Java Web Programming

Student Workbook

itcourseware
Training Materials For IT Professionals

JAVA WEB PROGRAMMING

Mike Naseef, Jamie Romero, and Rick Sussenbach

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112

Editors: Danielle Hopkins and Jan Waleri

Editorial Assistant: Ginny Jaranowski

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- Write web applications that combine Java Servlets, JavaServer Pages, and JavaBeans using the Model-View-Controller architecture.
- Use JavaBeans to encapsulate business and data access logic.
- Generate HTML output with JavaServer Pages.
- Process HTTP requests with Java Servlets.
- Configure your web applications with the *web.xml* deployment descriptor.

COURSE OVERVIEW

- **Audience**: Java programmers who need to develop web applications using JSPs and Servlets.

- **Prerequisites**: Java programming experience and basic HTML knowledge are required.

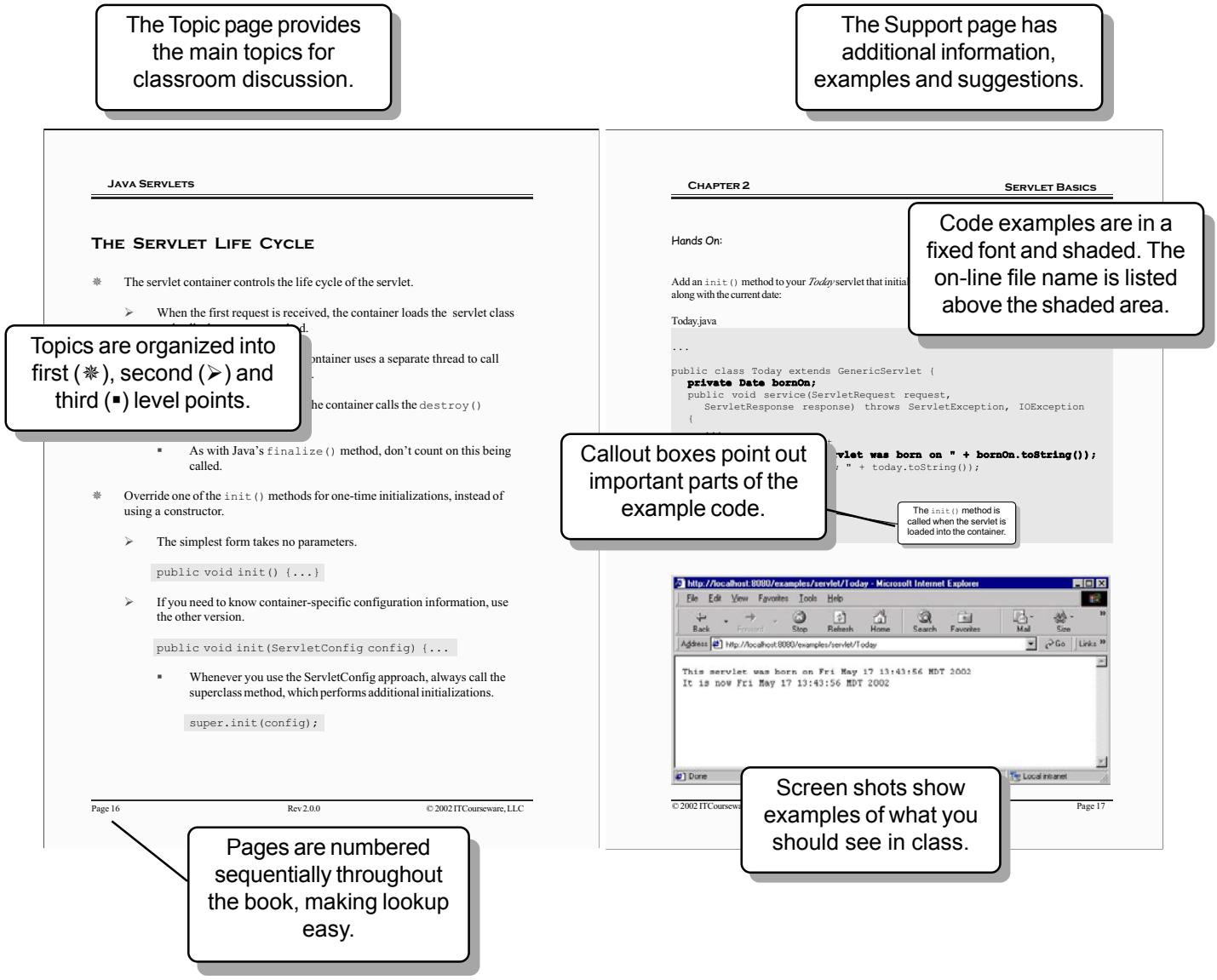
- **Classroom Environment:**

  - A workstation per student.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. Hands On sections provide opportunities for practical application of key concepts. Try It and Investigate sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.



SUGGESTED REFERENCES

Basham, Bryan, Kathy Sierra, and Bert Bates. 2004. ***Head First Servlets and JSP: Passing the Sun Certified Web Component Developer Exam (SCWCD)***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596005407.

Bergsten, Hans. 2003. ***JavaServer Pages, 3rd Edition***. O'Reilly & Associates, Sebastopol, CA. ISBN 0596005636.

Hall, Marty and Larry Brown. 2003. ***Core Servlets and JavaServer Pages, Vol. 1: Core Technologies, 2nd Edition***. Prentice Hall, Englewood Cliffs, NJ. ISBN 0130092290.

Hall, Marty, Larry Brown and Yaakov Chaikin. 2006. ***Core Servlets and JavaServer Pages, Volume II (2nd Edition)***. Prentice Hall, Englewood Cliffs, NJ. ISBN 0131482602.

