

JAVASERVER PAGES

Objectives

- To create a simple JSP page (§38.2).
- To explain how a JSP page is processed (§38.3).
- To use JSP constructs to code JSP script (§38.4).
- To use predefined variables and directives in JSP (§§38.5–38.6).
- To use JavaBeans components in JSP (§38.7).
- To get and set JavaBeans properties in JSP (§38.8).
- To associate JavaBeans properties with input parameters (§38.9).
- To forward requests from one JSP page to another (§38.10).
- To develop an application for browsing database tables using JSP (§38.11).

CHAPTER 38





38.1 Introduction

JavaServer Pages are the Java scripts and code embedded in an HTML file.

Servlets can be used to generate dynamic Web content. One drawback, however, is that you have to embed HTML tags and text inside the Java source code. Using servlets, you have to modify the Java source code and recompile it if changes are made to the HTML text. If you have a lot of HTML script in a servlet, the program is difficult to read and maintain, since the HTML text is part of the Java program. JavaServer Pages (JSP) was introduced to remedy this drawback. JSP enables you to write regular HTML script in the normal way and embed Java code to produce dynamic content.



38.2 Creating a Simple JSP Page

An IDE such as NetBeans is an effective tool for creating JavaServer Pages.

JSP provides an easy way to create dynamic webpages and simplify the task of building Web applications. A JavaServer page is like a regular HTML page with special tags, known as *JSP tags*, which enable the Web server to generate dynamic content. You can create a webpage with HTML script and enclose the Java code for generating dynamic content in the JSP tags. Here is an example of a simple JSP page:

```
<!-- CurrentTime.jsp -->
<html>
  <head>
    <title>
      CurrentTime
    </title>
  </head>
  <body>
    Current time is <%= new java.util.Date() %>
  </body>
</html>
```

The dynamic content is enclosed in the tag that begins with `<%=` and ends with `%>`. The current time is returned as a string by invoking the `toString` method of an object of the `java.util.Date` class.

An IDE like NetBeans can greatly simplify the task of developing JSP. To create JSP in NetBeans, first you need to create a Web project. A Web project named **liangweb** was created in the preceding chapter. For convenience, this chapter will create JSP in the **liangweb** project.

Here are the steps to create and run `CurrentTime.jsp`:

1. Right-click the **liangweb** node in the project pane and choose **New, JSP** to display the New JSP dialog box, as shown in Figure 38.1.
2. Enter **CurrentTime** in the JSP File Name field and click *Finish*. You will see `CurrentTime.jsp` appearing under the webpages node in **liangweb**.
3. Complete the code for `CurrentTime.jsp`, as shown in Figure 38.2.
4. Right-click `CurrentTime.jsp` in the project pane and choose *Run File*. You will see the JSP page displayed in a Web browser, as shown in Figure 38.3.



Note

Like servlets, you can develop JSP in NetBeans, create a `.war` file, and then deploy the `.war` file in a Java Web server such as Tomcat and GlassFish.

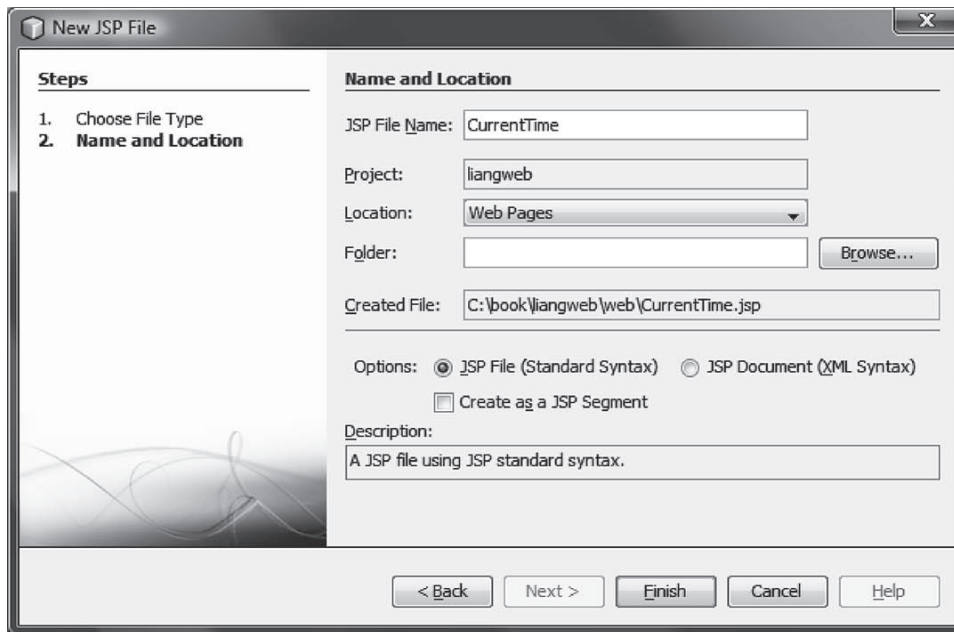


FIGURE 38.1 You can create a JSP page using NetBeans.

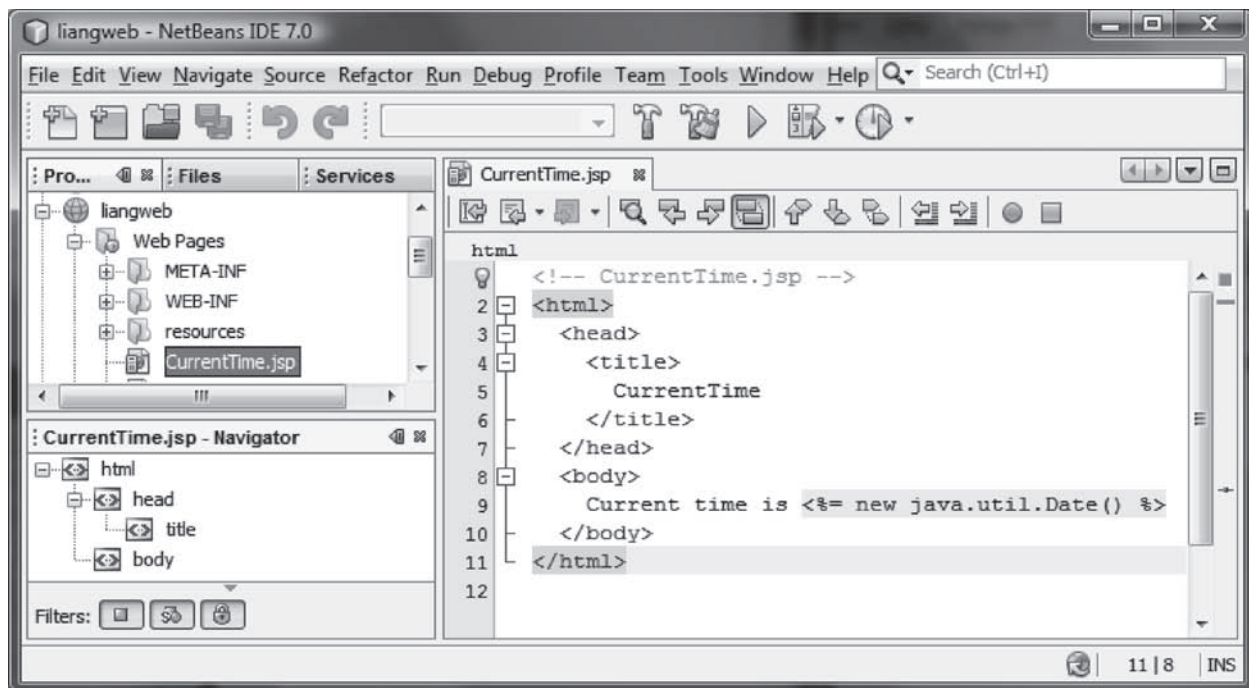


FIGURE 38.2 A template for a JSP page is created.

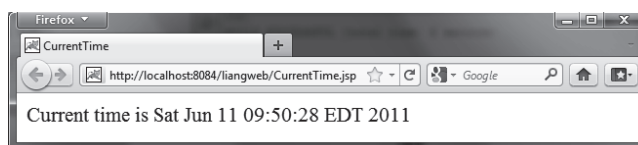


FIGURE 38.3 The result from a JSP page is displayed in a Web browser.

Key
Point

38.3 How Is a JSP Page Processed?

JavaServer Pages are preprocessed and compiled into Java servlets by a Java Web server.

A JSP page must first be processed by a Web server before it can be displayed in a Web browser. The Web server must support JSP, and the JSP page must be stored in a file with a .jsp extension. The Web server translates the .jsp file into a Java servlet, compiles the servlet, and executes it. The result of the execution is sent to the browser for display. Figure 38.4 shows how a JSP page is processed by a Web server.



Note

A JSP page is translated into a servlet when the page is requested for the first time. It is not retranslated if the page is not modified. To ensure that the first-time real user does not encounter a delay, JSP developers should test the page after it is installed.

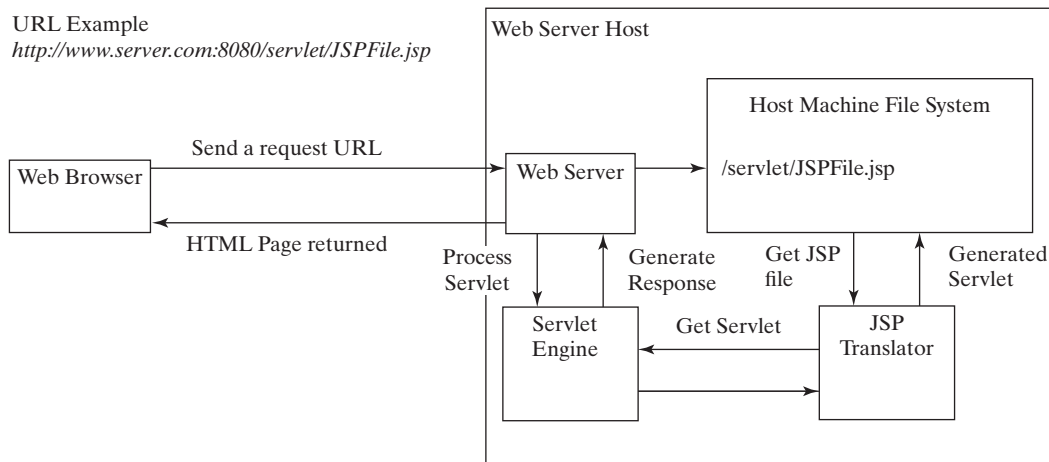


FIGURE 38.4 A JSP page is translated into a servlet.

