CHAPTER

39

JavaServer Faces

Objectives

- To explain what JSF is (§39.1).
- To create a JSF project in NetBeans (§39.2.1).
- To create a JSF page (§39.2.2).
- To create a JSF managed bean (§39.2.3).
- To use JSF expressions in a facelet (§39.2.4).
- To use JSF GUI components (§39.3).
- To obtain and process input from a form (§39.4).
- To develop a calculator using JSF (§39.5).
- To track sessions in application, session, view, and request scopes (§39.6).
- To validate input using the JSF validators (§39.7).
- To bind database with facelets (§39.8).
- To open a new JSF page from the current page (§39.9).
- To program using contexts and dependency injection (§39.10).

39.1 Introduction

JavaServer Faces (JSF) is a new technology for developing server-side Web applications using Java.

JSF

JSF enables you to completely separate Java code from HTML. You can quickly build Web applications by assembling reusable UI components in a page, connecting these components to Java programs and wiring client-generated events to server-side event handlers. The application developed using JSF is easy to debug and maintain.

Note

JSF 2
XHTML
CSS

This chapter introduces JSF 2, the latest standard for JavaServer Faces. You need to know XHTML (eXtensible HyperText Markup Language) and CSS (Cascading Style Sheet) to start this chapter. For information on XHTML and CSS, see Supplements V.A and V.B on the Companion Website.

Caution

NetBeans 7.3.1
GlassFish 4
Java EE 7

The examples and exercises in this chapter were tested using NetBeans 7.3.1, GlassFish 4, and Java EE 7. You need to use NetBeans 7.3.1 or a higher version with GlassFish 4 and Java EE 7 to develop your JSF projects.

39.2 Getting Started with JSF

NetBeans is an effective tool for developing JSF applications.

We begin with a simple example that illustrates the basics of developing JSF projects using NetBeans. The example is to display the date and time on the server, as shown in Figure 39.1.

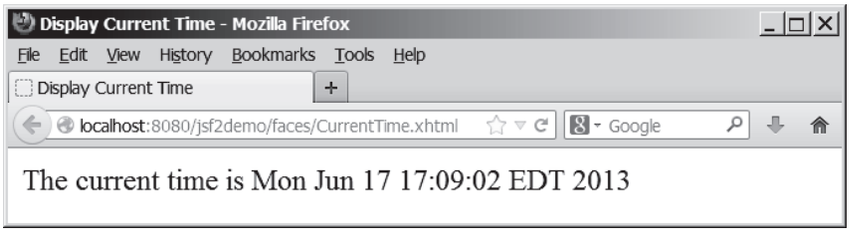


FIGURE 39.1 The application displays the date and time on the server.

39.2.1 Creating a JSF Project

Here are the steps to create the application.

create a project

Step 1: Choose ***File*, *New Project*** to display the New Project dialog box. In this box, choose *Java Web* in the Categories pane and *Web Application* in the Projects pane. Click *Next* to display the New Web Application dialog box.

In the New Web Application dialog box, enter and select the following fields, as shown in Figure 39.2a:

Project Name: `jsf2demo`
Project Location: `c:\book`

choose server and Java EE 7

Step 2: Click *Next* to display the dialog box for choosing servers and settings. Select the following fields as shown in Figure 39.2b. (Note: You can use any server such as GlassFish 4.x that supports Java EE 6.)

Server: `GlassFish 4`
Java EE Version: `Java EE 7 Web`

Step 3: Click *Next* to display the dialog box for choosing frameworks, as shown in Figure 39.3. Check *JavaServer Faces* and JSF 2.0 as Server Library. Click *Finish* to create the project, as shown in Figure 39.4.

choose JavaServer Faces and JSF2.2

(a) (b)

FIGURE 39.2 The New Web Application dialog box enables you to create a new Web project.

