality, such as maintaining the state of the component, interacting with managed beans, handling events, and rendering the markup language suitable for the client.



---

**Note** – Using advanced Java programming skills, custom UI components can be created by extending various component classes. For more information, see the course: Developing JavaServer™ Faces Components With AJAX (DTJ-3108).

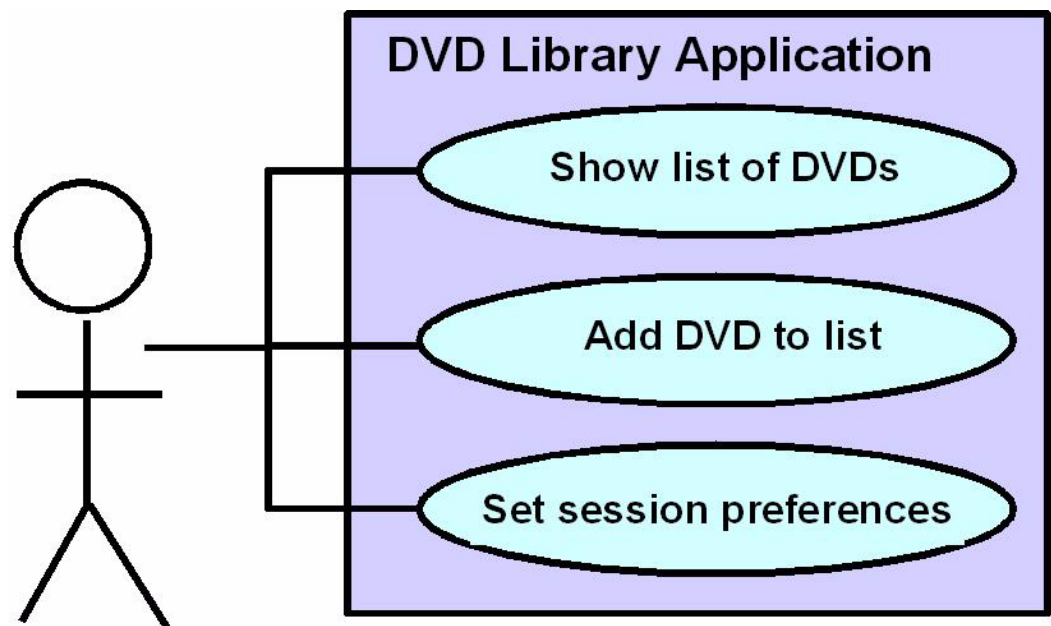
---

## DVD Library Sample Project: Use Cases

The application you build for this course is a JavaServer Faces-based re-implementation of the DVD Library web application used in other Sun Web-Tier technologies courses. Specifically, it was first introduced in:

Web Component Development Using Java Technology (SL-314).

The DVD Library presents three high-level use cases as shown in .



**Figure 12-10** DVD Library Sample Project: Use Cases

The client requests data from the following backing beans:

- `DVDLibrarySessionBean` – This bean manages login, and all other session-related information. It is bound to the `USERS` database table in the DVD Library schema.
- `DVDLibraryApplicationBean` – This bean manages the insertion of new DVDs to a user's collection and controls the assignment of next item number when a DVD is added. It is bound to the `ITEMS` and `OBJECTIDS` database table in the DVD Library schema.



## Appendix B

---

# Quick Reference for HTML

---

## Objectives

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

- Define HTML and markup languages
- Create a simple HTML document that illustrates the 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
- Explain how HTML supports text structure and highlighting
- Identify the four main elements used in creating HTML forms
- Create a basic HTML table
- Describe the main purpose for JavaScript technology, Cascading Style Sheets (CSS), and frames

## Additional Resources



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

- Graham, Ian S. *HTML 4.0 Sourcebook*. New York: John Wiley & Sons, Inc., 1998.
- JavaScript Tutorials. [Online]. Available:  
<http://www.wsabstract.com/> and  
<http://www.javascript.com/>
- Cascading Style Sheets. [Online]. Available:  
<http://www.w3.org/Style/CSS/>
- HTML 4.01 Specification. [Online]. Available:  
<http://www.w3.org/TR/html4/>
- Dave Raggett's Introduction to HTML. [Online]. Available:  
<http://www.w3.org/MarkUp/Guide/>

# HTML and Markup Languages

This section provides a brief overview of Hypertext Markup Language (HTML) and the markup languages.

## Definition

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 define the appearance of the text.

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

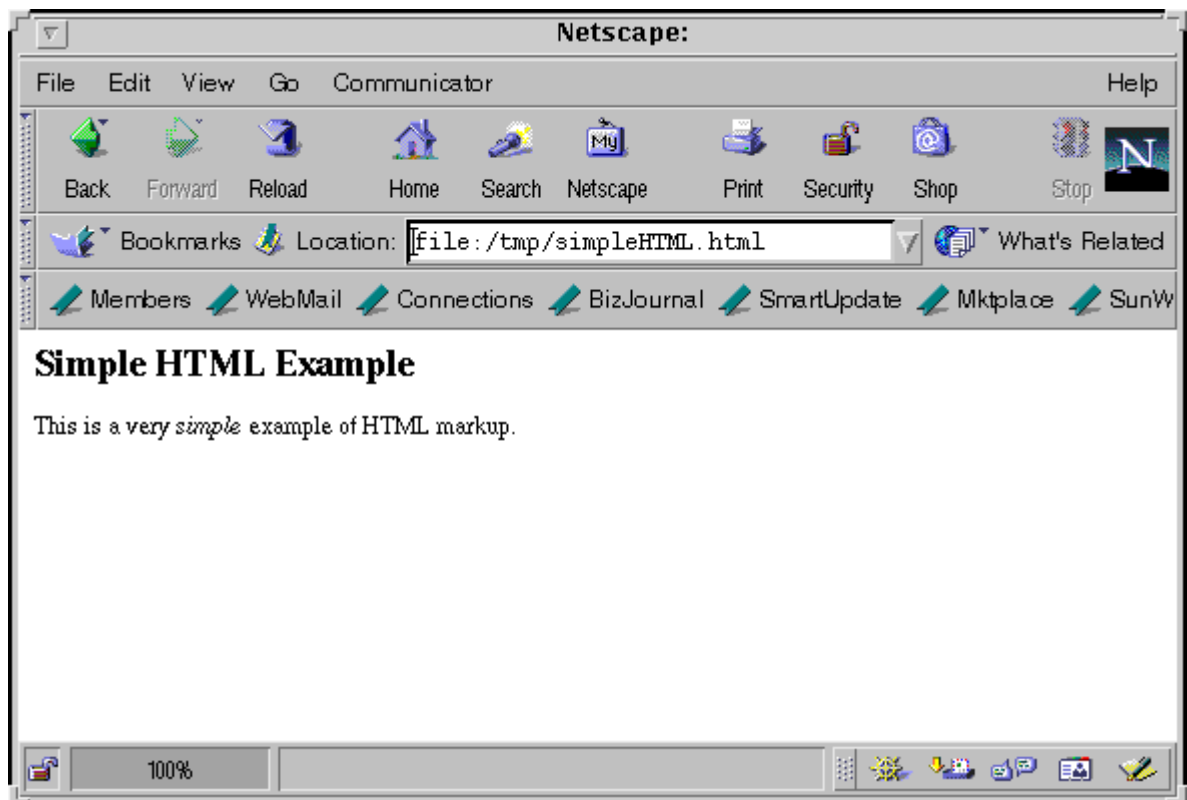
In HTML, the markup is specified within a pair of angle brackets (< >). These markings are referred to as *tags*. Figure B-1 shows a simple HTML file.

```
<HTML>
<BODY BGCOLOR="#ffffff">
<H1>Simple HTML Example</H1>
This is a very <I>simple</I> example of HTML markup.
</BODY>
</HTML>
```

**Figure B-1** Simple HTML File

Most HTML documents contain the tag <HTML> to indicate the type of document you are viewing. The <BODY> tag is used around the entire document text that appears in the browser window.