Check
Point

- 38.2.1** What is the file-name extension of a JavaServer page? How is a JSP page processed?
- 38.2.2** Can you create a .war that contains JSP in NetBeans? Where should the .war be placed in a Java application server?
- 38.2.3** You can display an HTML file (e.g., c:\test.html) by typing the complete file name in the Address field of Internet Explorer. Why can't you display a JSP file by simply typing the file name?

Key
Point

38.4 JSP Scripting Constructs

There are three main types of JSP constructs: scripting constructs, directives, and actions.

Scripting elements enable you to specify Java code that will become part of the resultant servlet. *Directives* enable you to control the overall structure of the resultant servlet. *Actions* enable you to control the behavior of the JSP engine. This section introduces scripting constructs.

Three types of JSP scripting constructs can be used to insert Java code into a resultant servlet: expressions, scriptlets, and declarations.

A JSP *expression* is used to insert a Java expression directly into the output. It has the following form:

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

The expression is evaluated, converted into a string, and sent to the output stream of the servlet.

A JSP *scriptlet* enables you to insert a Java statement into the servlet's `jspService` method, which is invoked by the `service` method. A JSP scriptlet has the following form:

```
<% Java statement %>
```

A JSP *declaration* is for declaring methods or fields into the servlet. It has the following form:

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

HTML comments have the following form:

```
<!-- HTML Comment -->
```

If you don't want the comment to appear in the resultant HTML file, use the following comment in JSP:

```
<%-- JSP Comment --%>
```

Listing 38.1 creates a JavaServer page that displays factorials for numbers from 0 to 10, as shown in Figure 38.5.

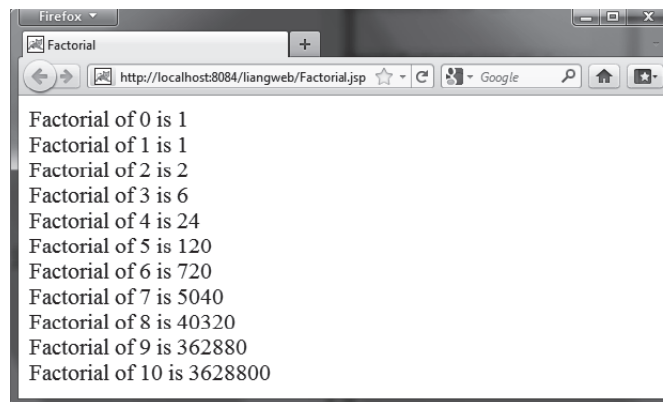


FIGURE 38.5 The JSP page displays factorials.

LISTING 38.1 Factorial.jsp

```

1  <html>
2  <head>
3  <title>
4  Factorial
5  </title>
6  </head>
7  <body>
8
9  <% for (int i = 0; i <= 10; i++) { %>
10     Factorial of <%= i %> is
11     <%= computeFactorial(i) %> <br />
12 <% } %>
13
14 <%! private long computeFactorial(int n) {
15     if (n == 0)
16         return 1;
17     else
18         return n * computeFactorial(n - 1);

```

```

19     }
20     %>
21
22     </body>
23 </html>

```

JSP scriptlets are enclosed between `<%` and `%>`. Thus,

```
for (int i = 0; i <= 10; i++) {, (line 9)
```

is a scriptlet and as such is inserted directly into the servlet's `jspService` method.

JSP expressions are enclosed between `<%=` and `%>`. Thus,

```
<%= i %>, (line 10)
```

is an expression and is inserted into the output stream of the servlet.

JSP declarations are enclosed between `<%!` and `%>`. Thus,

```

<%! private long computeFactorial(int n) {
    ...
}
%>

```

is a declaration that defines methods or fields in the servlet.

What will be different if line 9 is replaced by the two alternatives shown below? Both work fine, but there is an important difference. In (a), `i` is a local variable in the servlet, whereas in (b), `i` is an instance variable when translated to the servlet.

```

<% int i = 0; %>
<% for ( ; i <= 10; i++) { %>

```

(a)

```

<%! int i; %>
<% for (i = 0; i <= 10; i++) { %>

```

(b)

**Caution:**

For JSP, the loop body, even though it contains a single statement, must be placed inside braces. It would be wrong to delete the opening brace (`{`) in line 9 and the closing brace (`<% } %>`) in line 12.

**Caution:**

There is no semicolon at the end of a JSP expression. For example, `<%= i; %>` is incorrect. But there must be a semicolon for each Java statement in a JSP scriptlet. For example, `<% int i = 0 %>` is incorrect.

**Caution:**

JSP and Java elements are case sensitive, but HTML is not.



38.4.1 What are a JSP expression, a JSP scriptlet, and a JSP declaration? How do you write these constructs in JSP?

38.4.2 Find three syntax errors in the following JSP code:

```

<%! int k %>
<% for (int j = 1; j <= 9; j++) %>
    <%= j; %> <br />

```

38.4.3 In the following JSP, which variables are instance variables, and which are local variables when it is translated into the servlet?

```

<%! int k; %>
<%! int i; %>

```

```

<% for (int j = 1; j <= 9; j++) k += 1;%>
<%= k><br /> <%= i><br /> <%= getTime()><br />
<% private long getTime() {
    long time = System.currentTimeMillis();
    return time;
} %>

```

38.5 Predefined Variables

JSP provides predefined variables that can be conveniently used in the JSP code.

You can use variables in JSP. For convenience, JSP provides eight predefined variables from the servlet environment that can be used with JSP expressions and scriptlets. These variables are also known as *JSP implicit objects*.



- **request** represents the client's request, which is an instance of **HttpServletRequest**. You can use it to access request parameters and HTTP headers, such as cookies and host name.
- **response** represents the servlet's response, which is an instance of **HttpServletResponse**. You can use it to set response type and send output to the client.
- **out** represents the character output stream, which is an instance of **PrintWriter** obtained from **response.getWriter()**. You can use it to send character content to the client.
- **session** represents the **HttpSession** object associated with the request, obtained from **request.getSession()**.
- **application** represents the **ServletContext** object for storing persistent data for all clients. The difference between **session** and **application** is that session is tied to one client, but **application** is for all clients to share persistent data.
- **config** represents the **ServletConfig** object for the page.
- **pageContext** represents the **PageContext** object. **PageContext** is a new class introduced in JSP to give a central point of access to many page attributes.
- **page** is an alternative to **this**.

As an example, let us write an HTML page that prompts the user to enter loan amount, annual interest rate, and number of years, as shown in Figure 38.6a. Clicking the *Compute Loan Payment* button invokes a JSP to compute and display the monthly and total loan payments, as shown in Figure 38.6b.

The HTML file is named **ComputeLoan.html** (Listing 38.2). The JSP file is named **ComputeLoan.jsp** (Listing 38.3).



FIGURE 38.6 The JSP computes the loan payments.

LISTING 38.2 ComputeLoan.html

```

1  <!-- ComputeLoan.html -->
2  <html>
3    <head>
4      <title>ComputeLoan</title>
5    </head>
6    <body>
7      <form method = "get" action = "ComputeLoan.jsp">
8        Compute Loan Payment<br />
9        Loan Amount
10       <input type = "text" name = "loanAmount" /><br />
11       Annual Interest Rate
12       <input type = "text" name = "annualInterestRate" /><br />
13       Number of Years
14       <input type = "text" name = "numberOfYears" size = "3" /><br />
15       <p><input type = "submit" name = "Submit"
16         value = "Compute Loan Payment" />
17       <input type = "reset" value = "Reset" /></p>
18     </form>
19   </body>
20 </html>

```

LISTING 38.3 ComputeLoan.jsp

```

1  <!-- ComputeLoan.jsp -->
2  <html>
3    <head>
4      <title>ComputeLoan</title>
5    </head>
6    <body>
7      <% double loanAmount = Double.parseDouble(
8        request.getParameter("loanAmount"));
9        double annualInterestRate = Double.parseDouble(
10       request.getParameter("annualInterestRate"));
11       double numberOfYears = Integer.parseInt(
12       request.getParameter("numberOfYears"));
13       double monthlyInterestRate = annualInterestRate / 1200;
14       double monthlyPayment = loanAmount * monthlyInterestRate /
15       (1 - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
16       double totalPayment = monthlyPayment * numberOfYears * 12; %>
17     Loan Amount: <%= loanAmount %><br />
18     Annual Interest Rate: <%= annualInterestRate %><br />
19     Number of Years: <%= numberOfYears %><br />
20     <b>Monthly Payment: <%= monthlyPayment %><br />
21     Total Payment: <%= totalPayment %><br /></b>
22   </body>
23 </html>

```

ComputeLoan.html is displayed first to prompt the user to enter the loan amount, annual interest rate, and number of years. Since this file does not contain any JSP elements, it is named with an .html extension as a regular HTML file.

ComputeLoan.jsp is invoked upon clicking the *Compute Loan Payment* button in the HTML form. The JSP page obtains the parameter values using the predefined variable **request** in lines 7–12 and computes monthly payment and total payment in lines 13–16. The formula for computing monthly payment is given in Listing 2.9, *ComputeLoan.java*.