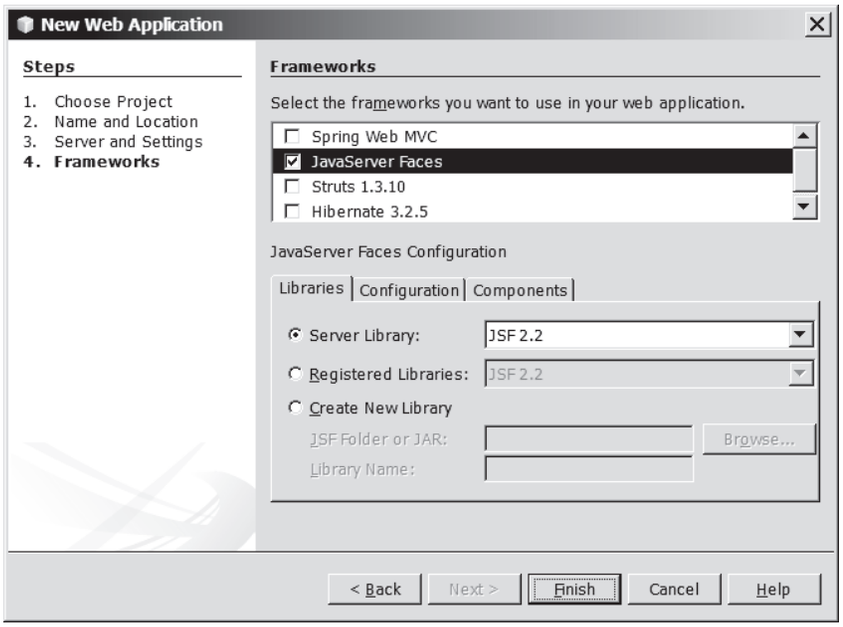


FIGURE 39.3 Check JavaServer Faces and JSF 2.2 to create a Web project.

39.2.2 A Basic JSF Page

A new project was just created with a default page named index.xhtml, as shown in Figure 39.4. This page is known as a *facelet*, which mixes JSF tags with XHTML tags. Listing 39.1 lists the contents of index.xhtml.

facelet

LISTING 39.1 index.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!-- index.xhtml -->
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5  <html xmlns="http://www.w3.org/1999/xhtml"
6        xmlns:h="http://xmlns.jcp.org/jsf/html">
```

xml version
comment
DOCTYPE

default namespace
JSF namespace

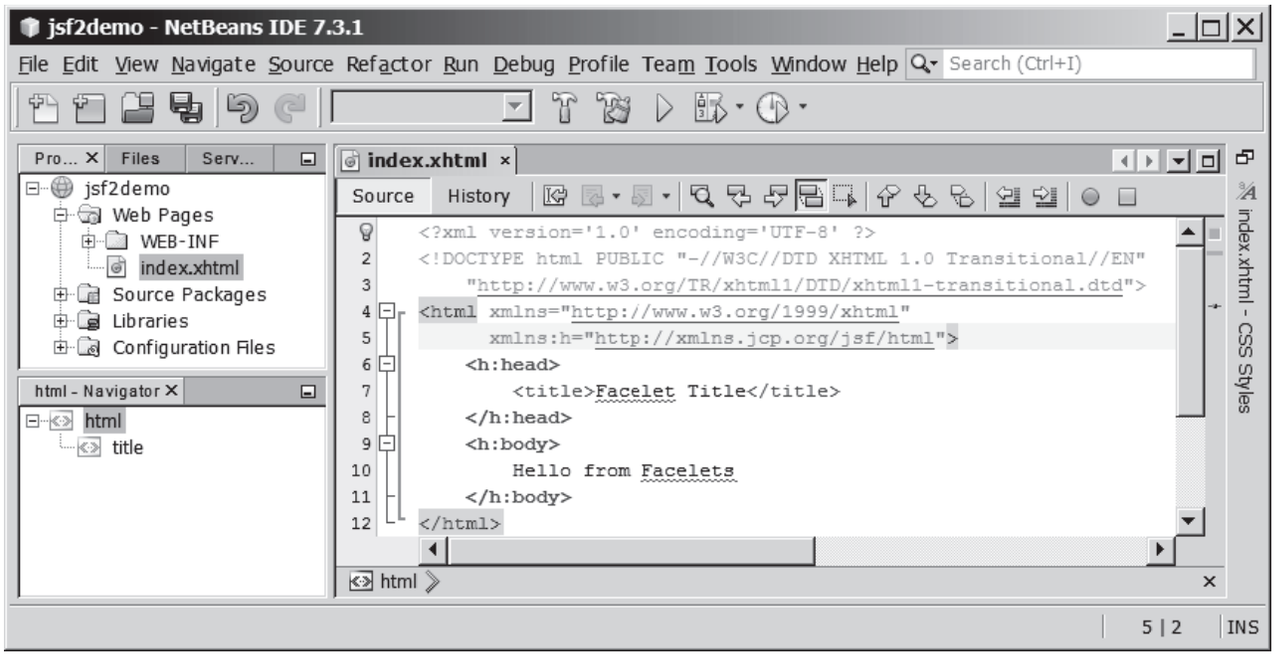


FIGURE 39.4 A default JSF page is created in a new Web project.

h:head
h:body