Figure B-2 shows the HTML file in a browser.



**Figure B-2** Simple HTML File Browser Display

# Creating an HTML Document

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 B-1 on page B-4 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 B-1 on page B-4, the element BODY has an attribute called BGCOLOR. This attribute 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 use capital letters.

---

## Comments

You can include comments in your HTML file using the following syntax:

```
<!-- put your comment here -->
```

The string `<!--` begins a comment and the string `-->` ends a comment.

## Spaces, Tabs, and New Lines Within Text

Browsers that display HTML documents ignore extra spaces, tabs, blank lines, and new lines 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).

## 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 following characters:

- Open bracket (<)
- Close bracket (>)
- Ampersand (&)
- Double quotes (“

This is because these characters have special meaning in HTML syntax.

Table B-1 shows the decimal code and entity references for some typical characters.

**Table B-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;
```

Use the syntax `&#60;` to refer to the open bracket character (<) in HTML using the decimal code. That is, you place the decimal code numbers between the ampersand number sign combination (&#) and a semicolon (;).

## Creating Links and Media Tags

The HTML specification provides elements that enable you to create hyperlinks to other documents. These elements also enable you to create hyperlinks to another location in the same document, to create inline images, and to 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 in this section for completeness.

---

### The `A` Element and the `HREF` Attribute

You use the `A` element (referred to as the *anchor* element) and its `HREF` attribute to create a hypertext link. The syntax is as follows:

```
<A HREF="reference_to_someplace">the displayed link</A>
```

The *reference\_to\_someplace* can be any of the following:

- URL – For example: `http://w3.org/xxx.html`
- Path name relative to the location of the current file – For example: `example.html` or `../samples/sample1.html`
- Fragment identifier – For example: `#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_someplace">` 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. You can use other attributes of the `IMG` element 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 (by default, pixels).
- `WIDTH` – Specifies the width of the image in integer units (by default, pixels).

## The APPLET Element

You use the `APPLET` element in an HTML document to indicate that a Java technology applet is to be downloaded and run. The usual syntax is as follows:

```
<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, you use the `CODEBASE` attribute to specify the URL or directory containing the class file.

## The OBJECT Element

The OBJECT element is new as of HTML 4.0. You use it wherever you would use an SRC or APPLET element. It is also 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 might be a conflict with the use of CLASSID. You can use the following code:

```
<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 – The base URL for CLASSID, DATA, and ARCHIVE
- CODETYPE – The content type for code
- DATA – A URL to refer to the object's data
- TYPE – The content type for data (such as application/java)
- HEIGHT – The height override
- WIDTH – The width override

# Text Structure and Highlighting

This section describes the HTML tags for structuring and highlighting text.

## Text Structure Tags

HTML provides the following elements to control the structure of your document:

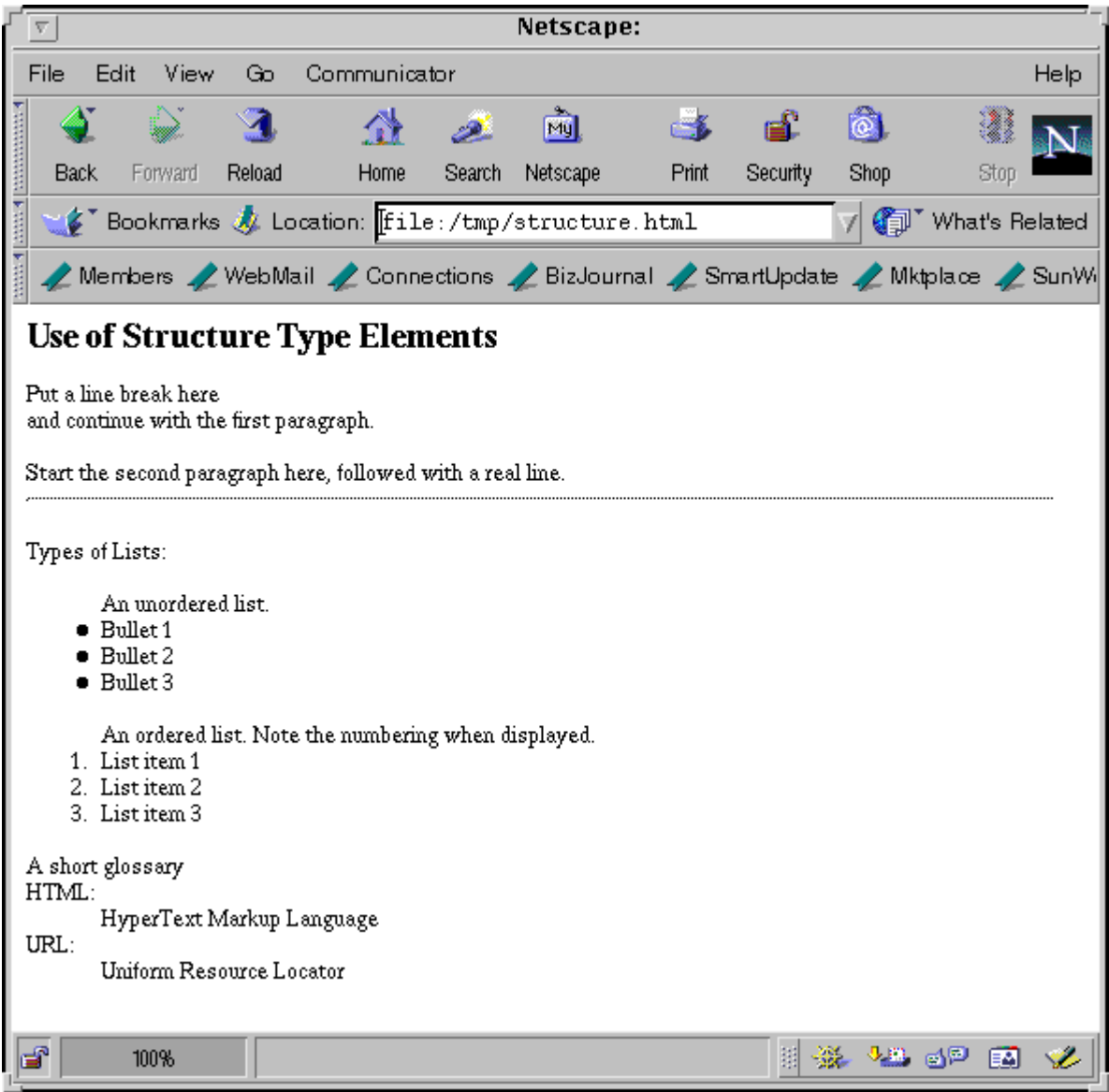
- P – Indicates the beginning of a paragraph and creates logical blocks of text
- BR – Indicates a line break and forces a new line in the display by browsers
- HR – Indicates a horizontal rule, which displays as a horizontal line across the screen
- DIV – Indicates a block of text in a document that 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 B-3 shows the use of these elements in an HTML document. This HTML document is shown in a browser in Figure B-4 on page B-13.