Heffelfinger, David, 2010. ***Java EE 6 with GlassFish 3 Application Server***. Packt Publishing, Birmingham, UK. ISBN 1849510369

Jendrock, Eric, et.al. 2010. ***The Java EE 6 Tutorial: Basic Concepts (4th Edition).*** Prentice Hall, Upper Saddle River, NJ. ISBN 0137081855

Steelman, Andrea, Joel Murach. Bergsten, Hans. 2008. ***Murach's Java Servlets and JSP, 2nd Edition***. Mike Murach & Associates. ISBN 1890774448.

Java Servlet Technology: ***http://www.oracle.com/technetwork/java/index-jsp-135475.html***
JSP Technology: ***http://www.oracle.com/technetwork/java/jsp-138432.html***
JSTL Technology: ***http://www.oracle.com/technetwork/java/jstl-137486.html***
Java EE 6 Tutorial: ***http://download.oracle.com/javaee/6/tutorial/doc/***

CHAPTER 2 - WEB APPLICATIONS

OBJECTIVES

- Describe Java web technologies.
- Encapsulate a web application within a WAR file.
- Build and deploy a web application.

WEB APPLICATIONS

- *Web applications* are applications that the end user can access using a standard web browser.

- The Java Platform Enterprise Edition (Java EE) defines a web application as a collection of web components and supporting files.

  - Web components include Java servlets and JSP files.

  - Supporting files include static HTML documents, image files, and supporting classes.

- Your web application runs in the environment of a web container, which is managed by an application server.

  - Web containers can contain several web applications.

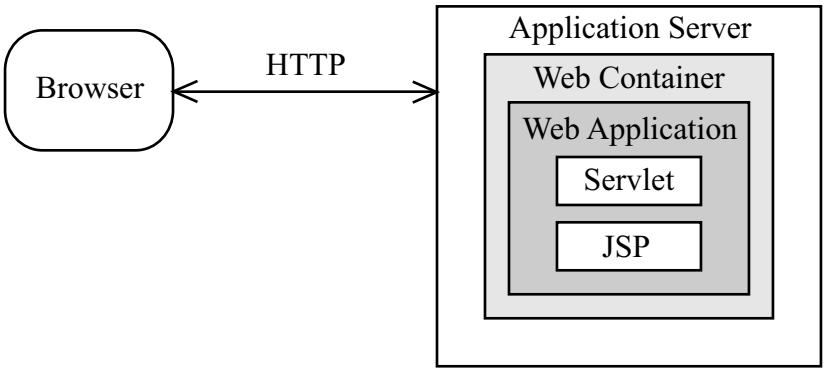
    - Your applications can work together or operate independently.

- Each web application is addressed with a context path.

  - The context path is determined when the application is deployed.

  - A web container can contain a "default" application, which has an empty context path.

  - To access a component or file in the web application from a browser, you must include the context path in the request URL.


Application Server
Browser
HTTP
Web Container
Web Application
Servlet
JSP

JSPs and Servlets

- The standard protocol for communication between browsers and servers is designed for static documents.
  - Typically, a web server returns the contents of a static file in response to a browser request.
- Use servlets and JavaServer Pages (JSP) to handle requests from a browser dynamically.
  - A *servlet* is a web component which receives an object encapsulating the browser request and constructs a response to the browser.
    - The response typically contains an HTML document.
  - JSP pages start as text documents containing HTML with special tags for executing Java code.
    - JSP pages are compiled into servlets automatically.
    - HTML designers do not need to learn Java.
    - Java developers do not need to learn HTML.
- You get some important benefits by using Java's web component architecture:
  - Your application will be portable across web containers.
  - You can make full use of the vast set of Java APIs.

HelloWorld/src/web/HelloWorldServlet.java

```
package web;

import java.io.IOException;
import java.io.PrintWriter;

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

public class HelloWorldServlet extends HttpServlet {
      @Override
      protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                  throws ServletException, IOException {
            String name = req.getParameter("nm");
            String outputText = "Hello " + name;

            PrintWriter pw = resp.getWriter();
            pw.println("<html>");
            pw.println("<head><title>Hello World</title></head>");
            pw.println("  <body>" + outputText + "</body>");
            pw.println("</html>");
            pw.close();
      }
}
```

THE WAR FILE

- You must organize your web application using a specific directory structure.
  - The application root directory acts as the document root for your application.
    - You put your JSP, HTML, and other supporting files here.
    - You can use subdirectories to organize your application.
  - Store your application files in a subdirectory named *WEB-INF*.
    - Place the optional *web.xml* configuration file here.
    - This subdirectory is not accessible via the web server.
  - Put your servlet classes and supporting classes in the *WEB-INF/classes* directory.
  - Put any JAR files specific to your application in the *WEB-INF/lib* directory.
    - This is the preferred method for storing your JavaBeans.
    - If a JAR file will be used by other applications, it may make more sense to put it in a system-wide or server-wide directory.
- You can package your application for distribution in a Web ARchive (WAR) file.
  - A *WAR* file is a JAR file that contains all of the files in your application.
  - Since WAR files must conform to the Java EE specifications, they are portable between different web containers.

WEB.XML

- Provide an optional *deployment descriptor* to supply additional configuration information for your web application.
  - Create it as *WEB-INF/web.xml* in your web application directory.
- List files the container should look for when the user request specifies a context with the **<welcome-file-list>** element.
- Use a **<servlet>** element to define each of your servlets.
  - Specify a unique **<servlet-name>** for each component.
  - For servlets, specify the full class name in the **<servlet-class>** element.
- Use the **<servlet-mapping>** element to map a servlet or JSP page to a specific URL within your web application.
  - The **<servlet-name>** is the name you specified in the **<servlet>** element.
  - The URL you specify in the **<url-pattern>** element is relative to the context path.