```
 7    <h:head>
 8      <title>Facelet Title</title>
 9    </h:head>
10    <h:body>
11      Hello from Facelets
12    </h:body>
13  </html>
```

XML declaration

Line 1 is an XML declaration to state that the document conforms to the XML version 1.0 and uses the UTF-8 encoding. The declaration is optional, but it is a good practice to use it. A document without the declaration may be assumed of a different version, which may lead to errors. If an XML declaration is present, it must be the first item to appear in the document. This is because an XML processor looks for the first line to obtain information about the document so that it can be processed correctly.

XML comment

Line 2 is a comment for documenting the contents in the file. XML comment always begins with `<!--` and ends with `-->`.

DOCTYPE

Lines 3 and 4 specify the version of XHTML used in the document. This can be used by the Web browser to validate the syntax of the document.

element
tag

An XML document consists of elements described by tags. An element is enclosed between a start tag and an end tag. XML elements are organized in a tree-like hierarchy. Elements may contain subelements, but there is only one root element in an XML document. All the elements must be enclosed inside the root tag. The root element in XHTML is defined using the `html` tag (line 5).

Each tag in XML must be used in a pair of the start tag and the end tag. A start tag begins with `<` followed by the tag name and ends with `>`. An end tag is the same as its start tag except that it begins with `</`. The start tag and end tag for `html` are `<html>` and `</html>`.

html tag

The `html` element is the root element that contains all other elements in an XHTML page. The starting `<html>` tag (lines 5 and 6) may contain one or more `xmlns` (XML namespace) attributes to specify the namespace for the elements used in the document. Namespaces are like Java packages. Java packages are used to organize classes and to avoid naming conflict. XHTML namespaces are used to organize tags and resolve naming conflict. If an element with the same name is defined in two namespaces, the fully qualified tag names can be used to differentiate them.

Each `xmlns` attribute has a name and a value separated by an equal sign (`=`). The following declaration (line 5) xmlns

```
xmlns="http://www.w3.org/1999/xhtml"
```

specifies that any unqualified tag names are defined in the default standard XHTML namespace.

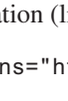
The following declaration (line 6)