What is wrong if the JSP scriptlet **<%** in line 7 is replaced by the JSP declaration **<%!?** The predefined variables (e.g., **request**, **response**, and **out**) correspond to local variables

defined in the servlet methods **doGet** and **doPost**. They must appear in JSP scriptlets, not in JSP declarations.



Tip

ComputeLoan.jsp can also be invoked using the following query string: `http://localhost:8084/liangweb/ComputeLoan.jsp?loanAmount=10000&annualInterestRate=6&numberOfYears=15`.

- 38.5.1** Describe the predefined variables in JSP.
- 38.5.2** What is wrong if the JSP scriptlet `<%` in line 7 in **ComputeLoan.jsp** (Listing 38.3) is replaced by JSP declaration `<%!?`
- 38.5.3** Can you use predefined variables (e.g., **request**, **response**, and **out**) in JSP declarations?



38.6 JSP Directives

You can use JSP directives to instruct JSP engine on how to process the JSP code.

A JSP directive is a statement that gives the JSP engine information about the JSP page. For example, if your JSP page uses a Java class from a package other than the **java.lang** package, you have to use a directive to import this package. The general syntax for a JSP directive is shown below:



```
<%@ directive attribute = "value" %>, or
<%@ directive attribute1 = "value1"
      attribute2 = "value2"
      ...
      attributen = "valuen" %>
```

The possible directives are:

- **page** lets you provide information for the page, such as importing classes and setting up content type. The **page** directive can appear anywhere in the JSP file.
- **include** lets you insert a file into the servlet when the page is translated to a servlet. The **include** directive must be placed where you want the file to be inserted.
- **taglib** lets you define custom tags.

The following are useful attributes for the **page** directive:

- **import** specifies one or more packages to be imported for this page. For example, the directive `<%@ page import="java.util.*", java.text.*" %>` imports **java.util.*** and **java.text.***.
- **contentType** specifies the content type for the resultant JSP page. By default, the content type is **text/html** for JSP. The default content type for servlets is **text/plain**.
- **session** specifies a **boolean** value to indicate whether the page is part of the session. By default, **session** is **true**.
- **buffer** specifies the output stream buffer size. By default, it is 8KB. For example, the directive `<%@ page buffer="10KB" %>` specifies that the output buffer size is 10KB. The directive `<%@ page buffer="none" %>` specifies that a buffer is not used.

- **autoFlush** specifies a **boolean** value to indicate whether the output buffer should be automatically flushed when it is full or whether an exception should be raised when the buffer overflows. By default, this attribute is **true**. In this case, the buffer attribute cannot be **none**.
- **isThreadSafe** specifies a **boolean** value to indicate whether the page can be accessed simultaneously without data corruption. By default, it is **true**. If it is set to **false**, the JSP page will be translated to a servlet that implements the **SingleThreadModel** interface.
- **errorPage** specifies a JSP page that is processed when an exception occurs in the current page. For example, the directive `<%@ page errorPage="HandleError.jsp" %>` specifies that `HandleError.jsp` is processed when an exception occurs.
- **isErrorPage** specifies a **boolean** value to indicate whether the page can be used as an error page. By default, this attribute is **false**.

Listing 38.4 gives an example that shows how to use the page directive to import a class. The example uses the **Loan** class created in Listing 10.2, `Loan.java`, to simplify Listing 38.3, `ComputeLoan.jsp`. You can create an object of the **Loan** class and use its `monthlyPayment()` and `totalPayment()` methods to compute the monthly payment and total payment.

LISTING 38.4 ComputeLoan1.jsp

```

1  <!-- ComputeLoan1.jsp -->
2  <html>
3    <head>
4      <title>ComputeLoan Using the Loan Class</title>
5    </head>
6    <body>
7      <%@ page import = "chapter38.Loan" %>
8      <% double loanAmount = Double.parseDouble(
9        request.getParameter("loanAmount"));
10     double annualInterestRate = Double.parseDouble(
11       request.getParameter("annualInterestRate"));
12     int numberOfYears = Integer.parseInt(
13       request.getParameter("numberOfYears"));
14     Loan loan =
15       new Loan(annualInterestRate, numberOfYears, loanAmount);
16     %>
17     Loan Amount: <%= loanAmount %><br />
18     Annual Interest Rate: <%= annualInterestRate %><br />
19     Number of Years: <%= numberOfYears %><br />
20     <b>Monthly Payment: <%= loan.getMonthlyPayment() %><br />
21     Total Payment: <%= loan.getTotalPayment() %><br /></b>
22   </body>
23 </html>

```

This JSP uses the **Loan** class. You need to create the class in the `liangweb` project in package `chapter38` as follows:

```

package chapter38;
public class Loan {
    // Same as lines 2-71 in Listing 10.2, Loan.java, so omitted

```

The directive `<%@ page import = "chapter38.Loan" %>` imports the **Loan** class in line 7. Line 14 creates an object of **Loan** for the given loan amount, annual interest rate, and number of years. Lines 20–21 invoke the **Loan** object's `monthlyPayment()` and `totalPayment()` methods to display monthly payment and total payment.

- 38.6.1** Describe the JSP directives and attributes for the **page** directive.
- 38.6.2** If a class does not have a package statement, can you import it?
- 38.6.3** If you use a custom class from a JSP, where should the class be placed?



38.7 Using JavaBeans in JSP

You can use JavaBeans to create objects for sharing among different JSP pages.



Normally you create an instance of a class in a program and use it in that program. This method is for sharing the class, not the object. JSP allows you to share the object of a class among different pages. To enable an object to be shared, its class must be a JavaBeans component. Recall that this entails the following three features:

- The class is public.
- The class has a public constructor with no arguments.
- The class is serializable. (This requirement is not necessary in JSP.)

To create an instance for a JavaBeans component, use the following syntax:

```
<jsp:useBean id = "objectName" scope = "scopeAttribute"
class = "ClassName" />
```

This syntax is roughly equivalent to

```
<% ClassName objectName = new ClassName() %>
```

except that the **scope** attribute is missing. The scope attribute specifies the scope of the object, and the object is not recreated if it is already within the scope. Listed below are four possible values for the scope attribute:

- **application** specifies that the object is bound to the application. The object can be shared by all sessions of the application.
- **session** specifies that the object is bound to the client's session. Recall that a client's session is automatically created between a Web browser and a Web server. When a client from the same browser accesses two servlets or two JSP pages on the same server, the session is the same.
- **page** is the default scope, which specifies that the object is bound to the page.
- **request** specifies that the object is bound to the client's request.

When `<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" />` is processed, the JSP engine first searches for an object of the class with the same id and scope. If found, the preexisting bean is used; otherwise, a new bean is created.

Here is another syntax for creating a bean:

```
<jsp:useBean id = "objectName" scope = "scopeAttribute"
class = "ClassName" >
    statements
</jsp:useBean>
```

The statements are executed when the bean is created. If a bean with the same ID and class name already exists in the scope, the statements are not executed.

Listing 38.5 creates a JavaBeans component named **Count** and uses it to count the number of visits to a JSP page, as shown in Figure 38.7.

38-12 Chapter 38 Javasever Pages

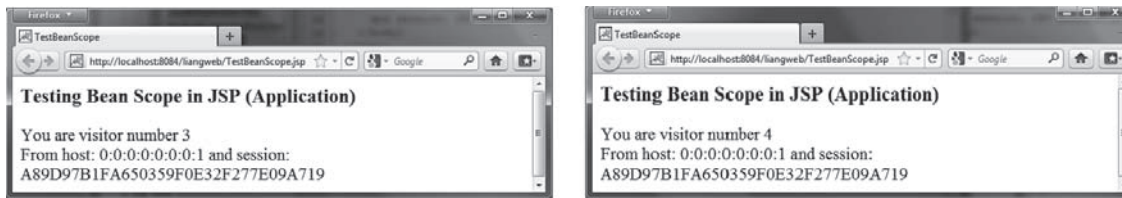


FIGURE 38.7 The number of visits to the page is increased when the page is visited.

LISTING 38.5 Count.java

```
1 package chapter38;
2
3 public class Count {
4     private int count = 0;
5
6     /** Return count property */
7     public int getCount() {
8         return count;
9     }
10
11    /** Increase count */
12    public void increaseCount() {
13        count++;
14    }
15 }
```

The JSP page named TestBeanScope.jsp is created in Listing 38.6.

LISTING 38.6 TestBeanScope.jsp

```
1 <-- TestBeanScope.jsp -->
2 <%@ page import = "chapter38.Count" %>
3 <jsp:useBean id = "count" scope = "application"
4     class = "chapter38.Count">
5 </jsp:useBean>
6 <html>
7     <head>
8         <title>TestBeanScope</title>
9     </head>
10    <body>
11        <h3>Testing Bean Scope in JSP (Application)</h3>
12        <% count.increaseCount(); %>
13        You are visitor number <%= count.getCount() %><br />
14        From host: <%= request.getRemoteHost() %>
15        and session: <%= session.getId() %>
16    </body>
17 </html>
```

The `scope` attribute specifies the scope of the bean. `scope="application"` (line 3) specifies that the bean is alive in the JSP engine and available for all clients to access. The bean can be shared by any client with the directive `<jsp:useBean id="count" scope="application" class="Count">` (lines 3–4). Every client accessing TestBeanScope.jsp causes the count to increase by 1. The first client causes `count` object to be created, and subsequent access to TestBeanScope uses the same object.

If `scope="application"` is changed to `scope="session"`, the scope of the bean is limited to the session from the same browser. The count will increase only if the page is requested from the same browser. If `scope="application"` is changed to `scope="page"`,

the scope of the bean is limited to the page, and any other page cannot access this bean. The page will always display count 1. If `scope="application"` is changed to `scope="request"`, the scope of the bean is limited to the client's request, and any other request on the page will always display count 1.