```
<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>
</HTML>
```

**Figure B-3** HTML Document Structure Type Tags

Figure B-4 shows the browser display of the HTML document code shown previously in Figure B-3 on page B-12.



**Figure B-4**    Structure Type HTML Elements Display

## Text Highlighting

HTML provides elements that you can use 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 emphasized text, 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 – Used to display text as boldface
- I – Used to display text as italics
- TT – Used to display text as fixed-width typewriter font
- U – Used to display text as underlined



---

**Note** – Do not use underlined text because users can confuse the underlined text with a hypertext link.

---

# HTML Forms

This section describes the basic syntactic structure of HTML forms.

## The FORM Tag

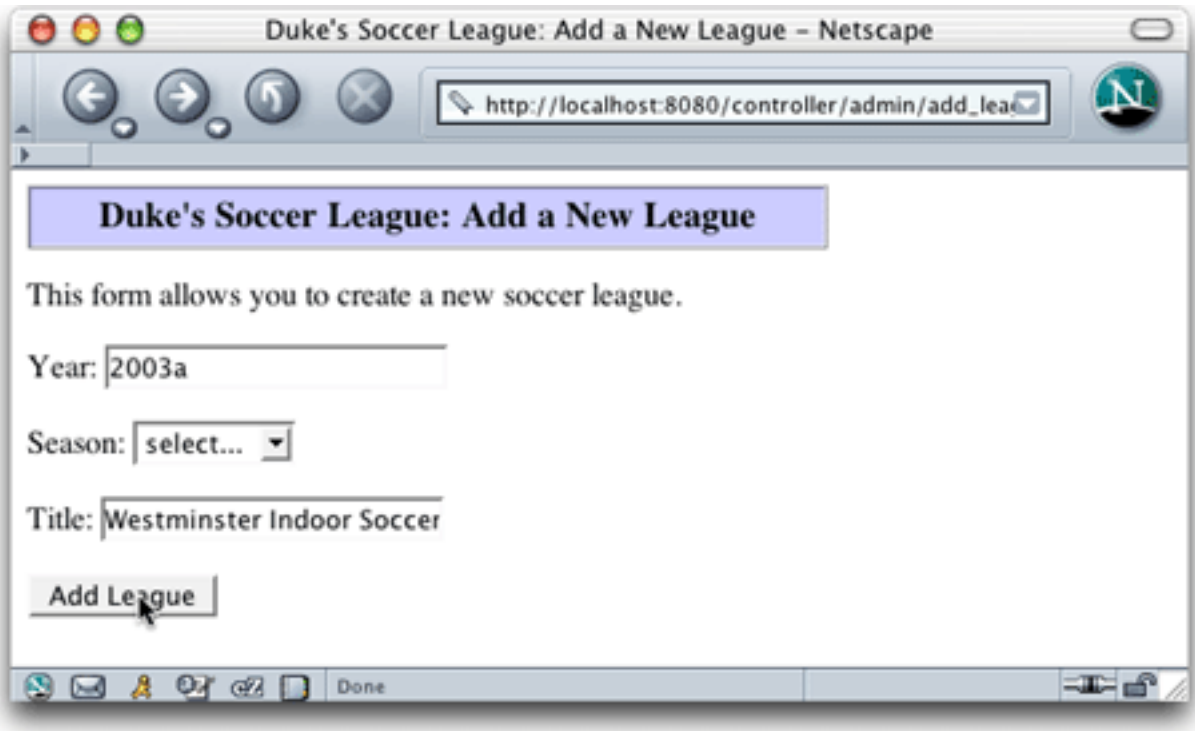
In an HTML page, the FORM tag acts as a container for a specific set of GUI components. The GUI components are specified with input tags. There are several varieties of input tags, which are shown in the next few sections. Code B-1 shows an example HTML form.

### Code B-1 Simple FORM Tag Example

```
20 <form action='add_league.do' method='POST'>
21 Year: <input type='text' name='year' /> <br/><br/>
22 Season: <select name='season'>
23         <option value='UNKNOWN'>select...</option>
24         <option value='Spring'>Spring</option>
25         <option value='Summer'>Summer</option>
26         <option value='Fall'>Fall</option>
27         <option value='Winter'>Winter</option>
28     </select> <br/><br/>
29 Title: <input type='text' name='title' /> <br/><br/>
30 <input type='submit' value='Add League' />
31 </form>
```

## HTML Forms

This HTML form creates a GUI for entering a new soccer league. Figure B-5 shows this form.



The screenshot shows a Netscape browser window titled "Duke's Soccer League: Add a New League - Netscape". The address bar displays "http://localhost:8080/controller/admin/add\_lea...". The main content area has a purple header bar with the text "Duke's Soccer League: Add a New League". Below this, a message states "This form allows you to create a new soccer league." The form contains three input fields: "Year:" with the value "2003a", "Season:" with a dropdown menu showing "select...", and "Title:" with the value "Westminster Indoor Soccer". At the bottom of the form is a button labeled "Add League". The browser's status bar at the bottom shows "Done" and various navigation icons.

**Figure B-5** Soccer League Add a New League Form



**Note** – An HTML page might 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, as shown in Table B-2.

**Table B-2** HTML Form Components

Form Element	Tag	User Function
Text field	<code>&lt;INPUT TYPE='text' ...&gt;</code>	Enter a single line of text
Submit button	<code>&lt;INPUT TYPE='submit'&gt;</code>	Submit the form
Reset button	<code>&lt;INPUT TYPE='reset'&gt;</code>	Reset the fields in the form
Checkbox	<code>&lt;INPUT TYPE='checkbox' ...&gt;</code>	Select one or more options
Radio button	<code>&lt;INPUT TYPE='radio' ...&gt;</code>	Select only one option
Password	<code>&lt;INPUT TYPE='password' ...&gt;</code>	Enter a single line of text, but the text entered cannot be seen
Select drop-down list	<pre> &lt;SELECT ...&gt;   &lt;OPTION ...&gt;     ... &lt;/SELECT&gt; </pre>	Select one or more options from a list box
Text area	<code>&lt;TEXTAREA ...&gt; ... &lt;/TEXTAREA&gt;</code>	Enter a paragraph of text

The final HTML form component is called Hidden. This is a static data field that does not show up in the HTML form in the browser window. However, the data is sent to the server in the request. You use the following tag for this form element:

```
<INPUT TYPE='hidden' ...>
```

## 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 optional attribute has a value that is set in the GUI component when the HTML page is initially rendered in the web browser.

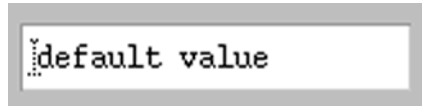
## Text Fields

An `INPUT` tag of type `text` creates a text field GUI component in the HTML form. Code B-2 shows an example text field component.

### Code B-2 Text Field HTML Tag

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

Figure B-6 shows a rendered text field.



**Figure B-6** A text Component

A text field component lets the user 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 text field accepts an optional `SIZE` attribute, which lets the HTML developer 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. Code B-3 shows an example submit button component.

### Code B-3 Submit Button HTML Tags

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

Figure B-7 shows the rendered example submit buttons.



**Figure B-7** Multiple submit Components

The submit button triggers an HTTP request for this HTML form. The implications of the attributes are as follows:

- 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 lets you represent multiple actions that the form can process. For example, if the third submit button tag in Code B-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. Code B-4 shows an example Reset button component.

**Code B-4** Reset Button HTML Tag

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

Figure B-8 shows a rendered Reset button.



**Figure B-8** A reset 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.

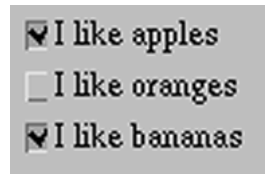
## Checkboxes

An `INPUT` tag of type `checkbox` creates a checkbox GUI component in the HTML form. Code B-5 shows an example checkbox component.

### Code B-5 Checkbox HTML Tag

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

Figure B-9 shows a rendered checkbox.



**Figure B-9** Multiple checkbox Components

The checkbox component lets the user 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 lets the user 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.

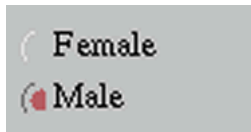
## Radio Buttons

An `INPUT` tag of type `radio` creates a radio button GUI component in the HTML form. Code B-6 shows an example radio button component.

### Code B-6 Radio Button HTML Tag

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

Figure B-10 shows a rendered radio button.



**Figure B-10** Multiple radio Components

The radio button component lets the user 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. The name of the radio button groups determine the mutually exclusive set of values mentioned previously.

## Password

An `INPUT` tag of type `password` creates a password GUI component in the HTML form. Code B-7 shows an example password component.

**Code B-7** The password HTML Tag

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

Figure B-11 shows a rendered password component.



**Figure B-11** A password Component

The password component is similar to a text field, but the characters typed into the field are obscured. However, the value of the field is sent in the form data *in the open*. This 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. Code B-8 shows an example hidden component.

**Code B-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 B-8 includes `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.

Code B-9 shows an example single selection component.

### Code B-9 Single Selection HTML Component

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

Figure B-12 shows a rendered single selection component.



**Figure B-12** 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.

Code B-10 shows an example multiple selection component.

### Code B-10 Multiple Selection HTML Component

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

Figure B-13 shows a rendered multiple selection component.



**Figure B-13** Multiple Selection Component

## The TEXTAREA Tag

A TEXTAREA tag creates a text area GUI component in the HTML form. Code B-11 shows an example text area component.

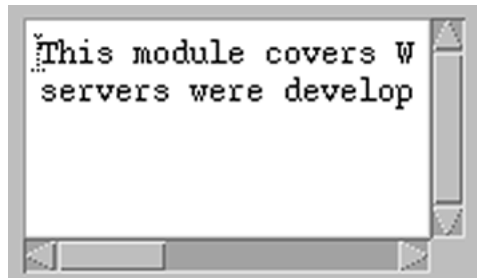
### Code B-11 The TEXTAREA HTML Tag