```
xmlns:h="http://xmlns.jcp.org/jsf/html"
```

allows the tags defined in the JSF tag library to be used in the document. These tags must have a prefix `h`.

An `html` element contains a head and a body. The `h:head` element (lines 7–9) defines an HTML `title` element. The title is usually displayed in the browser window's title bar. h:head

An `h:body` element defines the page's content. In this simple example, it contains a string to be displayed in the Web browser. h:body



Note

The XML tag names are case sensitive, whereas HTML tags are not. So, `<html>` is different from `<HTML>` in XML. Every start tag in XML must have a matching end tag, whereas some tags in HTML do not need end tags.

You can now display the page in index.xhtml by right-clicking on index.xhtml in the projects pane and choose *Run File*. The page is displayed in a browser, as shown in Figure 39.5.

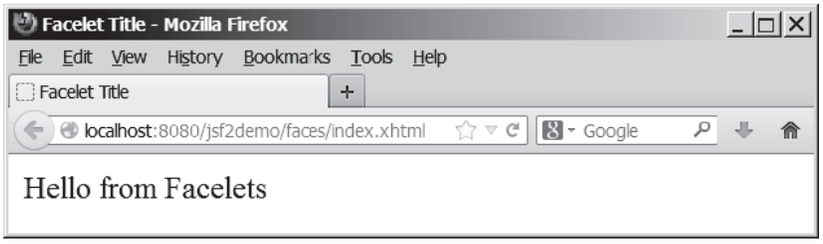


FIGURE 39.5 The index.xhtml is displayed in the browser.

Note

The JSF page is processed and converted into a regular HTML page for displaying by a browser. The Java software that runs on the server side for producing the HTML page is known as *Java server container* or simply *container*. The container is responsible for handling all server-side tasks for Java EE. GlassFish is a Java server container. container

39.2.3 Managed JavaBeans for JSF

JSF applications are developed using the Model-View-Controller (MVC) architecture, which separates the application's data (contained in the model) from the graphical presentation (the view). The controller is the JSF framework that is responsible for coordinating interactions between view and the model.