If the page is destroyed, the count restarts from 0. You can fix the problem by storing the count in a random access file or in a database table. Assume you store the count in the `Count` table in a database. The `Count` class can be modified in Listing 38.7.

Listing 38.7 Count.java (Revised Version)

```

1 package chapter38;
2
3 import java.sql.*;
4
5 public class Count {
6     private int count = 0;
7     private Statement statement = null;
8
9     public Count() {
10         initializeJdbc();
11     }
12
13     /** Return count property */
14     public int getCount() {
15         try {
16             ResultSet rset = statement.executeQuery
17                 ("select countValue from Count");
18             rset.next();
19             count = rset.getInt(1);
20         }
21         catch (Exception ex) {
22             ex.printStackTrace();
23         }
24
25         return count;
26     }
27
28     /** Increase count */
29     public void increaseCount() {
30         count++;
31         try {
32             statement.executeUpdate(
33                 "update Count set countValue = " + count);
34         }
35         catch (Exception ex) {
36             ex.printStackTrace();
37         }
38     }
39
40     /** Initialize database connection */
41     public void initializeJdbc() {
42         try {
43             Class.forName("com.mysql.jdbc.Driver");
44
45             // Connect to the sample database
46             Connection connection = DriverManager.getConnection
47                 ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
48
49             statement = connection.createStatement();
50         }

```

```

51     catch (Exception ex) {
52         ex.printStackTrace();
53     }
54 }
55 }

```



- 38.7.1** You can create an object in a JSP scriptlet. What is the difference between an object created using the **new** operator and a bean created using the **<jsp:useBean ... >** tag?
- 38.7.2** What is the **scope** attribute for? Describe four scope attributes.
- 38.7.3** Describe how a **<jsp:useBean ... >** statement is processed by the JSP engine.



38.8 Getting and Setting Properties

JSP provides convenient syntax for getting and setting JavaBeans properties.

By convention, a JavaBeans component provides the get and set methods for reading and modifying its private properties. You can get the property in JSP using the syntax shown below:

```
<jsp:getProperty name = "beanId" property = "sample" />
```

This is roughly equivalent to

```
<%= beanId.getSample() %>
```

You can set the property in JSP using the following syntax:

```
<jsp:setProperty name = "beanId"
property = "sample" value = "test1" />
```

This is equivalent to

```
<% beanId.setSample("test1"); %>
```

38.9 Associating Properties with Input Parameters

Often properties are associated with input parameters. Suppose you want to get the value of the input parameter named **score** and set it to the JavaBeans property named **score**. You could write the following code:

```
<% double score = Double.parseDouble(
    request.getParameter("score")); %>
<jsp:setProperty name = "beanId" property = "score"
value = "<%= score %>" />
```

This is cumbersome. JSP provides a convenient syntax that can be used to simplify it:

```
<jsp:setProperty name = "beanId" property = "score"
param = "score" />
```

Instead of using the **value** attribute, you use the **param** attribute to name an input parameter. The value of this parameter is set to the property.



Note

Simple type conversion is performed automatically when a bean property is associated with an input parameter. A string input parameter is converted to an appropriate primitive

data type or a wrapper class for a primitive type. For example, if the bean property is of the `int` type, the value of the parameter will be converted to the `int` type. If the bean property is of the `Integer` type, the value of the parameter will be converted to the `Integer` type.

Often the bean property and the parameter have the same name. You can use the following convenient statement to associate all the bean properties in `beanId` with the parameters that match the property names:

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

38.9.1 Example: Computing Loan Payments Using JavaBeans

This example uses JavaBeans to simplify Listing 38.4, `ComputeLoan1.jsp`, by associating the bean properties with the input parameters. The new `ComputeLoan2.jsp` is given in Listing 38.8.

LISTING 38.8 `ComputeLoan2.jsp`

```
1  <!-- ComputeLoan2.jsp -->
2  <html>
3    <head>
4      <title>ComputeLoan Using the Loan Class</title>
5    </head>
6    <body>
7      <%@ page import = "chapter38.Loan" %>
8      <jsp:useBean id = "loan" class = "chapter38.Loan"
9        scope = "page" ></jsp:useBean>
10     <jsp:setProperty name = "loan" property = "*" />
11     Loan Amount: <%= loan.getLoanAmount() %><br />
12     Annual Interest Rate: <%= loan.getAnnualInterestRate() %><br />
13     Number of Years: <%= loan.getNumberOfYear() %><br />
14     <b>Monthly Payment: <%= loan.monthlyPayment() %><br />
15     Total Payment: <%= loan.totalPayment() %><br /></b>
16   </body>
17 </html>
```

Lines 8–9

```
<jsp:useBean id = "loan" class = "chapter38.Loan"
  scope = "page" ></jsp:useBean>
```

create a bean named `loan` for the `Loan` class. Line 10

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

associates the bean properties `loanAmount`, `annualInterestRate`, and `numberOfYears` with the input parameter values and performs type conversion automatically.

Lines 11–13 use the accessor methods of the loan bean to get the loan amount, annual interest rate, and number of years.

This program acts the same as in Listings 38.3 and 38.4, `ComputeLoan.jsp` and `ComputeLoan1.jsp`, but the coding is much, more simplified.

38.9.2 Example: Computing Factorials Using JavaBeans

This example creates a JavaBeans component named `FactorialBean` and uses it to compute the factorial of an input number in a JSP page named `FactorialBean.jsp`, as shown in Figure 38.8.

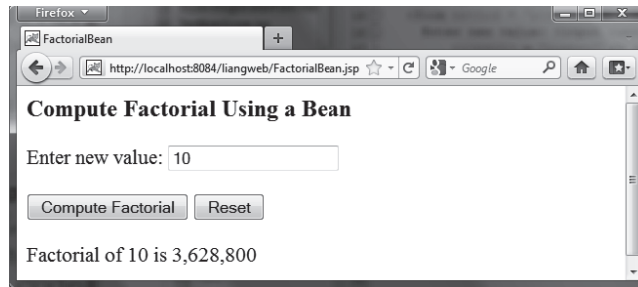


FIGURE 38.8 The factorial of an input integer is computed using a method in **FactorialBean**.

Create a JavaBeans component named **FactorialBean.java** (Listing 38.9). Create **FactorialBean.jsp** (Listing 38.10).

LISTING 38.9 FactorialBean.java

```

1 package chapter38;
2
3 public class FactorialBean {
4     private int number;
5
6     /** Return number property */
7     public int getNumber() {
8         return number;
9     }
10
11     /** Set number property */
12     public void setNumber(int newValue) {
13         number = newValue;
14     }
15
16     /** Obtain factorial */
17     public long getFactorial() {
18         long factorial = 1;
19         for (int i = 1; i <= number; i++)
20             factorial *= i;
21         return factorial;
22     }
23 }

```

LISTING 38.10 FactorialBean.jsp

```

1 <!-- FactorialBean.jsp -->
2 <%@ page import = "chapter38.FactorialBean" %>
3 <jsp:useBean id = "factorialBeanId"
4     class = "chapter38.FactorialBean" scope = "page" >
5 </jsp:useBean>
6 <jsp:setProperty name = "factorialBeanId" property = "*" />
7 <html>
8     <head>
9         <title>
10             FactorialBean
11         </title>
12     </head>
13     <body>
14         <h3>Compute Factorial Using a Bean</h3>

```

```

15 <form method = "post">
16   Enter new value: <input name = "number" /><br /><br />
17   <input type = "submit" name = "Submit"
18     value = "Compute Factorial" />
19   <input type = "reset" value = "Reset" /><br /><br />
20   Factorial of
21     <jsp:getProperty name = "factorialBeanId"
22       property = "number" /> is
23     <%@ page import = "java.text.*" %>
24     <% NumberFormat format = NumberFormat.getNumberInstance(); %>
25     <%= format.format(factorialBeanId.getFactorial()) %>
26 </form>
27 </body>
28 </html>

```

The `jsp:useBean` tag (lines 3–4) creates a bean `factorialBeanId` of the `FactorialBean` class. Line 5 `<jsp:setProperty name="factorialBeanId" property="*" />` associates all the bean properties with the input parameters that have the same name. In this case, the bean property `number` is associated with the input parameter `number`. When you click the *Compute Factorial* button, JSP automatically converts the input value for `number` from string into `int` and sets it to `factorialBean` before other statements are executed.

Lines 21–22 `<jsp:getProperty name="factorialBeanId" property="number" />` tag (line 21) is equivalent to `<%= factorialBeanId.getNumber() %>`. The method `factorialBeanId.getFactorial()` (line 25) returns the factorial for the number in `factorialBeanId`.



Design Guide

Mixing a lot of Java code with HTML in a JSP page makes the code difficult to read and to maintain. You should move the Java code to a .java file as much as you can.

Following the preceding design guide, you may improve the preceding example by moving the Java code in lines 23–25 to the `FactorialBean` class. The new `FactorialBean.java` and `FactorialBean.jsp` are given in Listings 38.11 and 38.12.