HelloWorld/WebContent/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://java.sun.com/xml/ns/javaee"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
      version="3.0">
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

  <servlet>
      <servlet-name>TheServletName</servlet-name>
      <servlet-class>web.HelloWorldServlet</servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>TheServletName</servlet-name>
      <url-pattern>/SayHello</url-pattern>
  </servlet-mapping>
</web-app>
```

This name is used to reference the servlet in other elements, e.g. **servlet-mapping** elements.

This pattern matches any path that ends with **/SayHello/**.

Note:

Prior to the Servlet 3.0 specification, *web.xml* was required. Servlet 3.0 defined several annotations that you can add to your code to take the place of many, but not all, of the *web.xml* entries.

BUILDING AND DEPLOYING THE WAR

- You build your application using these steps:

  1. Create a directory in which to build your web application.

  2. Compile your classes putting the resulting class files in *WEB-INF/classes*.

     - If you create any JAR files, put them in *WEB-INF/lib*.

  3. Copy your JSP files, HTML files, and other supporting files into the application directory.

  4. Optionally, create your deployment descriptor in *WEB-INF/web.xml*.

- To build the WAR file, use the **jar** command to archive the application directory.

```
jar cvf MyApplication.war webapp-directory
```

- You deploy a web application with these fundamental steps.

  1. Pass the WAR file to your web container.

     - You might simply copy the file to a specific location or use a tool to locate the file.

  2. Specify the context path for the application.

     - The context path often defaults to the name of the WAR file.

  3. Configure any container-managed resources as specified in the deployment descriptor.

     - These might include database connections, JNDI services, and security roles.

The mechanisms for performing the deployment steps is determined by your web container. Some container providers have GUI or web-based tools for deploying applications. You may need to create the appropriate configuration files manually and include them in your WAR file.

Try It:

Your instructor will show you how to build and deploy the HelloWorld web application on your system.

LABS

❶ Modify the HelloWorld web application so that the name is passed from the browser to the server as two fields: firstName and lastName.

❷ Modify the HelloWorld web application again, this time change the output so that the greeting is "Hi" rather than "Hello."

❸ Use any additional HTML or CSS knowledge you may have to dress up the HelloWorld web application.

CHAPTER 3 - JAVA SERVLETS

OBJECTIVES

- Write a Java servlet to process a request and generate a response for your web application.
- Describe the lifecycle of a servlet.

HTTP REQUESTS

- HTTP defines several request methods.
  - A **GET** request usually comes from a user typing a URL into the browser or clicking on a link.
    - **GET** requests typically retrieve information.
    - **GET**s encode HTTP request parameters within the URL.
  - A **POST** request usually comes from an HTML form, where **<form method="post" ...>**.
    - **POST**s are usually used to modify a resource.
    - **POST**s pass HTTP request parameters as part of the message body.
  - The **PUT** method requests that the body of the request be stored at the specified URI.
  - The **DELETE** method requests that the data at the specified URI is removed.
  - The **TRACE** method requests that the body of the request be returned intact; this is used for debugging.
  - The **HEAD** method is used whenever the client wants only header information, not the data in the document.
- **GET**s and **POST**s make up the majority of HTTP requests.

A sample HTTP **GET** request:

```
GET /Stocks/StockPrice?symbol=goog HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7.8)
Gecko/20050511 Firefox/1.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/
html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en,en-us;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/Servlet/StockQuery1.html
```

The **request** parameter is passed in the URL for a **GET**.

A sample HTTP **POST** request:

```
POST /Stocks/StockPrice HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7.8)
Gecko/20050511 Firefox/1.0.4
Accept: text/xml,application/xml,application/xhtml+xml,text/
html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en,en-us;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/Servlet/StockQuery1.html
Content-Type: application/x-www-form-urlencoded
Content-Length: 10
symbol=goog
```

The **request** parameter is passed as part of the message body for a **POST**.

For more information on HTTP request methods, see:
***http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html*.**

HTTPSERVLET

- A *servlet* is a class that extends **javax.servlet.http.HttpServlet**.
  - **HttpServlet** declares **doGet()** and **doPost()** methods, that are called by the web container whenever it receives a **GET** or **POST** request for the servlet.
  - Your servlet should override the **doGet()** or **doPost()** methods.
    - This is where you process the request from the browser.
    - If you don't differentiate between **GET** and **POST** requests, have your **doPost()** method call your **doGet()**, or vice versa.
  - Other **do*xxx*()** methods are available, but rarely used for web applications.
- **doGet()** and **doPost()** are passed an **HttpServletRequest** and an **HttpServletResponse** as parameters.
  - The **HttpServletRequest** gives you information about the request.
    - Call the **getParameter()** method on the **HttpServletRequest** to retrieve HTML form data.

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

  - Use the **HttpServletResponse** to write content back to the browser.
    - The **getWriter()** method returns a **PrintWriter** that you can use to write a document to the browser.

```
PrintWriter pw = response.getWriter();
pw.println("<html>");
```


Stocks/src/web/StockServlet.java

```
package web;
...
@WebServlet("/StockPrice")
public class StockServlet extends HttpServlet {
   ...
  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {

     String symbol = req.getParameter("symbol");
     double amount = stockDAO.getPrice(symbol);

     PrintWriter pw = resp.getWriter();
     pw.println("<html>");
     pw.println("<head><title>Stocks</title></head>");
     pw.print("<body>");
     if (amount != -1) {
        pw.printf("<p>%s = %.2f</p>", symbol, amount);
     }
     else {
        pw.println("<p>Invalid Symbol</p>");
        pw.println("<p><a href=\"select.html\">Try Again?</a></p>");
     }

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

SERVLET LIFECYCLE

- The web container controls the lifecycle of the servlet.
  - When the first request is received, the container instantiates the servlet and calls the **init()** method on it.
  - For every request, the container uses a separate thread to call the **doGet()** or **doPost()** methods.
  - When the servlet is unloaded, the container calls the **destroy()** method.
  - There will typically only be one instance of the servlet created per web container.
  - The time between **init()** and **destroy()** can usually be measured in days or even months.