```
<TEXTAREA NAME='comment' ROWS='5' COLUMNS='70'>
```

This module covers Web application basics: how browsers and servers were developed and what they do. It also...

```
</TEXTAREA>
```

Figure B-14 shows a rendered text area.



**Figure B-14** A TEXTAREA Component

The TEXTAREA component lets the user enter an arbitrary amount of text. The user can type multiline input in the TEXTAREA component.

# Creating HTML Tables

HMTL tables let you organize information on a web page. Most web pages contain at least one table. Table B-3 shows the basic HTML elements used in table creation and their attributes.

**Table B-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 B-15 shows an HTML document that includes a table.

```
<HTML>
<BODY BGCOLOR="#ffffff">
<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>
  </TR>
</TABLE>
</BODY>
</HTML>
```

**Figure B-15** HTML Document Table Specification



**Note** – If you did not use the BORDER="1" attribute, the table would not have lines separating the rows and columns.

Figure B-16 shows the previous HTML document, shown in Figure B-15 on page B-28, in a browser.

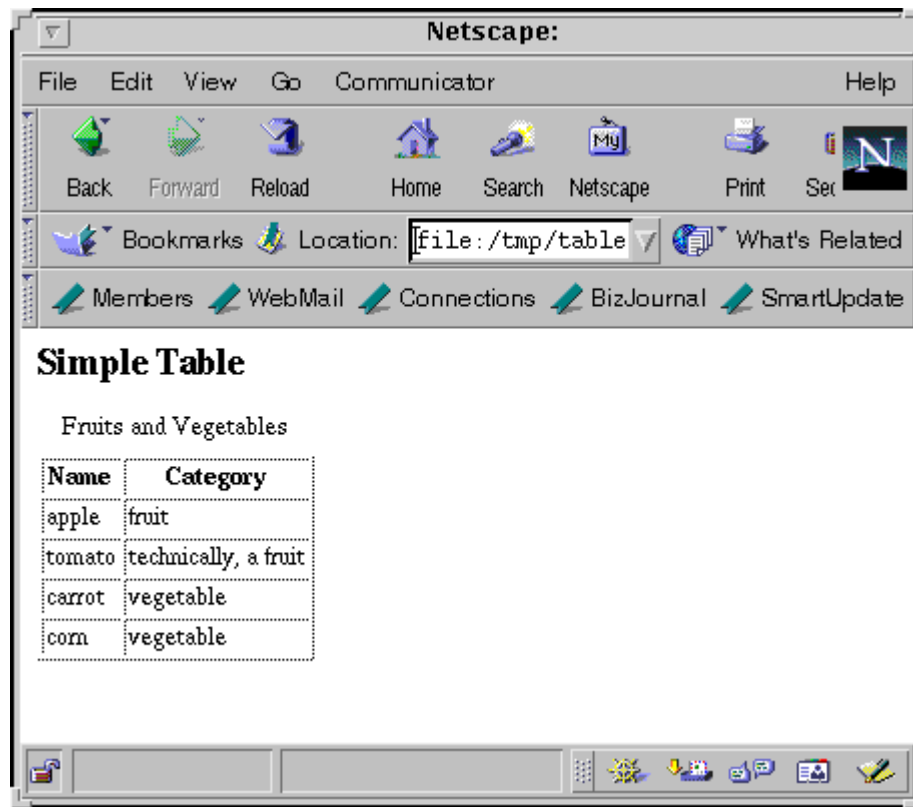


Figure B-16 HTML Table Browser Display

## Advanced HTML

This section describes the basic function and syntactic structure of the JavaScript language.

### JavaScript™ Programming Language

JavaScript technology is a scripting language (does not need to be compiled) that you can use to make your HTML more dynamic.

You use the element `<SCRIPT LANGUAGE="JavaScript">` to indicate the beginning of your JavaScript code, ending your code with the `</SCRIPT>` end tag. The example in Code B-12 calls a function `setUp()` to set the document background color to white and display the date of last modification.

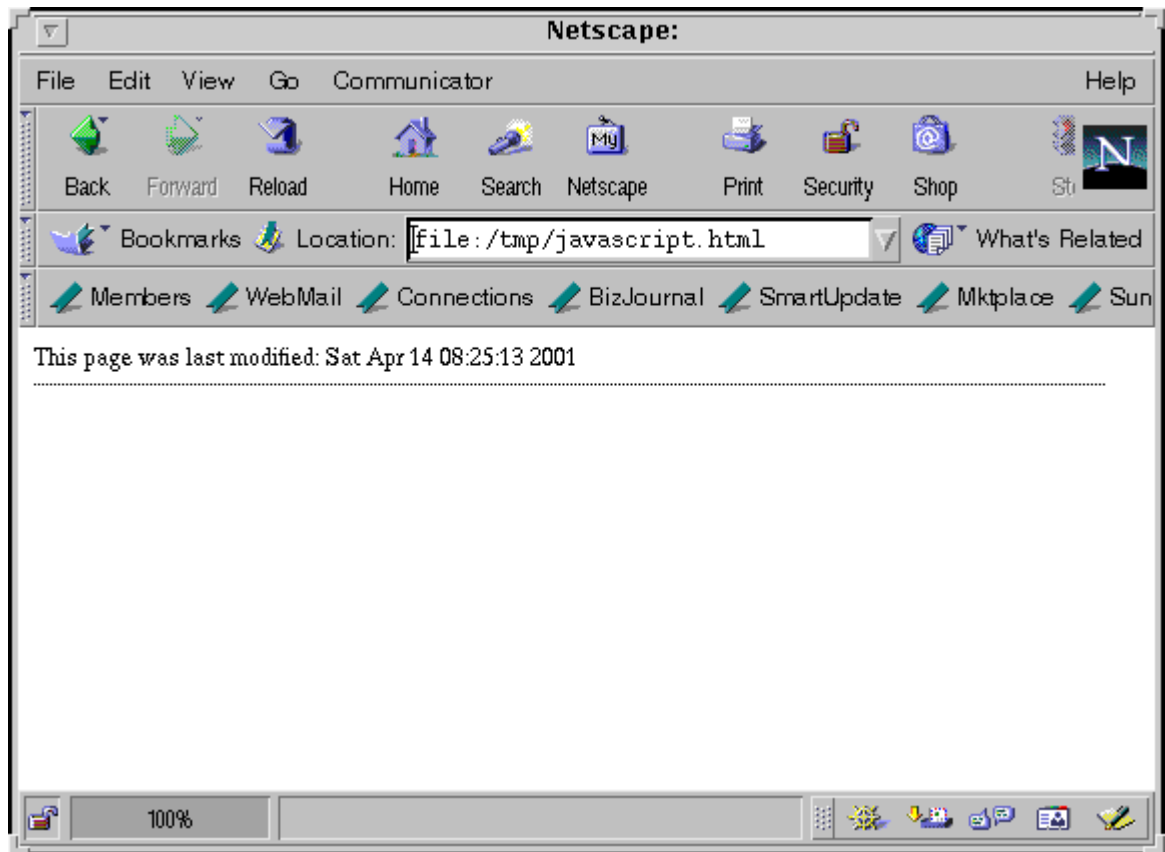
#### Code B-12 Simple JavaScript Technology Example