LISTING 38.11 NewFactorialBean.java

```

1 package chapter38;
2
3 import java.text.*;
4
5 public class NewFactorialBean {
6   private int number;
7
8   /** Return number property */
9   public int getNumber() {
10    return number;
11  }
12
13   /** Set number property */
14   public void setNumber(int newValue) {
15    number = newValue;
16  }
17
18   /** Obtain factorial */
19   public long getFactorial() {
20    long factorial = 1;

```

```

21     for (int i = 1; i <= number; i++)
22         factorial *= i;
23     return factorial;
24 }
25
26 /** Format number */
27 public static String format(long number) {
28     NumberFormat format = NumberFormat.getNumberInstance();
29     return format.format(number);
30 }
31 }

```

LISTING 38.12 NewFactorialBean.jsp

```

1  <!-- NewFactorialBean.jsp -->
2  <%@ page import = "chapter38.NewFactorialBean" %>
3  <jsp:useBean id = "factorialBeanId"
4      class = "chapter38.NewFactorialBean" scope = "page" >
5  </jsp:useBean>
6  <jsp:setProperty name = "factorialBeanId" property = "*" />
7  <html>
8      <head>
9          <title>
10             FactorialBean
11          </title>
12      </head>
13      <body>
14          <h3>Compute Factorial Using a Bean</h3>
15          <form method = "post">
16              Enter new value: <input name = "number" /><br /><br />
17              <input type = "submit" name = "Submit"
18                  value = "Compute Factorial" />
19              <input type = "reset" value = "Reset" /><br /><br />
20              Factorial of
21                  <jsp:getProperty name = "factorialBeanId"
22                      property = "number" /> is
23                  <%= NewFactorialBean.format(factorialBeanId.getFactorial()) %>
24          </form>
25      </body>
26  </html>

```

There is a problem in this page. The program cannot display large factorials. For example, if you entered value **21**, the program would display an incorrect factorial. To fix this problem, all you need to do is to revise the **NewFactorialBean** class using **BigInteger** to computing factorials (see Exercise 38.18).

38.9.3 Example: Displaying International Time

Listing 37.5, **TimeForm.java**, gives a Java servlet that uses the **doGet** method to generate an HTML form for the user to specify a locale and time zone (see Figure 37.18a) and uses the **doPost** method to display the current time for the specified time zone in the specified locale (see Figure 37.18b). This section rewrites the servlet using JSP. You have to create two JSP pages, one for displaying the form, and the other for displaying the current time.

In the **TimeForm.java** servlet, arrays **allLocale** and **allTimeZone** are the data fields. The **doGet** and **doPost** methods both use the arrays. Since the available locales and time zones are used in both pages, it is better to create an object that contains all available locales and time zones. This object can be shared by both pages.

Let us create a JavaBeans component named `TimeBean.java` (Listing 38.13). This class obtains all the available locales in an array in line 7 and all time zones in an array in line 8. The bean properties `localeIndex` and `timeZoneIndex` (lines 9–10) are defined to refer to an element in the arrays. The `currentTimeString()` method (lines 42–52) returns a string for the current time with the specified locale and time zone.

LISTING 38.13 `TimeBean.java`

```

1  package chapter38;
2
3  import java.util.*;
4  import java.text.*;
5
6  public class TimeBean {
7      private Locale[] allLocale = Locale.getAvailableLocales();
8      private String[] allTimeZone = TimeZone.getAvailableIDs();
9      private int localeIndex;
10     private int timeZoneIndex;
11
12     public TimeBean() {
13         Arrays.sort(allTimeZone);
14     }
15
16     public Locale[] getAllLocale() {
17         return allLocale;
18     }
19
20     public String[] getAllTimeZone() {
21         return allTimeZone;
22     }
23
24     public int getLocaleIndex() {
25         return localeIndex;
26     }
27
28     public int getTimeZoneIndex() {
29         return timeZoneIndex;
30     }
31
32     public void setLocaleIndex(int index) {
33         localeIndex = index;
34     }
35
36     public void setTimeZoneIndex(int index) {
37         timeZoneIndex = index;
38     }
39
40     /** Return a string for the current time
41      * with the specified locale and time zone */
42     public String currentTimeString(
43         int localeIndex, int timeZoneIndex) {
44         Calendar calendar =
45             new GregorianCalendar(allLocale[localeIndex]);
46         TimeZone timeZone =
47             TimeZone.getTimeZone(allTimeZone[timeZoneIndex]);
48         DateFormat dateFormat = DateFormat.getDateInstance(
49             DateFormat.FULL, DateFormat.FULL, allLocale[localeIndex]);
50         dateFormat.setTimeZone(timeZone);
51         return dateFormat.format(calendar.getTime());
52     }
53 }
```

Create `DisplayTimeForm.jsp` (Listing 38.14). This page displays a form just like the one shown in Figure 37.18a. Line 2 imports the `TimeBean` class. A bean is created in lines 3–5 and is used in lines 17, 19, 24, and 26 to return all locales and time zones. The scope of the bean is application (line 4), so the bean can be shared by all sessions of the application.

LISTING 38.14 `DisplayTimeForm.jsp`

```

1  <!-- DisplayTimeForm.jsp -->
2  <%@ page import = "chapter38.TimeBean" %>
3  <jsp:useBean id = "timeBeanId"
4    class = "chapter38.TimeBean" scope = "application" >
5  </jsp:useBean>
6
7  <html>
8    <head>
9      <title>
10       Display Time Form
11     </title>
12   </head>
13   <body>
14     <h3>Choose locale and time zone</h3>
15     <form method = "post" action = "DisplayTime.jsp">
16       Locale <select size = "1" name = "localeIndex">
17         <% for (int i = 0; i < timeBeanId.getAllLocale().length; i++) {%>
18           <option value = "<%= i %>">
19             <%= timeBeanId.getAllLocale()[i] %>
20           </option>
21         <%}%>
22       </select><br />
23       Time Zone <select size = "1" name = "timeZoneIndex">
24         <% for (int i = 0; i < timeBeanId.getAllTimeZone().length; i++) {%>
25           <option value = "<%= i %>">
26             <%= timeBeanId.getAllTimeZone()[i] %>
27           </option>
28         <%}%>
29       </select><br />
30       <input type = "submit" name = "Submit"
31         value = "Get Time" />
32       <input type = "reset" value = "Reset" />
33     </form>
34   </body>
35 </html>

```

Create `DisplayTime.jsp` (Listing 38.15). This page is invoked from `DisplayTimeForm.jsp` to display the time with the specified locale and time zone, just as in Figure 37.18b.

LISTING 38.15 `DisplayTime.jsp`

```

1  <!-- DisplayTime.jsp -->
2  <%@ page pageEncoding = "GB18030"%>
3  <%@ page import = "chapter38.TimeBean" %>
4  <jsp:useBean id = "timeBeanId"
5    class = "chapter38.TimeBean" scope = "application" >
6  </jsp:useBean>
7  <jsp:setProperty name = "timeBeanId" property = "*" />
8
9  <html>
10   <head>

```

```

11     <title>
12         Display Time
13     </title>
14 </head>
15 <body>
16 <h3>Choose locale and time zone</h3>
17     Current time is
18     <%= timeBeanId.currentTimeString(timeBeanId.getLocaleIndex(),
19         timeBeanId.getTimeZoneIndex()) %>
20 </body>
21 <html>

```

Line 2 sets the character encoding for the page to GB18030 for displaying international characters. By default, it is UTF-8.

Line 5 imports `chapter38.TimeBean` and creates a bean using the same id as in the preceding page. Since the object is already created in the preceding page, the `timeBeanId` in this page (lines 4–6) and in the preceding page point to the same object.

38.9.4 Example: Registering Students

Listing 37.11, `RegistrationWithHttpSession.java`, gives a Java servlet that obtains student information from an HTML form (see Figure 37.21) and displays the information for user confirmation (see Figure 37.22). Once the user confirms it, the servlet stores the data into the database. This section rewrites the servlet using JSP. You will create two JSP pages, one named `GetRegistrationData.jsp` for displaying the data for user confirmation and the other named `StoreData.jsp` for storing the data into the database.

Since every session needs to connect to the same database, you should declare a class for connecting to the database and for storing a student to the database. This class named `StoreData` is given in Listing 38.16. The `initializeJdbc` method (lines 15–31) connects to the database and creates a prepared statement for storing a record to the Address table. The `storeStudent` method (lines 34–45) executes the prepared statement to store a student address. The `Address` class is created in Listing 37.12.

LISTING 38.16 StoreData.java

```

1  package chapter38;
2
3  import java.sql.*;
4  import chapter37.Address;
5
6  public class StoreData {
7      // Use a prepared statement to store a student into the database
8      private PreparedStatement pstmt;
9
10     public StoreData() {
11         initializeJdbc();
12     }
13
14     /** Initialize database connection */
15     private void initializeJdbc() {
16         try {
17             Class.forName("com.mysql.jdbc.Driver");
18
19             // Connect to the sample database
20             Connection connection = DriverManager.getConnection
21                 ("jdbc:mysql://localhost/javabook" , "scott", "tiger");
22

```

```

23         // Create a Statement
24         pstmt = connection.prepareStatement("insert into Address " +
25             "(lastName, firstName, mi, telephone, email, street, city, "
26             + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
27     }
28     catch (Exception ex) {
29         System.out.println(ex);
30     }
31 }
32
33 /** Store a student record to the database */
34 public void storeStudent(Address address) throws SQLException {
35     pstmt.setString(1, address.getLastName());
36     pstmt.setString(2, address.getFirstName());
37     pstmt.setString(3, address.getMi());
38     pstmt.setString(4, address.getTelephone());
39     pstmt.setString(5, address.getEmail());
40     pstmt.setString(6, address.getStreet());
41     pstmt.setString(7, address.getCity());
42     pstmt.setString(8, address.getState());
43     pstmt.setString(9, address.getZip());
44     pstmt.executeUpdate();
45 }
46 }

```

The HTML file that displays the form is identical to `Registration.html` in Listing 37.8 except that the action is replaced by `HGetRegistrationData.jsp`.

`GetRegistrationData.jsp`, which obtains the data from the form, is shown in Listing 38.17. A bean is created in lines 3–4. Line 5 obtains the property values from the form. This is a shorthand notation. Note the parameter names and the property names must be the same to use this notation.