Stocks/src/web/StockServlet.java

```
package web;
...
@WebServlet("/StockPrice")
public class StockServlet extends HttpServlet {
   private StockDAO stockDAO;

   @Override
   public void init() throws ServletException {
      System.out.println("In init() method");
      stockDAO = new StockFileDAO(getServletContext());
   }


   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {
      ...
   }

   @Override
   public void destroy() {
      System.out.println("In destroy method");
   }
}
```

@WebServlet Annotation

- Add the **@WebServlet** annotation to your servlet class to define the alias that will be used to identify your servlet within a URL.

```
@WebServlet("/MyServlet")
```

  - Use the URL pattern to access the servlet from an HTML form.

```
<form action="MyServlet" method="get">
```

  - You can specify multiple URL patterns if you want to have multiple access points.

```
@WebServlet(urlPatterns = {"/MyServlet", "/srv/*"})
```

- Versions of the Servlet API prior to 3.0 used *web.xml* to accomplish this.

```
<web-app ...>
  <servlet>
    <servlet-name>MyServletName</servlet-name>
    <servlet-class>web.TheServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyServletName</servlet-name>
    <url-pattern>/MyServlet</url-pattern>
  </servlet-mapping>
</web-app>
```


Stocks/WebContent/select.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Stocks</title>
</head>
<body>
   <form action="StockPrice" method="GET">
      <input type="text" name="symbol" />
      <input type="submit" value="Submit" />
   </form>
</body>
</html>
```

The **form action** matches the **urlPattern.**

Stocks/src/web/StockServlet.java

```
package web;
...
import javax.servlet.annotation.WebServlet;
...
@WebServlet("/StockPrice")
public class StockServlet extends HttpServlet {
   ...
  @Override
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {

     String symbol = req.getParameter("symbol");
     double amount = stockDAO.getPrice(symbol);