```
<HTML>
<SCRIPT LANGUAGE="JavaScript">
<!-- protects from older browsers
function setUp(){
    var date
    document.bgColor="#ffffff"
    date=document.lastModified
    document.write("This page was last modified: "+date)
}
setUp()
//-->
</SCRIPT>
<HR>
</HTML>
```



**Note** – Wrap your code with `<!--` and `//-->` so that older browsers that do not understand the JavaScript programming language will ignore the code.

Figure B-17 shows the generated HTML page from Code B-12 on page B-30.



**Figure B-17** HTML Document Containing JavaScript Code Display

For tutorials and free JavaScript technology, see  
<http://www.wsabstract.com/> or <http://www.javascript.com>.



**Note** – The JavaScript programming language has no relationship with the Java programming language.

## CSS

You use Cascading Style Sheets (CSS) to add style to web documents, such as fonts, spacing, colors, and so on. For the latest information about 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 following syntax:

```
property : value
```

An 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>. You can define multiple style definitions for a single selector. For example, 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 `-->`. Code B-13 on page B-34 shows an example.



---

**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 B-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>
</HTML>
```



**Note** – The set of curly braces must be used in the line beginning with `<!--` or Netscape 4.7 ignores the first style sheet rule.

Figure B-18 shows the HTML page generated from Code B-13 on page B-34.

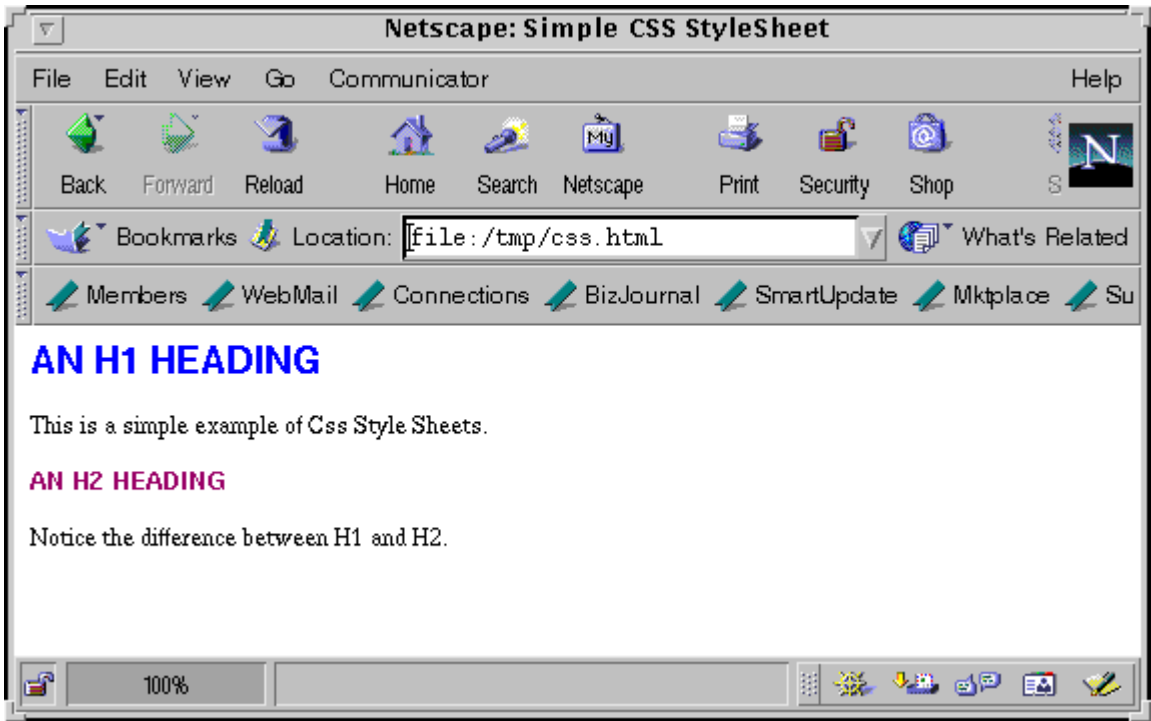


Figure B-18 CSS Browser Display

## Frames

Frames are used to provide several panes within the same browser window for document viewing. Table B-4 shows the basic elements and attributes for using frames in HTML documents.

**Table B-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 the information using:

- Actual numbers – Interpreted as pixels
- Percentages – As shown in the previous example
- 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. 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 B-19 shows an HTML document with frames.



**Figure B-19** HTML Documents Using Frames

Figure B-20 shows the initial frames in a browser.

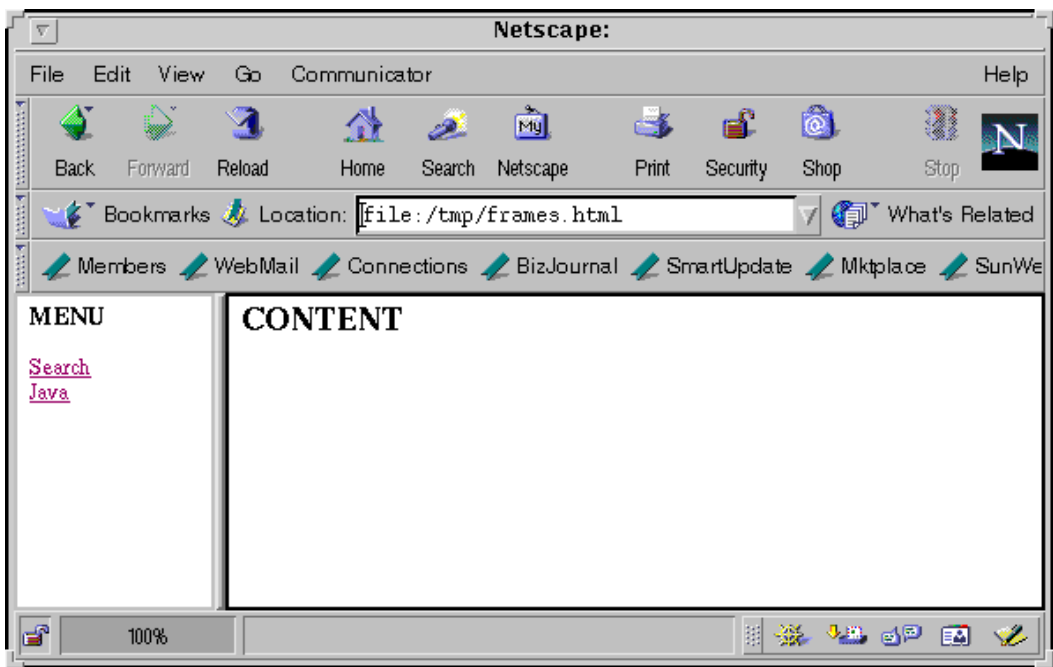


Figure B-20 Frame Browser Display

If the user clicks a link from the menu frame on the left, the right frame's content changes. Figure B-21 shows frames after clicking Java in the Menu frame.