LISTING 38.17 GetRegistrationData.jsp

```

1  <!-- GetRegistrationData.jsp -->
2  <%@ page import = "chapter37.Address" %>
3  <jsp:useBean id = "addressId"
4      class = "chapter37.Address" scope = "session"></jsp:useBean>
5  <jsp:setProperty name = "addressId" property = "*" />
6
7  <html>
8      <body>
9          <h1>Registration Using JSP</h1>
10
11      <%
12          if (addressId.getLastName() == null ||
13              addressId.getFirstName() == null) {
14              out.println("Last Name and First Name are required");
15              return; // End the method
16          }
17      %>
18
19      <p>You entered the following data</p>
20      <p>Last name: <%= addressId.getLastName() %></p>
21      <p>First name: <%= addressId.getFirstName() %></p>
22      <p>MI: <%= addressId.getMi() %></p>
23      <p>Telephone: <%= addressId.getTelephone() %></p>

```



```

24     <p>Email: <%= addressId.getEmail() %></p>
25     <p>Address: <%= addressId.getStreet() %></p>
26     <p>City: <%= addressId.getCity() %></p>
27     <p>State: <%= addressId.getState() %></p>
28     <p>Zip: <%= addressId.getZip() %></p>
29
30     <!-- Set the action for processing the answers -->
31     <form method = "post" action = "StoreStudent.jsp">
32         <input type = "submit" value = "Confirm">
33     </form>
34 </body>
35 </html>

```

GetRegistrationData.jsp invokes **StoreStudent.jsp** (line 31) when the user clicks the *Confirm* button. In Listing 38.18, the same **addressId** is shared with the preceding page within the scope of the same session in lines 3–4. A bean for **StoreData** is created in lines 5–6 with the scope of application.

LISTING 38.18 StoreStudent.jsp

```

1  <!-- StoreStudent.jsp -->
2  <%@ page import = "chapter37.Address" %>
3  <jsp:useBean id = "addressId" class = "chapter37.Address"
4      scope = "session"></jsp:useBean>
5  <jsp:useBean id = "storeDataId" class = "chapter38.StoreData"
6      scope = "application"></jsp:useBean>
7
8  <html>
9      <body>
10         <%
11             storeDataId.storeStudent(addressId);
12
13             out.println(addressId.getFirstName() + " " +
14                 addressId.getLastName() +
15                 " is now registered in the database");
16             out.close(); // Close stream
17         %>
18     </body>
19 </html>

```



Note

The scope for **addressId** is *session*, but the scope for **storeDataId** is *application*. Why? **GetRegistrationData.jsp** obtains student information, and **StoreData.jsp** stores the information in the same session. So the *session* scope is appropriate for **addressId**. All the sessions access the same database and use the same prepared statement to store data. With the *application* scope for **storeDataId**, the bean for **StoreData** needs to be created just once.



Note

The **storeStudent** method in line 11 may throw a **java.sql.SQLException**. In JSP, you can omit the try-block for checked exceptions. In case of an exception, JSP displays an error page.



Tip
Using beans is an effective way to develop JSP. You should put Java code into a bean as much as you can. The bean not only simplifies JSP programming but also makes code reusable. The bean can also be used to implement persistent sessions.



38.10 Forwarding Requests from JavaServer Pages

You can use the JSP forward tag to jump to navigate to another HTML page.

Web applications developed using JSP generally consist of many pages linked together. JSP provides a forwarding tag in the following syntax that can be used to forward a page to another page:

```
<jsp:forward page = "destination" />
```



- 38.10.1** How do you associate bean properties with input parameters?
- 38.10.2** How do you write a statement to forward requests to another JSP page?

38.11 Case Study: Browsing Database Tables

This section presents a very useful JSP application for browsing tables. When you start the application, the first page prompts the user to enter the JDBC driver, URL, username, and password for a database, as shown in Figure 38.9. After you log in to the database, you can select a table to browse, as shown in Figure 38.10. Clicking the *Browse Table Content* button displays the table content, as shown in Figure 38.11.

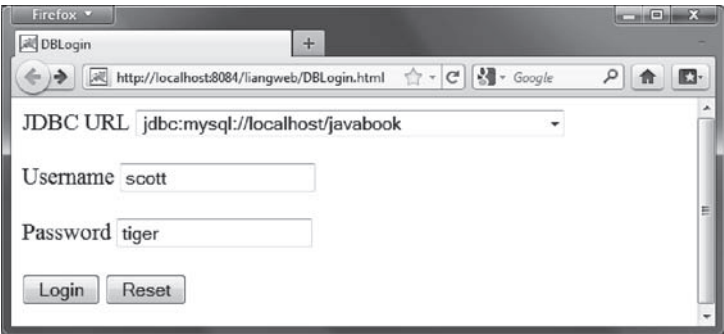


FIGURE 38.9 To access a database, you need to provide the JDBC driver, URL, username, and password.

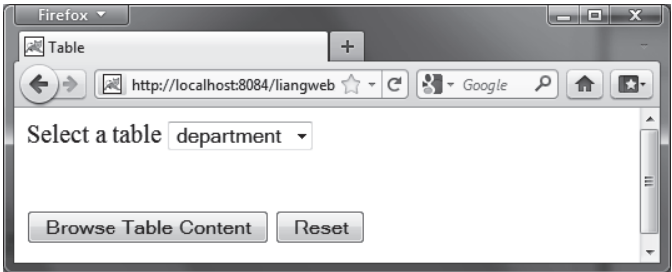
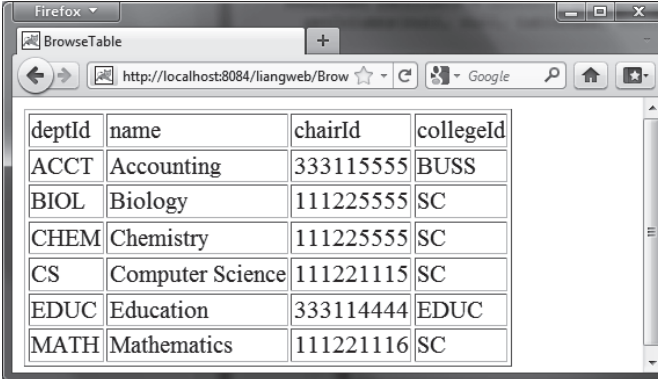


FIGURE 38.10 You can select a table to browse from this page.



deptId	name	chairId	collegeId
ACCT	Accounting	333115555	BUSS
BIOL	Biology	111225555	SC
CHEM	Chemistry	111225555	SC
CS	Computer Science	111221115	SC
EDUC	Education	333114444	EDUC
MATH	Mathematics	111221116	SC

FIGURE 38.11 The contents of the selected table are displayed.

Create a JavaBeans component named **DBBean.java** (see Listing 38.19).

LISTING 38.19 DBBean.java

```

1 package chapter38;
2
3 import java.sql.*;
4
5 public class DBBean {
6     private Connection connection = null;
7     private String username;
8     private String password;
9     private String driver;
10    private String url;
11
12    /** Initialize database connection */
13    public void initializeJdbc() {
14        try {
15            System.out.println("Driver is " + driver);
16            Class.forName(driver);
17
18            // Connect to the sample database
19            connection = DriverManager.getConnection(url, username,
20                password);
21        }
22        catch (Exception ex) {
23            ex.printStackTrace();
24        }
25    }
26
27    /** Get tables in the database */
28    public String[] getTables() {
29        String[] tables = null;
30
31        try {
32            DatabaseMetaData dbMetaData = connection.getMetaData();
33            ResultSet rsTables = dbMetaData.getTables(null, null, null,
34                new String[] { "TABLE" });
35
36            int size = 0;

```

```

37         while (rsTables.next()) size++;
38
39         rsTables = dbMetaData.getTables(null, null, null,
40             new String[] {"TABLE"});
41
42         tables = new String[size];
43         int i = 0;
44         while (rsTables.next())
45             tables[i++] = rsTables.getString("TABLE_NAME");
46     }
47     catch (Exception ex) {
48         ex.printStackTrace();
49     }
50
51     return tables;
52 }
53
54 /** Return connection property */
55 public Connection getConnection() {
56     return connection;
57 }
58
59 public void setUsername(String newUsername) {
60     username = newUsername;
61 }
62
63 public String getUsername() {
64     return username;
65 }
66
67 public void setPassword(String newPassword) {
68     password = newPassword;
69 }
70
71 public String getPassword() {
72     return password;
73 }
74
75 public void setDriver(String newDriver) {
76     driver = newDriver;
77 }
78
79 public String getDriver() {
80     return driver;
81 }
82
83 public void setUrl(String newUrl) {
84     url = newUrl;
85 }
86
87 public String getUrl() {
88     return url;
89 }
90 }

```

Create an HTML file named **DBLogin.html** (see Listing 38.20) that prompts the user to enter database information and three JSP files named **DBLoginInitialization.jsp** (see Listing 38.21), **Table.jsp** (see Listing 38.22), and **BrowseTable.jsp** (see Listing 38.23) to process and obtain database information.