     PrintWriter pw = resp.getWriter();
     pw.println("<html>");
     pw.println("<head><title>Stocks</title></head>");
     ...
  }
   ...
}
```

urlPatterns begin with a leading slash.

LABS

❶ Modify the Stock web application to display the name of the company in the results instead of the stock symbol.

❷ Add additional data to the *stocks.txt* file such as previous day closing price or 52 week range. Update *Stock.java*, *StockDAO.java*, *StockFileDAO.java*, and *StockServlet.java* to present the additional information for the selected stock.

❸ Create a lottery number generator web application. Your HTML form should retrieve the number of lottery numbers you wish to generate as well as the range the numbers should be drawn from. For example, if you were targeting Colorado Lotto which draws six numbers from a range of 1-42, you would send a value of 6 and 42 to the web application.

❹ Create a calculator web application whose HTML form contains two text boxes for numeric input as well as add, subtract, multiply, and divide buttons. Build a simple Java class that contains fields that correspond to the two numbers and methods for each calculation option (add, subtract, etc.). Your servlet should read values from the form, create an instance of your Java class using those values, call the appropriate method on the instance depending on which button was clicked, and generate an HTML response to the user containing the results.

CHAPTER 4 - JAVASERVER PAGES

OBJECTIVES

- Use JavaBeans, JavaServer Pages, and Servlets to implement the Model View Controller architecture.
- Use JSP Expression Language and the JSP Standard Tag Library to generate HTML.

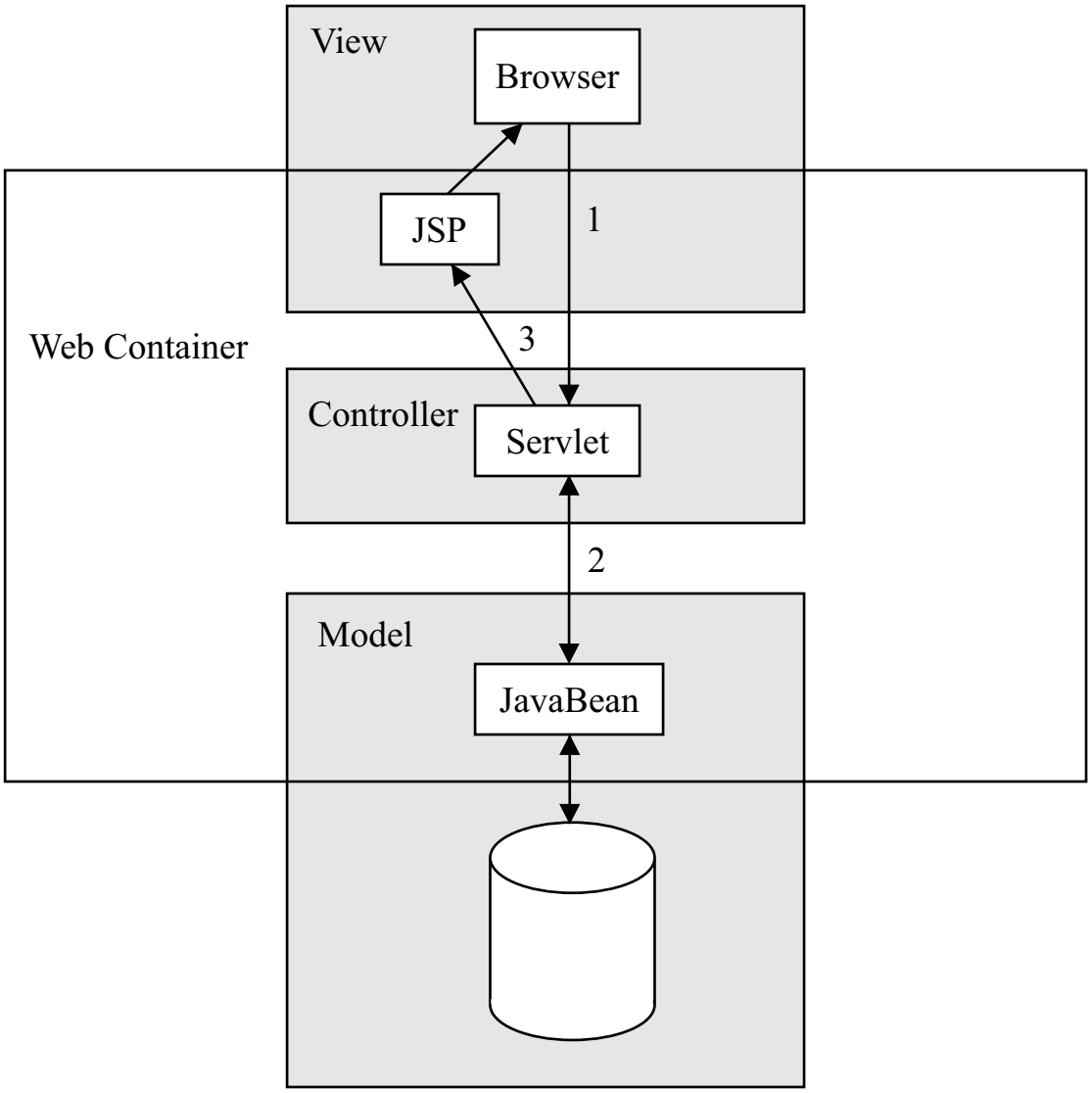
MVC and Web Applications

- The Model-View-Controller (MVC) architecture was originally created to separate user interface code (the *view*) from domain code (the *model*).
  - The *controller* was introduced as a separate body of code that manages the translation of events in the view to methods in the model.
  - The view only accesses the model to retrieve values for display.
  - The model should not have any knowledge of the view or the controller.
- For web applications, use JavaBeans to define the model.
  - A JavaBean is simply a Java class with a no-arg constructor and get/set methods.
  - The controller should encapsulate business object data in JavaBeans to make the data accessible to the JSP views.
- Use a servlet as the controller.
  - It will extract data needed to handle the request from the browser.
  - The servlet will also call methods on JavaBeans to process the request.
  - Finally, it will forward the request to the JSP page, including any beans needed to generate the view.
    - The servlet might choose between JSP pages based on the results of the request.
- Use JSP pages to generate the view.
  - The view will retrieve information to display from the beans included by the servlet.

The benefits of MVC are similar to encapsulation. Changes in the model can be made without impacting the view. The view can be modified, or new views can be implemented without impacting the model. Developers can focus on their skills — database programmers do not need to understand user interface issues.

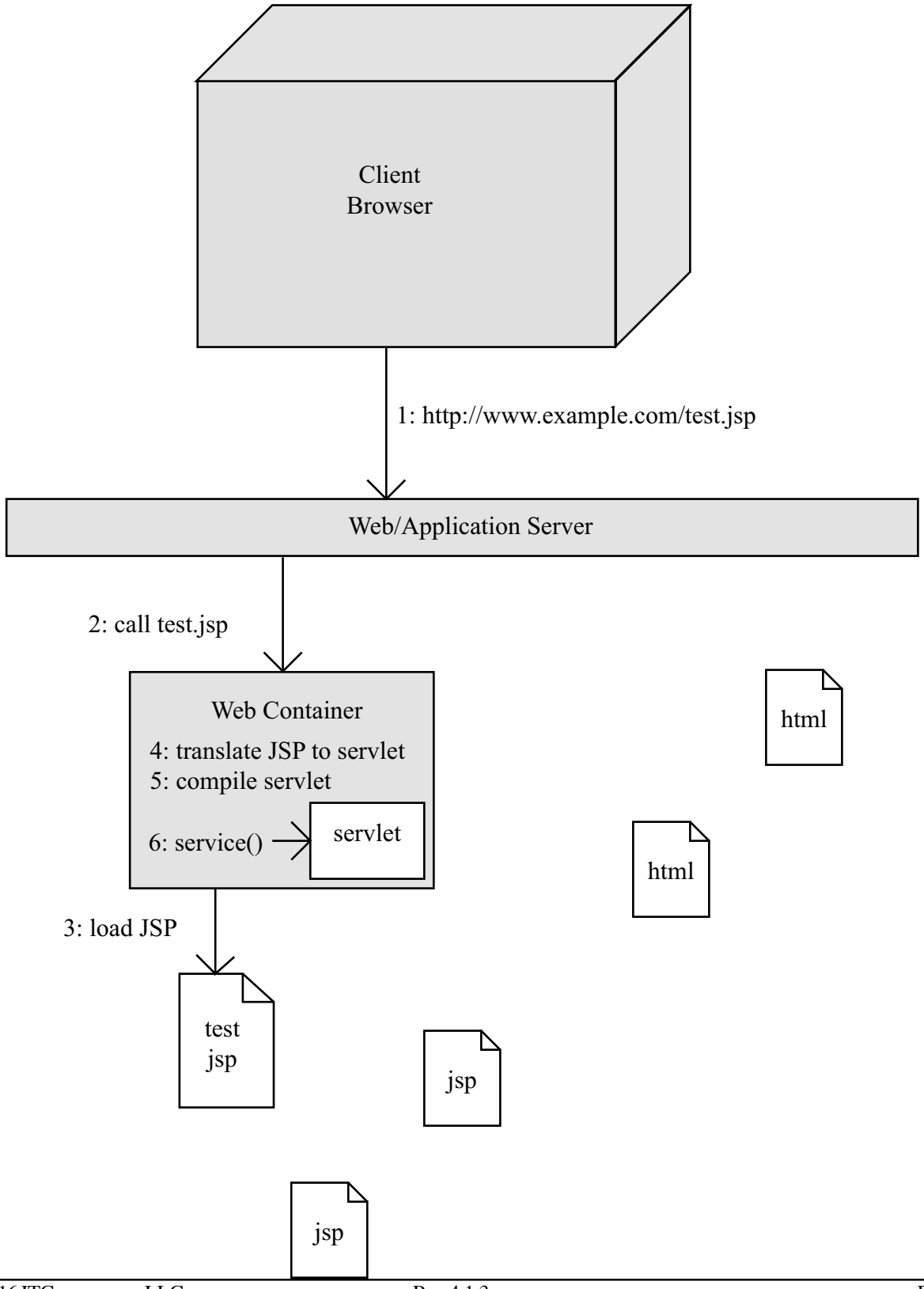
In the early days of JSP, the popular architecture was what is now referred to as "Model 1." In this architecture, a browser request is handled directly by a JSP file, which, in turn, creates JavaBeans to access the business objects. The more modern architecture described on the facing pageis referred to as the "Model 2" architecture.

In both Model 1 and Model 2 architectures, JavaBeans are the preferred mechanism for accessing business objects. The JSP specification has strong support for working with JavaBeans objects, which makes it easier to separate the display logic of the JSP file from the business logic of the application.



INTRODUCTION TO JSP

- JavaServer Pages (JSP) allows web developers to create dynamic content by combining HTML with JSP elements.
  - The simplest JSP looks just like HTML.
  - JSP elements are processed by the web container, not by the client browser.
  - Anything within your JSP page that is not a JSP element will be handled by the browser.
- Your initial request for a JSP page from your browser will result in a series of steps on the server:
  - The web container loads the JSP and dynamically translates it into a Java servlet.
  - The generated servlet is compiled into a *.class* file.
  - The web container executes the servlet's **service()** method in its own thread to generate the response back to the browser.


Client
Browser
1: http://www.example.com/test.jsp
Web/Application Server
2: call test.jsp
Web Container
4: translate JSP to servlet
5: compile servlet
6: service()
servlet
3: load JSP
test
jsp
html
html
jsp
jsp

JSP Expression Language Syntax

- JSP Expression Language (JSP EL) elements uses **${*expression*}** syntax.