In JSF, the facelets are the view for presenting data. Data are obtained from Java objects. Objects are defined using Java classes. In JSF, the objects that are accessed from a facelet are JavaBeans objects. A *JavaBean* class is simply a public Java class with a no-arg constructor. JavaBeans may contain properties. By convention, a property is defined with a getter and a setter method. If a property only has a getter method, the property is called a read-only property. If a property only has a setter method, the property is called a write-only property. A property does not need to be defined as a data field in the class. JavaBean

Our example in this section is to develop a JSF facelet to display current time. We will create a JavaBean with a `getTime()` method that returns the current time as a string. The facelet will invoke this method to obtain current time.

Here are the steps to create a JavaBean named `TimeBean`.

Step 1. Right-click the project node jsf2demo to display a context menu as shown in Figure 39.6. Choose *New*, *JSF Managed Bean* to display the New JSF Managed Bean dialog box, as shown in Figure 39.7. (Note: if you don't see JSF Managed Bean in the menu, choose *Other* to locate it in the JavaServer Faces category.)

Step 2. Enter and select the following fields, as shown in Figure 39.7:

Class Name: `TimeBean`
Package: `jsf2demo`
Name: `timeBean`
Scope: `request`
Click *Finish* to create TimeBean.java, as shown in Figure 39.8.

Step 3. Add the `getTime()` method to return the current time, as shown in Listing 39.2.

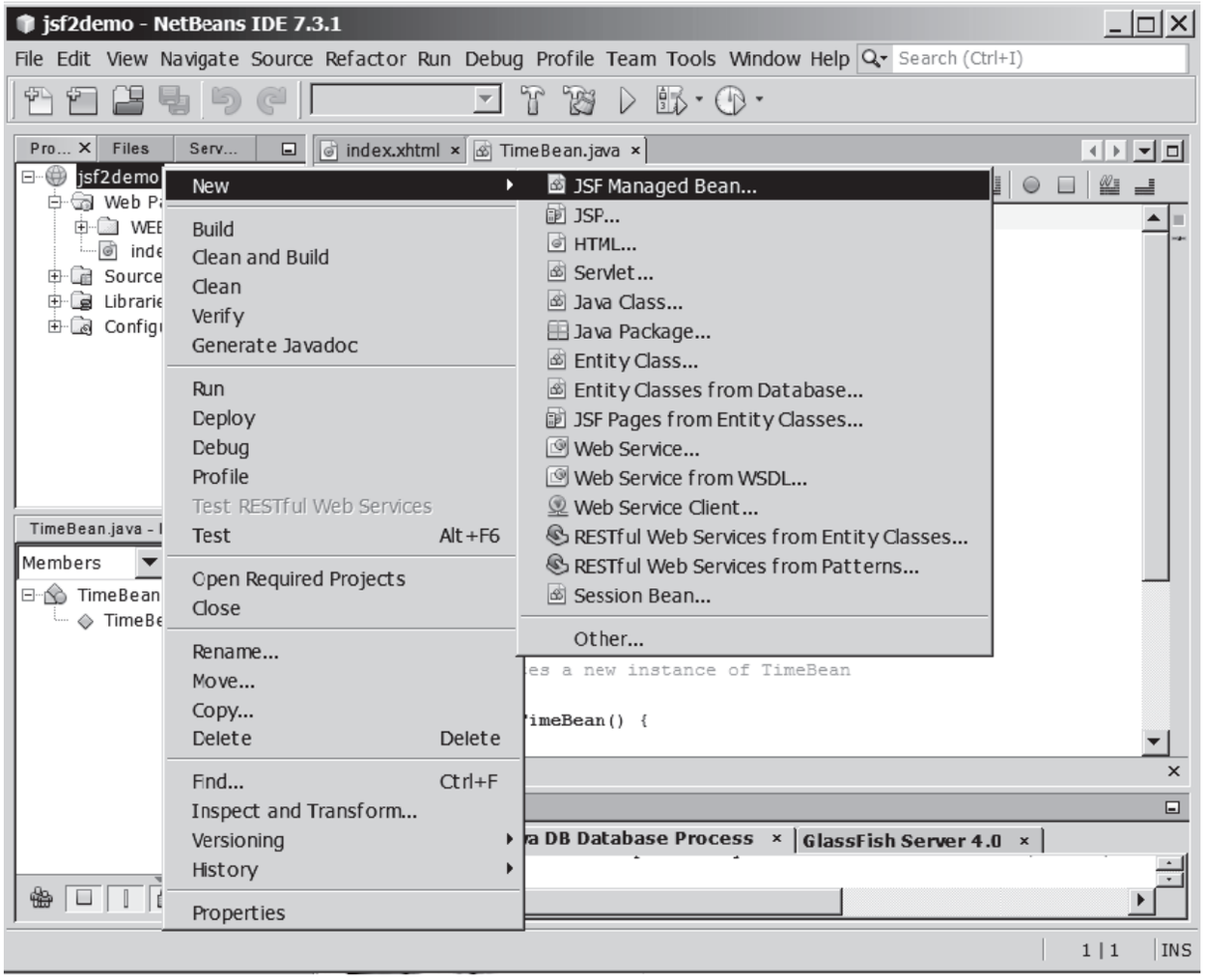


FIGURE 39.6 Choose JSF Managed Bean to create a JavaBean for JSF.

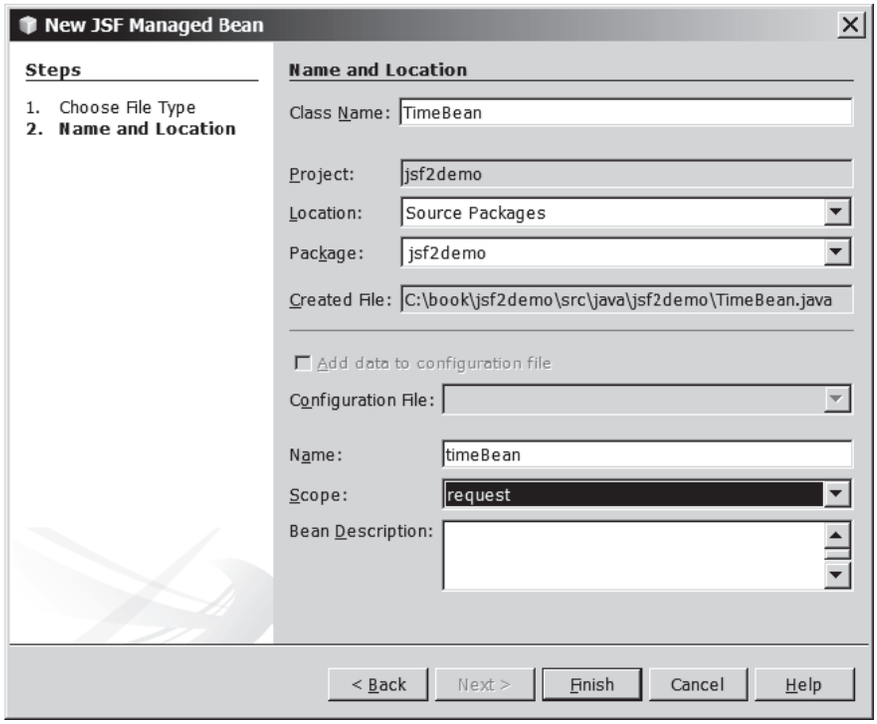


FIGURE 39.7 Specify the name, location, and scope for the bean.

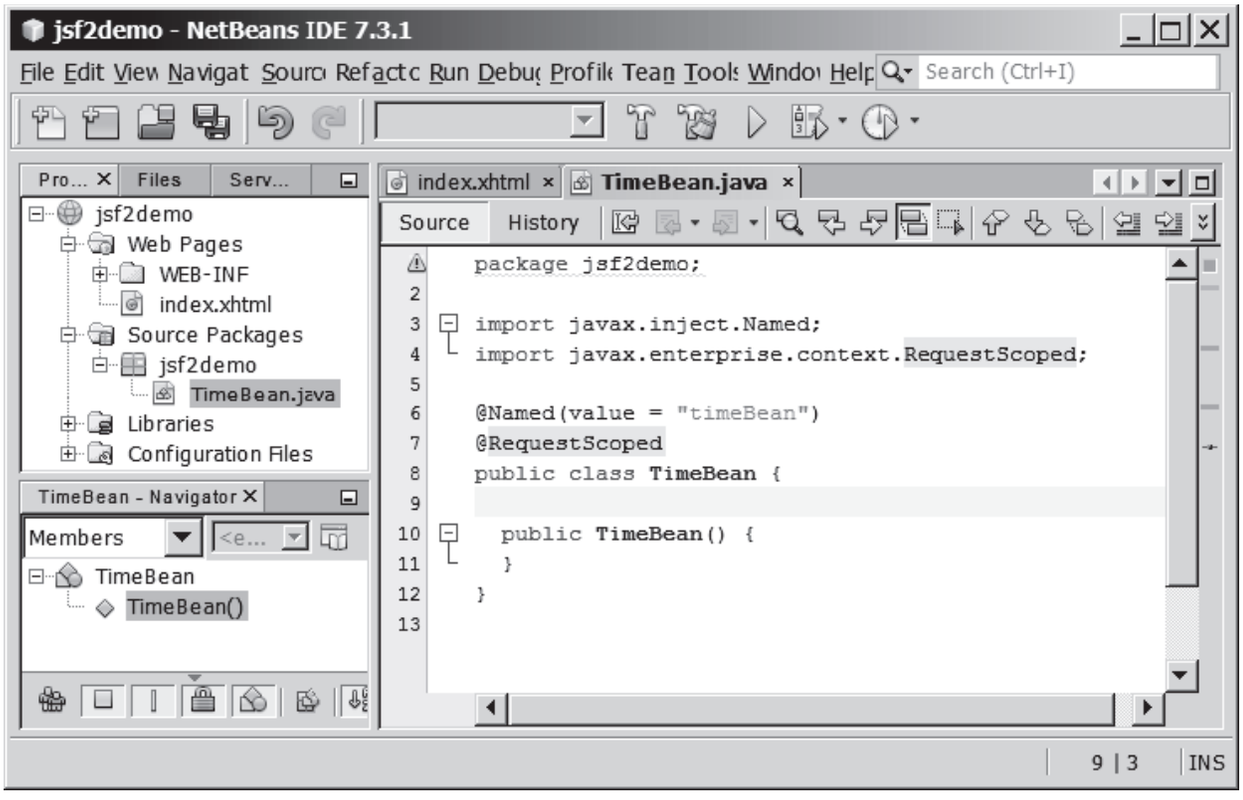


FIGURE 39.8 A JavaBean for JSF was created.

LISTING 39.2 TimeBean.java