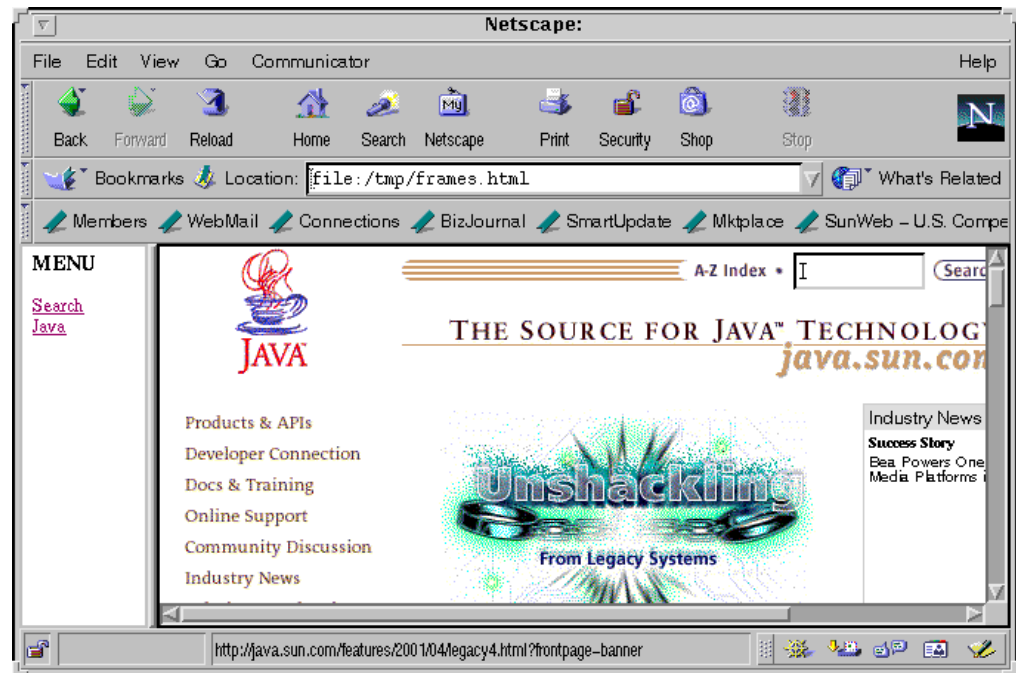


Figure B-21 Java Frames

## Appendix C

---

# 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
- Describe the structure of HTTP responses
- 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 appendix:

- IETF – Hypertext Transfer Protocol (HTTP) Working Group. [Online]. Available: <http://www.ics.uci.edu/pub/ietf/http/>
- HTTP Specifications and Drafts. [Online]. Available: <http://www.w3.org/Protocols/Specs.html#RFC>
- The Common Gateway Interface. [Online]. Available: <http://hoohoo.ncsa.uiuc.edu/cgi/>



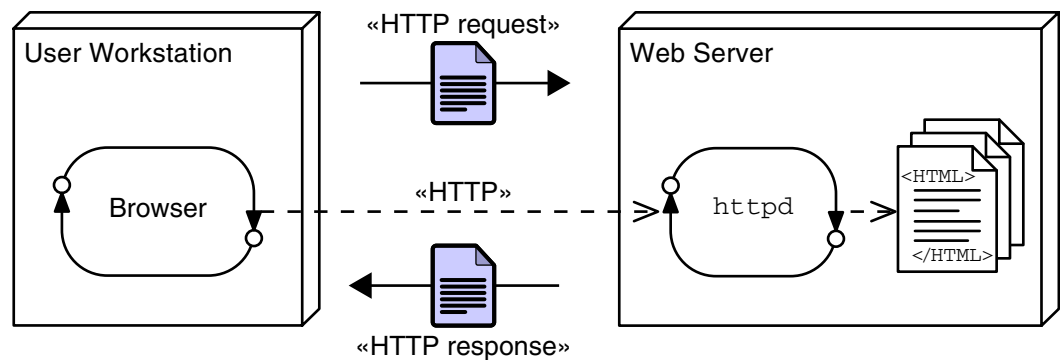
# HTTP Overview

This section defines the Hypertext Transfer Protocol (HTTP).

## Definition

The Request for Comments (RFC) 2068 defines the Hypertext Transfer Protocol as an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, object-oriented protocol. You can use HTTP 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. Figure C-1 shows this process.



**Figure C-1** Simple HTTP Request and Response

## Request Structure

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).

Code C-1 shows an example request stream.

### Code C-1 HTTP Request Header Example

```
1 GET /servlet/sl314.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/pjpeg, image/png,  
6 Accept-Encoding: gzip  
7 Accept-Language: en  
8 Accept-Charset: ISO-8859-1, *,utf-8
```

## 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.

Table C-1 shows the methods defined by HTTP 1.1.

**Table C-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 the 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 C-2 lists the request header names that are defined by HTTP 1.1.

**Table C-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 that a user agent is attempting 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 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 C-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.

## Response Structure

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.

Code C-2 shows an example.