```
${stock}
```

  - The value within the curly braces is usually a JavaBean that the servlet created and saved to a scope accessible by the JSP.

- Use the **.** (dot) operator to access bean properties by name.

```
${stock.price}
```

  - Specify the name of the property you wish to access after the dot.
    - The EL will automatically call the get method corresponding to the property.
  - You can navigate through a bean's nested properties by chaining multiple dots together.

- Use the **[ ]** operator when accessing data from a collection.
  - Place an integer within the square braces to access an item from an array or **List** based on its index (indexes are zero-based).

```
${cart.item[1].description}
```

  - If the collection you are working with is a **Map**, then use the name of the key as a **String** within the square braces to access its corresponding value.

```
${map["emp1234"].salary}
```


StocksWithJSP/WebContent/results.jsp

```
...
<body>
   <table border="0">
      ...
      <tr>
         <td>${stock.symbol}</td>
         <td>${stock.name}</td>
         ...
   </table>
</body>
</html>
```

If you want a comment to be visible to the client browser, use the HTML comment:

```
<!-- This is an HTML comment -->
```

A JSP comment is not compiled into the servlet and is never visible to the client browser.

```
<%-- This is a JSP comment --%>
```

JSP EL Literals
Boolean — Use the **true** and **false** literals to represent boolean content.
Integer — The rules for integer literals in EL are the same as for Java.
Floating Point — The rules for floating point literals in EL are also the same as for Java.
String — To embed literal strings in an EL expression, surround them with single or double quotes.
Null — Use the **null** literal to represent null content.

JSP EL Operators
Binary arithmetic operators: **+**, **-**, *****, **/**, **div**, **%**, **mod**
Logical binary operators: **and**, **or**, **&&**, **||**
Unary negation operators: **not**, **!**
Binary relational operators: **<**, **<=**, **==**, **>=**, **>**, **lt**, **le**, **eq**, **ge**, **gt**
Conditional/Ternary operator: **?:**
Empty operator: **empty**.
The **empty** operator checks to see if a value is **null** or empty, and, if so, evaluates to **true**.

The EL is very forgiving. It will convert the variable to the correct type based on what you are trying to do with it. **NullPointerException**s and **ArrayIndexOutOfBoundsException**s are not thrown; instead, the expression will yield an empty string.

CALLING A JSP

- Execute a JSP with a **RequestDispatcher**.
  - Obtain the **RequestDispatcher** from the **ServletContext**.

```
ServletContext context = getServletContext();
RequestDispatcher dispatcher =
   context.getRequestDispatcher("/jspName");
```

  - Call the **forward()** method on the **RequestDispatcher**.

```
dispatcher.forward(request, response);
```

    - You can not modify the response in any way after you have forwarded to a JSP.

- The calling servlet and JSP share the request object.
  - Use request attributes to share data between servlets and JSPs.
- You can set attributes on the request with the **setAttribute()** method.

```
request.setAttribute("stockPrice", 150.5);
```

  - Pass the attribute name as a **String** and the value as an **Object**.
  - There can only be one attribute with a given name on the request at a time.
- Access the attribute within JSP EL by name.

```
${stockPrice}
```

  - You can optionally use the **requestScope** implicit object for clarity.

```
${requestScope.stockPrice}
```


StocksWithJSP/src/web/StockServlet.java

```
package web;
...
@WebServlet("/Stocks")
public class StockServlet extends HttpServlet {
   ...
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {
      String symbol = req.getParameter("symbol");
      Stock stock = stockDAO.getStock(symbol);
      if (stock != null) {
         req.setAttribute("stock", stock);
         req.getRequestDispatcher("/results.jsp").forward(req, resp);
      }
      else {
         req.getRequestDispatcher("/error.jsp").forward(req, resp);
      }
   }
}
```

StocksWithJSP/WebContent/results.jsp

```
...
<body>
   <table border="0">
      ...
      <tr>
         <td>${stock.symbol}</td>
         <td>${stock.name}</td>
         ...
   </table>
</body>
</html>
```

The name you choose in the **setAttribute()** method is the same name you use in your JSP EL.

JSTL – CONDITIONALS

- The JSP Standard Tag Library (JSTL) was developed to encapsulate common functionality, such as conditionals and iteration into re-usable actions.

- To use the tags, you must add the **taglib** directive to specify the tag library **uri** and **prefix**.

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

- For a simple **if** statement, use the **<c:if>** tag (there is no **else** tag).