LISTING 38.20 DBLogin.html

```

1  <!-- DBLogin.html -->
2  <html>
3    <head>
4      <title>
5        DBLogin
6      </title>
7    </head>
8    <body>
9      <form method = "post" action = "DBLoginInitialization.jsp">
10     JDBC URL
11     <select name = "url" size = "1">
12       <option>jdbc:odbc:ExampleMDBDataSource</option>
13       <option>jdbc:mysql://localhost/javabook</option>
14       <option>jdbc:oracle:thin:@liang.armstrong.edu:1521:orcl</option>
15     </select><br /><br />
16     Username <input name = "username" /><br /><br />
17     Password <input name = "password" /><br /><br />
18     <input type = "submit" name = "Submit" value = "Login" />
19     <input type = "reset" value = "Reset" />
20   </form>
21 </body>
22 </html>

```

LISTING 38.21 DBLoginInitialization.jsp

```

1  <!-- DBLoginInitialization.jsp -->
2  <%@ page import = "chapter38.DBBean" %>
3  <jsp:useBean id = "dBBeanId" scope = "session"
4    class = "chapter38.DBBean">
5  </jsp:useBean>
6  <jsp:setProperty name = "dBBeanId" property = "*" />
7  <html>
8    <head>
9      <title>DBLoginInitialization</title>
10   </head>
11   <body>
12
13     <!-- Connect to the database --%>
14     <% dBBeanId.initializeJdbc(); %>
15
16     <% if (dBBeanId.getConnection() == null) { %>
17       Error: Login failed. Try again.
18     <% }
19     else {%>
20       <jsp:forward page = "Table.jsp" />
21     <% } %>
22   </body>
23 </html>

```

LISTING 38.22 Table.jsp

```

1  <!-- Table.jsp -->
2  <%@ page import = "chapter38.DBBean" %>
3  <jsp:useBean id = "dBBeanId" scope = "session"
4    class = "chapter38.DBBean">
5  </jsp:useBean>
6  <html>
7    <head>

```

```

8      <title>Table</title>
9  </head>
10 <body>
11 <% String[] tables = dBBeanId.getTables();
12     if (tables == null) { %>
13         No tables
14     } %>
15     else { %>
16         <form method = "post" action = "BrowseTable.jsp">
17             Select a table
18             <select name = "tablename" size = "1">
19                 <% for (int i = 0; i < tables.length; i++) { %>
20                     <option><%= tables[i] %></option>
21                 <% }
22             } %>
23             </select><br /><br /><br />
24             <input type = "submit" name = "Submit"
25                 value = "Browse Table Content">
26             <input type = "reset" value = "Reset">
27         </form>
28     </body>
29 </html>

```

LISTING 38.23 BrowseTable.jsp

```

1  <!-- BrowseTable.jsp -->
2  <%@ page import = "chapter38.DBBean" %>
3  <jsp:useBean id = "dBBeanId" scope = "session"
4      class = "chapter38.DBBean" >
5  </jsp:useBean>
6  <%@ page import = "java.sql.*" %>
7  <html>
8      <head>
9          <title>BrowseTable</title>
10     </head>
11     <body>
12
13     <% String tableName = request.getParameter("tablename");
14
15         ResultSet rsColumns = dBBeanId.getConnection().getMetaData().
16             getColumns(null, null, tableName, null);
17     %>
18     <table border = "1">
19         <tr>
20             <% // Add column names to the table
21             while (rsColumns.next()) { %>
22                 <td><%= rsColumns.getString("COLUMN_NAME") %></td>
23             <%}%>
24         </tr>
25
26         <% Statement statement =
27             dBBeanId.getConnection().createStatement();
28             ResultSet rs = statement.executeQuery(
29                 "select * from " + tableName);
30
31             // Get column count
32             int columnCount = rs.getMetaData().getColumnCount();
33
34             // Store rows to rowData
35             while (rs.next()) {

```

```

36         out.println("<tr>");
37         for (int i = 0; i < columnCount; i++) { %>
38             <td><%= rs.getObject(i + 1) %></td>
39         }
40         out.println("</tr>");
41     } %>
42 </table>
43 </body>
44 </html>

```

You start the application from DBLogin.html. This page prompts the user to enter a JDBC driver, URL, username, and password to log in to a database. A list of accessible drivers and URLs is provided in the selection list. You must make sure that these database drivers are added into the Libraries node in the project.

When you click the *Login* button, DBLoginInitialization.jsp is invoked. When this page is processed for the first time, an instance of **DBBean** named **dbBeanId** is created. The input parameters **driver**, **url**, **username**, and **password** are passed to the bean properties. The **initializeJdbc** method loads the driver and establishes a connection to the database. If login fails, the **connection** property is **null**. In this case, an error message is displayed. If login succeeds, control is forwarded to Table.jsp.

Table.jsp shares **dbBeanId** with DBLoginInitialization.jsp in the same session, so it can access **connection** through **dbBeanId** and obtain tables in the database using the database metadata. The table names are displayed in a selection box in a form. When the user selects a table name and clicks the *Browse Table Content* button, BrowseTable.jsp is processed.

BrowseTable.jsp shares **dbBeanId** with Table.jsp and DBLoginInitialization.jsp in the same session. It retrieves the table contents for the selected table from Table.jsp.

JSP Scripting Constructs Syntax

- **<%= Java expression %>** The expression is evaluated and inserted into the page.
- **<% Java statement %>** Java statements inserted in the **jspService** method.
- **<%! Java declaration %>** Defines data fields and methods.
- **<!-- JSP comment %>** The JSP comments do not appear in the resultant HTML file.
- **<%@ directive attribute="value" %>** The JSP directives give the JSP engine information about the JSP page. For example, **<%@ page import="java.util.*", java.text.*" %>** imports **java.util.*** and **java.text.***.
- **<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" />** Creates a bean if new. If a bean is already created, associates the id with the bean in the same scope.
- **<jsp:useBean id="objectName" scope="scopeAttribute" class="ClassName" > statements </jsp:useBean>** The statements are executed when the bean is created. If a bean with the same id and class name already exists, the statements are not executed.
- **<jsp:getProperty name="beanId" property="sample" />** Gets the property value from the bean, which is the same as **<%= beanId.getSample() %>**.
- **<jsp:setProperty name="beanId" property="sample" value="test1" />** Sets the property value for the bean, which is the same as **<% beanId.setSample("test1"); %>**.
- **<jsp:setProperty name="beanId" property="score" param="score" />** Sets the property with an input parameter.

- `<jsp:setProperty name="beanId" property="*" />` Associates and sets all the bean properties in `beanId` with the input parameters that match the property names.
- `<jsp:forward page="destination" />` Forwards this page to a new page.

JSP Predefined Variables

- `application` represents the `ServletContext` object for storing persistent data for all clients.
- `config` represents the `ServletConfig` object for the page.
- `out` represents the character output stream, which is an instance of `PrintWriter`, obtained from `response.getWriter()`.
- `page` is alternative to `this`.
- `request` represents the client's request, which is an instance of `HttpServletRequest` in the servlet's `service` method.
- `response` represents the client's response, which is an instance of `HttpServletResponse` in the servlet's `service` method.
- `session` represents the `HttpSession` object associated with the request, obtained from `request.getSession()`.

CHAPTER SUMMARY

1. A JavaServer page is like a regular HTML page with special tags, known as *JSP tags*, which enable the Web server to generate dynamic content. You can create a webpage with static HTML and enclose the code for generating dynamic content in the JSP tags.
2. A JSP page must be stored in a file with a `.jsp` extension. The Web server translates the `.jsp` file into a Java servlet, compiles the servlet, and executes it. The result of the execution is sent to the browser for display.
3. A JSP page is translated into a servlet when the page is requested for the first time. It is not retranslated if the page is not modified. To ensure that the first-time real user does not encounter a delay, JSP developers should test the page after it is installed.
4. There are three main types of JSP constructs: scripting constructs, directives, and actions. *Scripting* elements enable you to specify Java code that will become part of the resultant servlet. *Directives* enable you to control the overall structure of the resultant servlet. *Actions* enable you to control the behaviors of the JSP engine.
5. Three types of scripting constructs can be used to insert Java code into the resultant servlet: expressions, scriptlets, and declarations.
6. The scope attribute (application, session, page, and request) specifies the scope of a JavaBeans object. Application specifies that the object be bound to the application. Session specifies that the object be bound to the client's session. Page is the default scope, which specifies that the object be bound to the page. Request specifies that the object be bound to the client's request.
7. Web applications developed using JSP generally consist of many pages linked together. JSP provides a forwarding tag in the following syntax that can be used to forward a page to another page: `<jsp:forward page="destination" />`.

Quiz

Answer the quiz for this chapter online at the book Companion Website.



PROGRAMMING EXERCISES

MyProgrammingLab™



Note

Solutions to even-numbered exercises in this chapter are in `exercise\jsp\exercise` from `evennumberedexercise.zip`, which can be downloaded from the Companion Website.

Section 38.4

38.1 (*Factorial table in JSP*) Rewrite Exercise 37.1 using JSP.

38.2 (*Multiplication table in JSP*) Rewrite Exercise 37.2 using JSP.

Section 38.5

***38.3** (*Obtain parameters in JSP*) Rewrite the servlet in Listing 37.4, `GetParameters.java`, using JSP. Create an HTML form that is identical to `Student_Registration_Form.html` in Listing 37.3 except that the action is replaced by `Exercise40_3.jsp` for obtaining parameter values.

Section 38.6

38.4 (*Calculate tax in JSP*) Rewrite Exercise 37.4 using JSP. You need to import `ComputeTax` in the JSP.

***38.5** (*Find scores from text files*) Rewrite Exercise 37.6 using servlets.

****38.6** (*Find scores from database tables*) Rewrite Exercise 37.7 using servlets.

Section 38.7

****38.7** (*Change the password*) Rewrite Exercise 37.8 using servlets.

Comprehensive

***38.8** (*Store cookies in JSP*) Rewrite Exercise 37.10 using JSP. Use `response.addCookie(Cookie)` to add a cookie.

***38.9** (*Retrieve cookies in JSP*) Rewrite Exercise 37.11 using JSP. Use `Cookie[] cookies = request.getCookies()` to get all cookies.

38.10 (*Draw images*) Write a JSP program that displays a country's flag and description as shown in Figure 38.12. The country code such as `us` is passed as a parameter in the URL. The country's flag file is named `CountryCode.gif` and the description is stored in a text file named `CountryCode.txt` on the server. So, for the country code `us`, the flag file `us.gif` and the text file is `us.txt`.

*****38.11** (*Syntax highlighting*) Rewrite Exercise 37.12 using JSP.

****38.12** (*Opinion poll*) Rewrite Exercise 37.13 using JSP.

*****38.13** (*Multiple-question opinion poll*) The `Po11` table in Exercise 37.13 contains only one question. Suppose you have a `Po11` table that contains multiple questions. Write a JSP that reads all the questions from the table and display them in a form, as shown in Figure 38.13a. When the user clicks the *Submit* button, another JSP page is invoked. This page updates the Yes or No counts for each question and displays the current Yes and No counts for each question in the `Po11` table, as shown in Figure 38.13b. Note that the table may contain many questions. The questions in the figure are just examples. Sort the questions in alphabetical order.