### Code C-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>
```

## 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 C-3 lists the response-header names that are defined in HTTP 1.1.

**Table C-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 re-validation) was generated at the server.
Location	Redirects the recipient to a location other than the <i>Request-URL</i> for completion of the request or identification of a new resource.
Proxy-Authenticate	Requests the client or user agent to authenticate (which they can do with an <i>Authorization</i> request-header field). <i>Must</i> be included as part of a status code 407 (Proxy Authentication Required) response. Applies only to the current connection, unlike <i>WWW-Authenticate</i> .
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 might not be reflected by the response status code.
WWW-Authenticate	Requests the client or user agent to authenticate (which they can do with an <i>Authorization</i> request-header field). <i>Must</i> be included in 401 (Unauthorized) response messages.

## 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 C-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 has generated an error or cannot perform the request

**Table C-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 C-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 Overview

CGI, or Common Gateway Interface, 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.

## Environment Variables Set

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 C-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 C-5** CGI Environment Variables

CGI Variable	Description	HttpServletRequest 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 with which the request came in.	<code>getProtocol</code>

**Table C-5** CGI Environment Variables (Continued)

CGI Variable	Description	HttpServletRequest Method
SERVER_PORT	The port number of the host on which the server is running.	getServerPort
REQUEST_METHOD	The method with which the information request was issued.	getMethod
PATH_INFO	Extra path information passed to a CGI program.	getPathInfo
PATH_TRANSLATED	The translated version of the path given by the variable PATH_INFO.	getPathTranslated
SCRIPT_NAME	The virtual path (for example, /cgi-bin/script.pl) of the script being executed.	No method
QUERY_STRING	The query information passed to the program. It is appended to the URL with a question mark (?).	getQueryString
REMOTE_HOST	The remote host name of the user making the request.	getRemoteHost
REMOTE_ADDR	The remote IP address of the user making the request.	getRemoteAddr
AUTH_TYPE	The authentication method used to validate a user.	getAuthType
REMOTE_USER	The authenticated name of the user.	getRemoteUser
REMOTE_IDENT	The remote user name making the request. This variable is only set if the IdentityCheck flag is enabled and the client machine supports the RFC 931 identification scheme (identd identification daemon).	No method
CONTENT_TYPE	The MIME type of the query data, such as text/html.	getContentType
CONTENT_LENGTH	The length of the data (in bytes or the number of characters) passed to the CGI program through standard input.	getContentLength

**Table C-5** CGI Environment Variables (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 change all hexadecimal codes to ASCII characters



## Appendix D

---

# Quick Reference for XML

---

## Objectives

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

- Define XML
- Describe basic XML syntax
- 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: <http://www.w3.org/XML/>
- W3C XML Schema. [Online]. Available: <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.



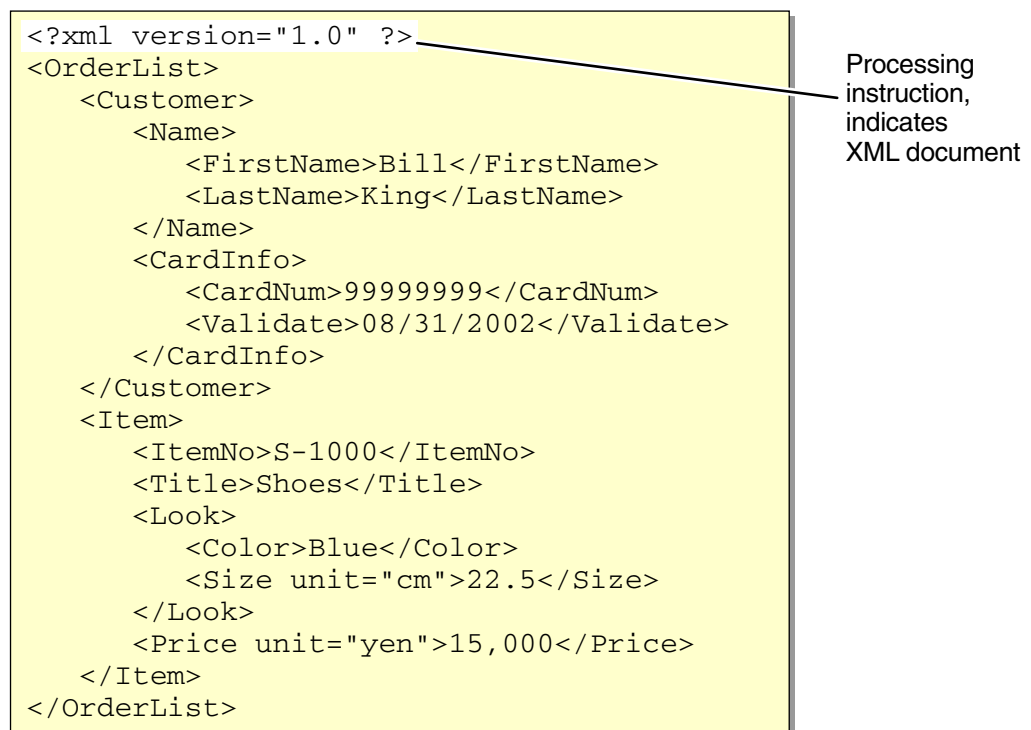
# Introduction to XML

XML 1.0 is a standard of the World Wide Web Consortium (W3C), and is available at <http://www.w3.org/TR/REC-xml>. This site 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 lets you create your own unique tags that are meaningful for your data, thus the term *extensible*.

## Simple Example

Figure D-1 shows that the syntax of XML is very much like that of HTML.



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

Processing instruction, indicates XML document

**Figure D-1** Simple XML File orderlist.xml

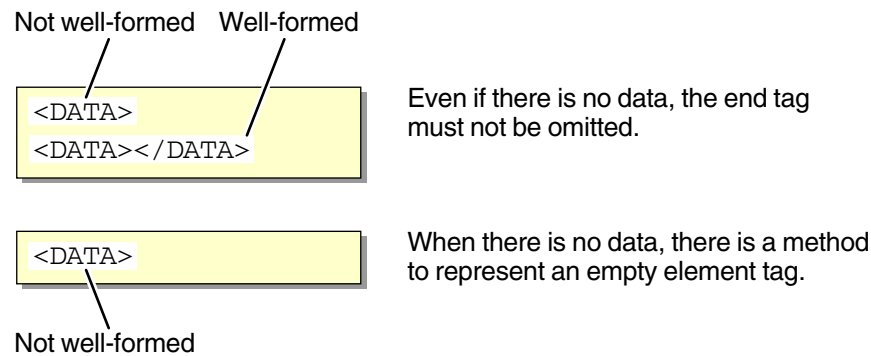
# Basic Syntax

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:

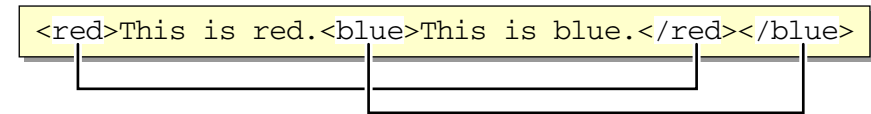
- Use a start tag and an end tag.  
Do not omit the end tag. If an element has no data, you can use empty element tags. Figure D-2 shows several examples.



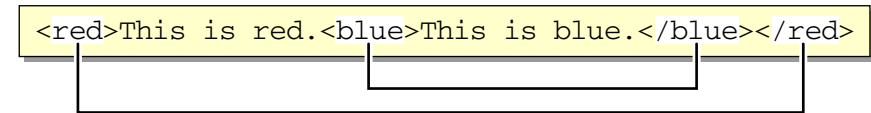
**Figure D-2** Matching Start and End Tags

- Ensure that tag nesting elements do not overlap, as shown in Figure D-3.

The blue tags described after the starting red tag must be terminated before the ending red tag.



The following tags are well-formed.



**Figure D-3** Elements Cannot Overlap



- Ensure that a root element exists.

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 D-1 on page D-3 is <OrderList>.

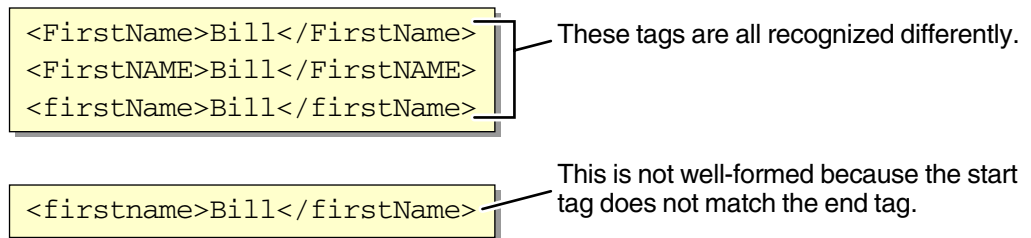
---

**Note** – Having an explicit root element makes the processing of the XML document easier when traversing the relevant tree structure.

---

- Enclose attribute values in double quotes ( " ") 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 quotes ( " ") or single quotes ( ' '). For example:  

```
<Size unit="cm">22.5</Size>
```
- Remember that characters are case-sensitive in tag and attribute names. (However, this is not true for HTML.) Ensure that uppercase and lowercase characters are distinguished from each other, as shown in Figure D-4.



**Figure D-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 D-5 shows a possible DTD for the XML document `orderlist.xml` from Figure D-1 on page D-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>
```

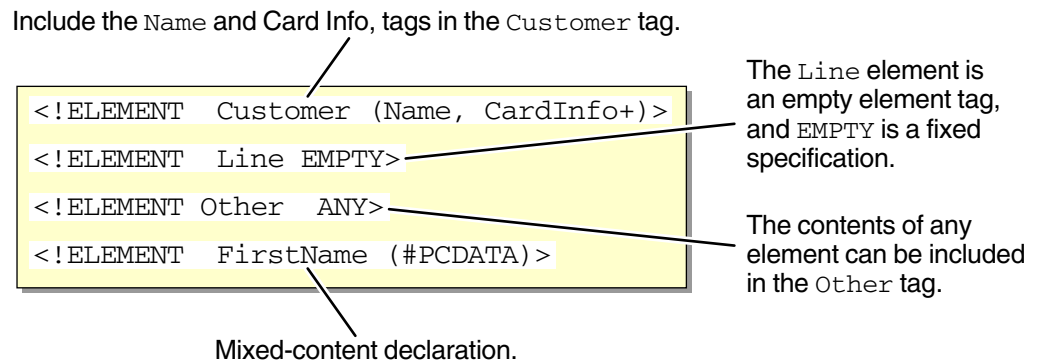
**Figure D-5** Sample DTD for `orderlist.xml`

## DTD-specific Information

The main syntax for DTDs is described as follows:

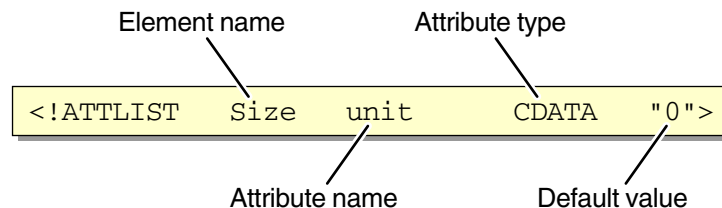
- Document type declaration `<!DOCTYPE . . . [` – Makes a declaration to describe a DTD. Within this declaration, the declaration of the element type as an XML document structure and other declarations are made.
  - Identifies a *root* element name after the string `<!DOCTYPE`
  - Describes the DTD within the document type declaration
- Element type declaration `<!ELEMENT . . . >` – Declares what elements are specified, the order the elements are specified, and the number of elements.