```
<c:if test="${boolean expression}">
  <%-- content to include if the conditional is true --%>
</c:if>
```

- For more complex conditionals, use the **<c:choose>** tag.

  - Use **<c:when>** tags for each condition and an optional **<c:otherwise>** tag for any conditions that do not match.

```
<c:choose>
    <c:when test="${boolean expression}">
      <%-- ... --%>
    </c:when>
    <c:when test="${boolean expression}">
      <%-- ... --%>
    </c:when>
    <c:otherwise>
      <%-- ... --%>
    </c:otherwise>
</c:choose>
```


StocksWithJSP/WebContent/results.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!DOCTYPE html>
<html>
<head>
<title>Stocks</title>
</head>
<body>
   <table border="0">
      <tr>
         <th>symbol</th>
         <th>name</th>
         <th>price</th>
         <th>open</th>
         <th>trend</th>
      </tr>
      <tr>
         <td>${stock.symbol}</td>
         <td>${stock.name}</td>
         ...
         <c:choose>
            <c:when test="${stock.price < stock.openPrice}">
               <td><img src="images/downarrow.png" /></td>
            </c:when>
            <c:otherwise>
               <td><img src="images/uparrow.png" /></td>
            </c:otherwise>
         </c:choose>
      </tr>
   </table>
</body>
</html>
```

JSTL – ITERATION

- Iteration can be accomplished by using the **<c:forEach>** tag.
  - Specify the loop variable using the **var** attribute.
  - The **items** attribute is used to indicate what to iterate over.
    - **Iterator**, **List**, **Set**, **Map**, and arrays are all allowed datatypes.

```
<c:forEach var="item"
    items="${cartBean.contents}">
    <tr>
      <td>${item.id}</td>
      <td>${item.quantity}</td>
    </tr>
</c:forEach>
```


StocksWithJSP/WebContent/select.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
   <title>Stocks</title>
</head>
<body>
   <form action="Stocks" method="POST">
      <c:forEach var="stock" items="${stocks}">
         <input type="radio"
                name="symbol"
                value="${stock.symbol}"/>${stock.name}<br/>
      </c:forEach>
      <input type="submit" value="Submit" />
   </form>
</body>
</html>
```

JSTL – FORMATTING

- JSTL provides a set of tags to assist with Internationalization (I18N) of your JSP documents.
  - These tags are also referred to as *formatting tags*.
  - To use these tags, you must add a different **taglib** directive to specify the tag library **uri** and **prefix**.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt"
  prefix="fmt" %>
```

- To format a number, use the **<fmt:formatNumber>** tag.
  - Use the **value** attribute to specify the number to format.
    - The **value** can contain JSP EL.
  - Use the **type** attribute to specify how to format.
    - Possible values are **currency**, **percent**, and **number**.

```
<fmt:formatNumber value="18.99" type="currency"/>
```

- To format a date, use the **<fmt:formatDate>** tag.
  - For the **type** attribute specify either **date**, **time**, or **both**.

StocksWithJSP/WebContent/results.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>
<!DOCTYPE html>
<html>
<head>
<title>Stocks</title>
</head>
<body>
   <table border="0">
      <tr>
         <th>symbol</th>
         <th>name</th>
         <th>price</th>
         <th>open</th>
         <th>trend</th>
      </tr>
      <tr>
         <td>${stock.symbol}</td>
         <td>${stock.name}</td>
         <td>
            <fmt:formatNumber value="${stock.price}"
                              type="currency" />
         </td>
         <td>
            <fmt:formatNumber value="${stock.openPrice}"
                              type="currency"/>
         </td>
         <c:choose>
            <c:when test="${stock.price < stock.openPrice}">
               <td><img src="images/downarrow.png" /></td>
            </c:when>
            <c:otherwise>
               <td><img src="images/uparrow.png" /></td>
            </c:otherwise>
         </c:choose>
      </tr>
   </table>
</body>
</html>
```

LABS

❶ Modify your version of the Stock web application to use JSPs for the view.

❷ Modify your Lottery web application so that it builds a java.util.List of ten winning numbers and stores them as an attribute on the HttpRequest object. Use JSP and JSTL in the view to present the winning numbers.

❸ Modify your Calculator web application to use JSPs for the view. If the result of a calculation is negative, display it in red otherwise display it in black.

CHAPTER 5 - SESSION AND APPLICATION SCOPE

OBJECTIVES

- Store user specific data in an **HttpSession** object.
- Access session scoped data in a JSP.

SHARING DATA BETWEEN SERVLETS AND JSPS

- You can share data between your Servlet and JSP using one of three different objects:
  - **ServletRequest**
  - **HttpSession**
  - **ServletContext**
- Use a **ServletRequest** or its subclass **HttpServletRequest** when the data to transfer is short lived.
  - Data stored as a request attribute is only available within the current request.
- Use an **HttpSession** when the data should be maintained on behalf of a particular user across requests.
  - By default, **HttpSession** uses a cookie to store a session identifier that is associated with an instance of **HttpSession**.
- Store global data within the **ServletContext**.
  - This data is available to all Servlets/JSPs for all users of a web application and should be used with caution.
- Call the **setAttribute(*key*, *value*)** method on any of the three aforementioned objects to actually store the data to the object.
  - The key must be a **String**, but the value can be any **Object**.

Each request to a Servlet or JSP is invoked on a separate thread. Therefore, any shared data is susceptible to race conditions. Be especially cautious when accessing Servlet fields, ServletContext attributes, and HttpSession attributes making sure to use read-only access or synchronization appropriately.

BEAN SCOPES IN JSPS

- JSP Expression Language can access any variable stored within the page, request, session, or application scope by name.
  - **page** scope is essentially local.
  - **request** and **session** scopes correspond to **ServletRequest** and **HttpSession**, respectively.
  - Use **application** scope when you wish to access data stored in the **ServletContext**.
- EL first searches within the page scope; if the variable cannot be found there, and then look in each of the other three scopes in turn.