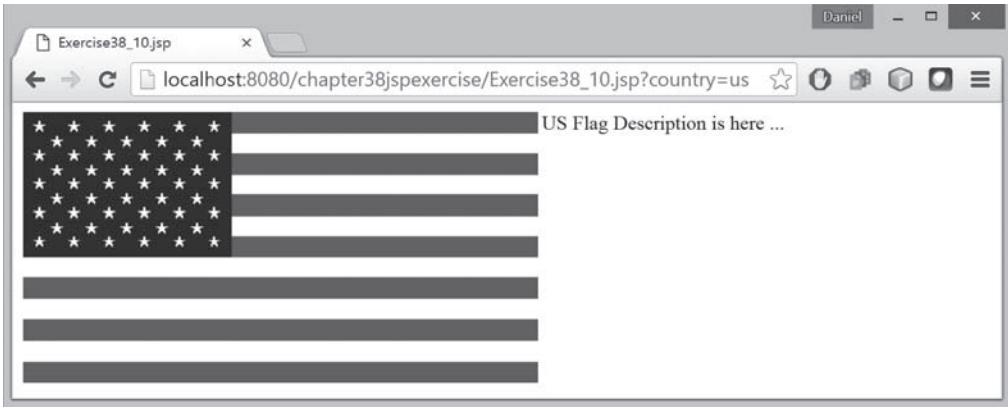


FIGURE 38.12 The program displays an image and the description of the image.

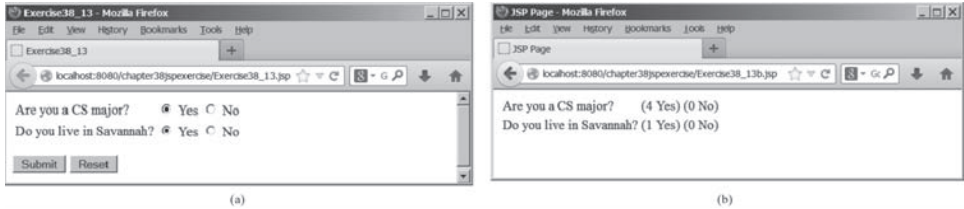


FIGURE 38.13 The form prompts the user to enter Yes or No for each question in (a), and the updated Yes or No counts are displayed in (b).

****38.14** (*Addition quiz*) Write a JSP program that generates addition quizzes randomly, as shown in Figure 38.14a. After the user answers all questions, the JSP displays the result, as shown in Figure 38.14b.

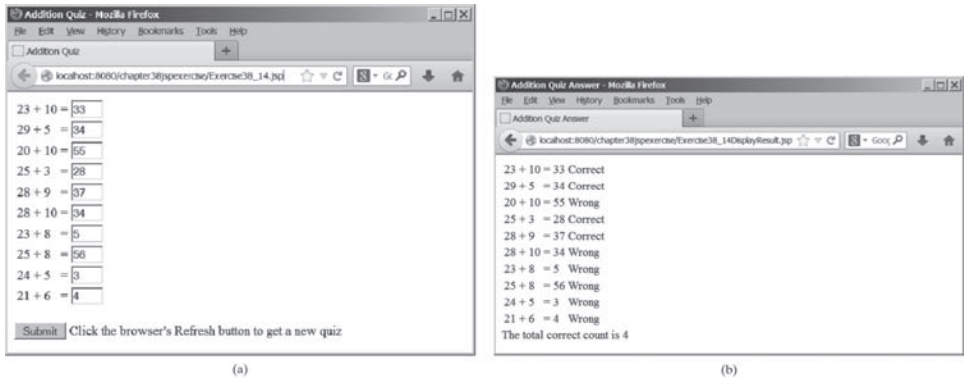


FIGURE 38.14 The program displays addition questions in (a) and answers in (b).

****38.15** (*Subtraction quiz*) Write a JSP program that generates subtraction quizzes randomly, as shown in Figure 38.14a. The first number must always be greater than or equal to the second number. After the user answers all questions, the JSP displays the result, as shown in Figure 38.14b.

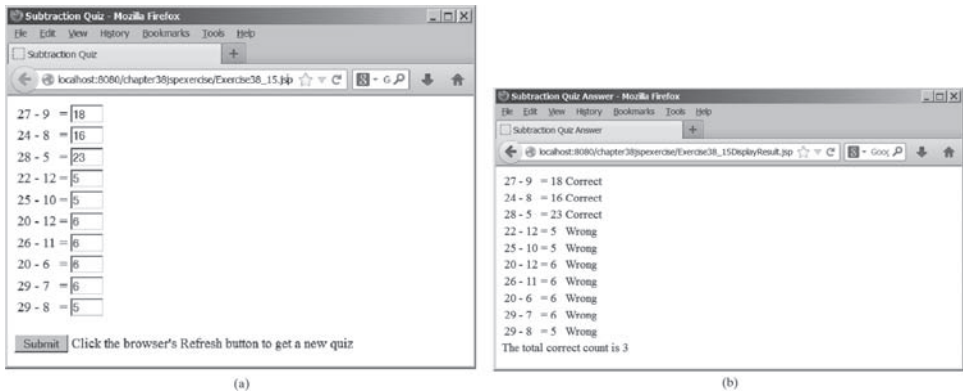


FIGURE 38.14 The program displays subtraction questions in (a) and answers in (b).

****38.16** (*Guess birthday*) Listing 3.3, `GuessBirthDay.java`, gives a program for guessing a birthday. Write a JSP program that displays five sets of numbers, as shown in Figure 38.15a. After the user checks the appropriate boxes and clicks the *Find Date* button, the program displays the date, as shown in Figure 38.15b.

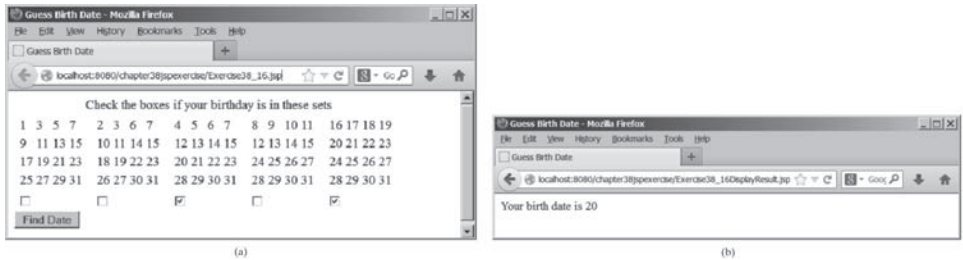


FIGURE 38.15 (a) The program displays five sets of numbers for the user to check the boxes. (b) The program displays the date.

****38.17** (*Guess capitals*) Write a JSP that prompts the user to enter a capital for a state, as shown in Figure 38.16a. Upon receiving the user input, the program reports whether the answer is correct, as shown in Figure 38.16b. You can click the *Next* button to display another question. You can use a two-dimensional array to store the states and capitals, as proposed in Exercise 9.22. Create a list from the array and apply the `shuffle` method to reorder the list so the questions will appear in random order.

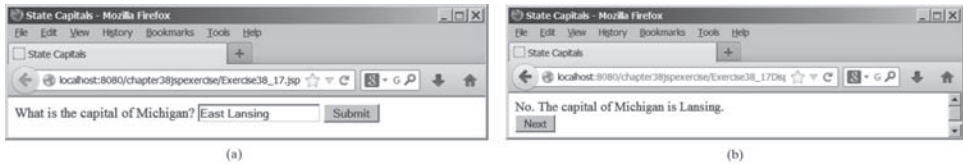


FIGURE 38.16 (a) The program displays a question. (b) The program displays the answer to the question.

***38.18** (*Large factorial*) Rewrite Listing 38.11 to handle a large factorial. Use the `BigInteger` class introduced in §14.12.

****38.19** (*Access and update a **Staff** table*) Write a JSP for Exercise 33.1, as shown in Figure 38.17.

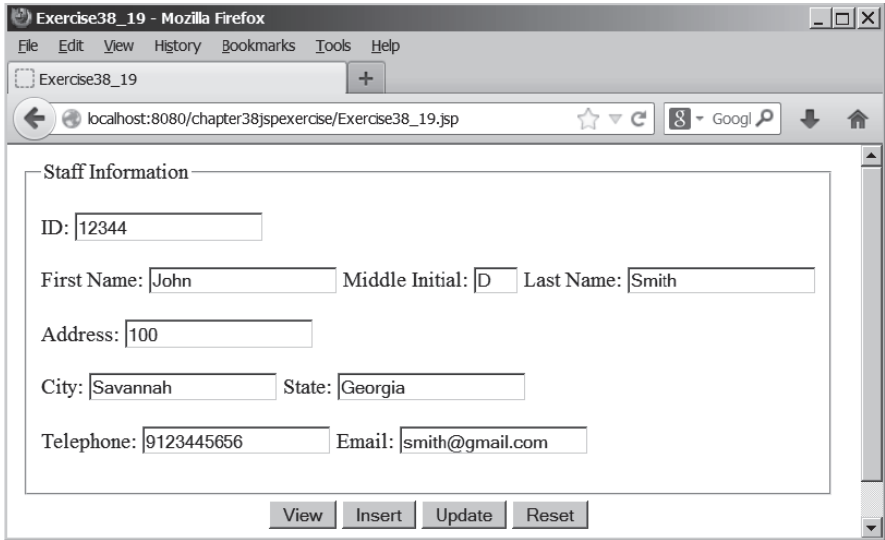


FIGURE 38.17 The JSP page lets you view, insert, and update staff information.

***38.20** (*Guess number*) Write a JSP page that generates a random number between **1** and **1000** and let the user enter a guess. When the user enters a guess, the program should tell the user whether the guess is correct, too high, or too low.