Figure D-6 shows element names after the string `<!ELEMENT`.



**Figure D-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, as shown in Figure D-7.



**Figure D-7** The `ATTRIBUTE` Declaration

Additional notation is used to indicate the multiplicity of sub-elements within an element declaration. Table D-1 lists this notation.

**Table D-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, Table D-2 lists two pre-defined, special element types.

**Table D-2** Pre-Defined Element Types

Symbol	Description
#PCDATA	The character data to be analyzed. Child elements and character data can be included in the elements. It is also called <i>parsed character data</i> and 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. This includes 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>
```

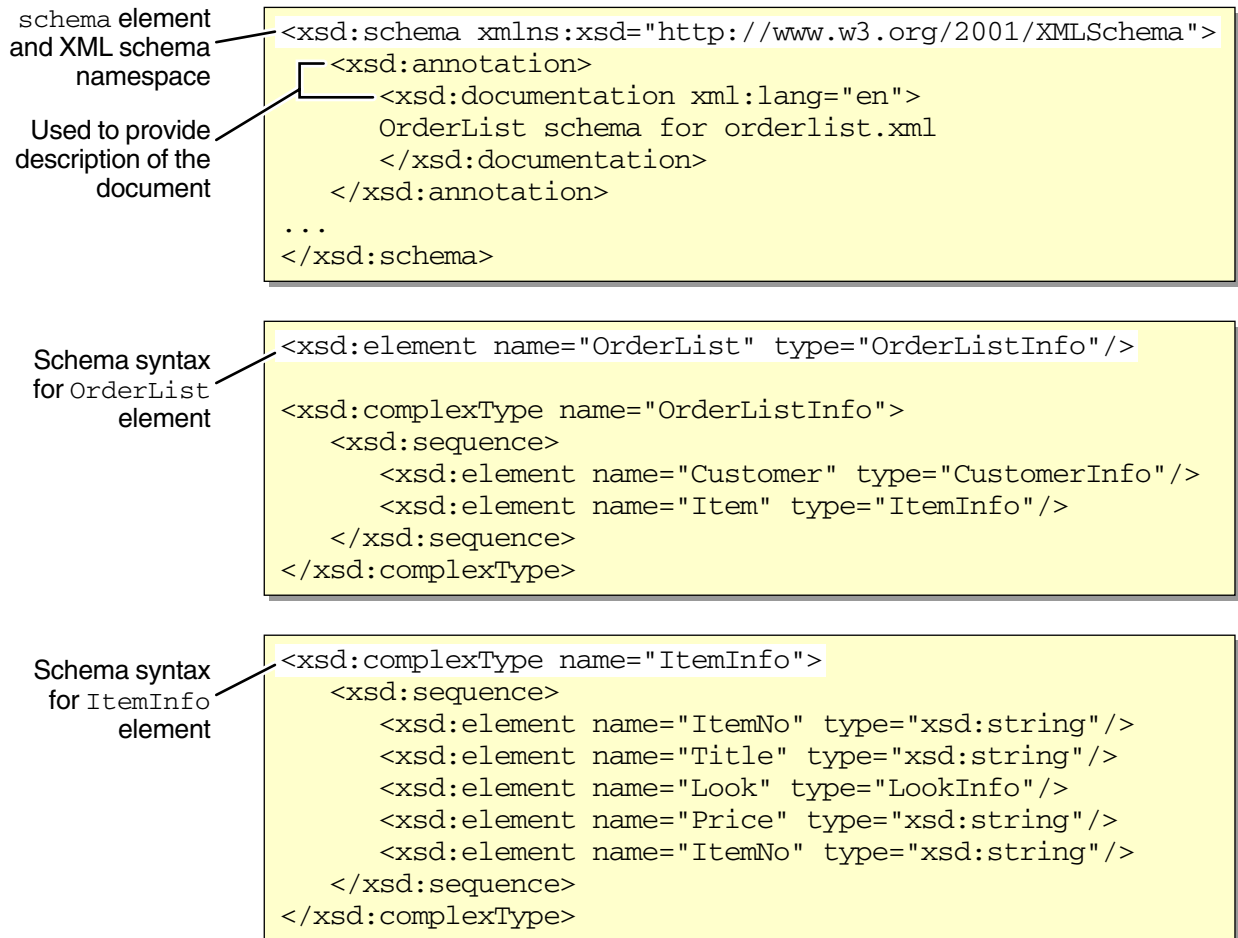
The main sub-elements in an XML schema document are:

- `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.

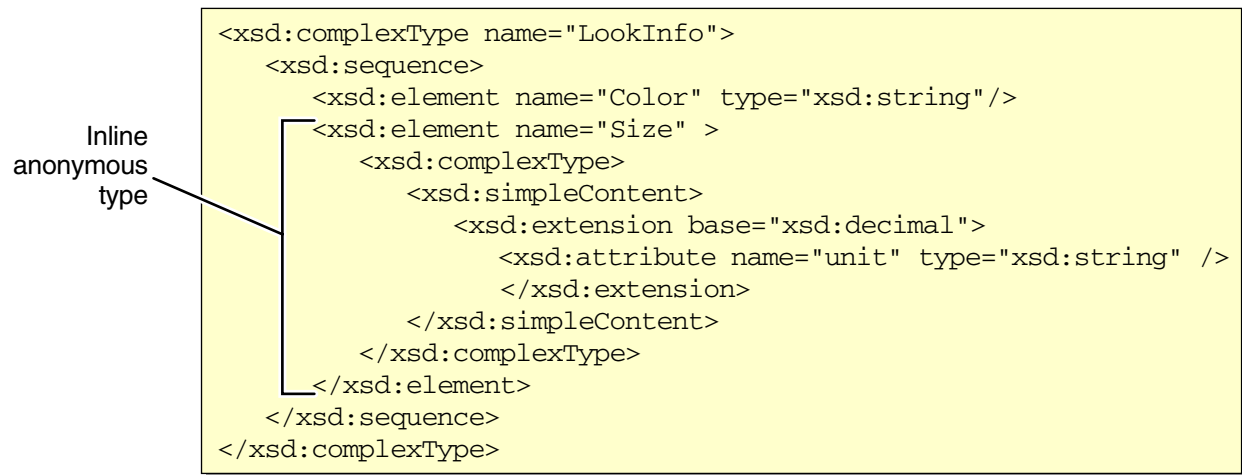
Figure D-8 shows an XML schema syntax for parts of the XML document in Figure D-1 on page D-3.



**Figure D-8** XML Schema for orderlist.xml

The OrderList element is composed of Customer and Item elements. Both are complex types because they can contain other elements. Therefore, the complex types for 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 D-9 on page D-11.





**Figure D-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 D-1 on page D-3), therefore, 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, therefore 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.