```
${personBean}
```

  - Once the variable has been found, EL will not continue to search for it under any other scopes.
  - If the variable can not be found under any of the four scopes, then its value will be **null**.
- You can also explicitly access a variable in a particular scope using an implicit object.

```
${sessionScope.personBean}
```

  - Available implicit objects include **pageScope**, **requestScope**, **sessionScope**, and **applicationScope**.

StocksWithSessions/WebContent/select.jsp

```
...
<body>
   <form action="Stocks" method="POST">
      <c:forEach var="stock"
                  items="${applicationScope.stockDAO.allStocks}">
         <input type="radio"
                name="symbol"
                value="${stock.symbol}"/>${stock.name}<br />
      </c:forEach>
      <input type="submit" value="Submit" />
   </form>
   <c:if test="${! empty(sessionScope.stocks)}">
      <table border="0">
         <tr>
            <th>symbol</th>
            <th>name</th>
            <th>price</th>
            <th>open</th>
            <th>trend</th>
         </tr>
         <c:forEach var="stock" items="${sessionScope.stocks}">
            <tr>
               <td>${stock.symbol}</td>
               <td>${stock.name}</td>
               <td>
                  <fmt:formatNumber value="${stock.price}"
                                    type="currency" />
               </td>
               <td>
                  <fmt:formatNumber value="${stock.openPrice}"
                                    type="currency" />
               </td>
               <c:choose>
                  <c:when test="${stock.price < stock.openPrice}">
                     <td><img src="images/downarrow.png"/></td>
                  </c:when>
                  <c:otherwise>
                     <td><img src="images/uparrow.png"/></td>
                  </c:otherwise>
               </c:choose>
            </tr>
         </c:forEach>
      </table>
   </c:if>
</body>
</html>
```

HTTPSESSION

- Since HTTP is a stateless protocol, tracking a user from one page to another within your website requires developers to make use of creative solutions, such as cookies, hidden form fields, and URL rewriting.

- The **javax.servlet.http.HttpSession** class simplifies session tracking for Java web applications.

- Retrieve the session object from the **HttpServletRequest** object:

```
HttpSession session = request.getSession();
```

  - If the session does not already exist, the request creates it.

- Set attributes on the session as a *key, value* pair.

```
session.setAttribute("name", "Jane");
```

- Set a timeout for the session with the **setMaxInactiveInterval()** method passing in the time in seconds.

```
session.getMaxInactiveInterval(600);
```

  - If the session has not been accessed before the time expires it will be invalidated and any objects bound to it will be unbound.

  - You can proactively destroy the session object by calling **invalidate()** on it.

```
session.invalidate();
```


StocksWithSessions/src/web/StockFileDAO.java

```
...
@WebServlet("/Stocks")
public class StockServlet extends HttpServlet {
   ...
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {
      ...
      HttpSession session = req.getSession();
      if (session.getAttribute("stocks") == null) {
         // first time
         session.setAttribute("stocks", new HashSet<Stock>());
      }

      Stock stock = stockDAO.getStock(symbol);
      if (stock != null) {
         Set<Stock> stockList =
               (Set<Stock>) session.getAttribute("stocks");
         stockList.add(stock);
         req.getRequestDispatcher("/select.jsp").forward(req, resp);
      }
      else {
         req.getRequestDispatcher("/error.jsp").forward(req, resp);
      }
   }
}
```

StocksWithSessions/WebContent/select.jsp

```
...
         <c:forEach var="stock" items="${sessionScope.stocks}">
            <tr>
               <td>${stock.symbol}</td>
               <td>${stock.name}</td>
               <td>
                  <fmt:formatNumber value="${stock.price}"
                                    type="currency" />
               </td>
               <td>
                  <fmt:formatNumber value="${stock.openPrice}"
                                    type="currency" />
               </td>
               ...
            </tr>
         </c:forEach>
...
```

SERVLETCONTEXT

- A **ServletContext** allows a servlet to work with its application.
  - There is one **ServletContext** for each web application per container.
- Call the **getServletContext()** method on your Servlet object to retrieve the context.

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

- You can set and retrieve attributes on the **ServletContext** that are available to any servlet or JSP in the web application using **getAttribute()** and **setAttribute()**, like you would with **HttpServletRequest** or **HttpSession**.
- You can also use **ServletContext**'s **getResourceAsStream()** method to retrieve an **InputStream** for reading in files within your web application.

StocksWithSessions/src/web/StockFileDAO.java

```
...
@WebServlet("/Stocks")
public class StockServlet extends HttpServlet {
   @Override
   public void init() throws ServletException {
      ServletContext context = getServletContext();
      StockDAO stockDAO = new StockFileDAO(context);
      context.setAttribute("stockDAO", stockDAO);
   }
   ...
   @Override
   public void doPost(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {
      String symbol = req.getParameter("symbol");
      ServletContext context = getServletContext();
      StockDAO stockDAO = (StockDAO)context.getAttribute("stockDAO");

      HttpSession session = req.getSession();
      ...
   }
}
```

StocksWithSessions/WebContent/select.jsp

```
...
<body>
   <form action="Stocks" method="POST">
      <c:forEach var="stock"
                  items="${applicationScope.stockDAO.allStocks}">
         <input type="radio"
                name="symbol"
                value="${stock.symbol}"/>${stock.name}<br />
      </c:forEach>
      <input type="submit" value="Submit" />
   </form>
   ...
</body>
</html>
```

LABS

❶ Modify your Lottery application to display all of the previous lottery numbers that were generated for the current user as well as the newest set of numbers.

❷ Create a new web application to display president information. First, read the data from **presidents.csv** into a java.util.List of President objects. Next create an HTML form that retrieves the president's term number from the user. Using the term number, have a Servlet send the President object to a JSP for presentation.

❸ Modify your President application so that the user can click next and previous buttons to scroll through all of the presidents.

❹ Find images on the Internet for each president and use them as part of your view.

❺ Use any additional HTML/CSS knowledge you have gained to improve the presentation of your application.

INDEX