```
1  package jsf2demo;
2
3  import javax.inject.Named;
4  import javax.enterprise.context.RequestScoped;
5
6  @Named (value = "timeBean")
7  @RequestScoped
8  public class TimeBean {
9    public TimeBean() {
10   }
11
12   public String getTime() {
13     return new java.util.Date().toString();
14   }
15 }
```

@Named
@RequestScoped
time property

`TimeBean` is a JavaBeans with the `@Named` annotation, which indicates that the JSF framework will create and manage the `TimeBean` objects used in the application. You have learned to use the `@Override` annotation in Chapter 11. The `@Override` annotation tells the compiler that the annotated method is required to override a method in a superclass. The `@Named` annotation tells the compiler to generate the code to enable the bean to be used by JSF facelets.

@RequestScope

The `@RequestScope` annotation specifies that the scope of the JavaBeans object is within a request. You can also use `@ViewScope`, `@SessionScope` or `@ApplicationScope` to specify the scope for a session or for the entire application.

39.2.4 JSF Expressions

We demonstrate JSF expressions by writing a simple application that displays the current time. You can display current time by invoking the `getTime()` method in a `TimeBean` object using a JSF expression.

To keep index.xhtml intact, we create a new JSF page named CurrentTime.xhtml as follows:

Step 1. Right-click the `jsf2demo` node in the project pane to display a context menu and choose *New*, *JSF Page* to display the New JSF File dialog box, as shown in Figure 39.9.

Step 2. Enter `CurrentTime` in the File Name field, choose Facelets and click *Finish* to generate CurrentTime.xhtml, as shown in Figure 39.10.

Step 3. Add a JSF expression to obtain the current time, as shown in Listing 39.3.

Step 4. Right-click on CurrentTime.xhtml in the project to display a context menu and choose *Run File* to display the page in a browser as shown in Figure 39.1.

LISTING 39.3 CurrentTime.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html">
6    <h:head>
7      <title>Display Current Time</title>
8      <meta http-equiv="refresh" content ="60" />
9    </h:head>
10   <h:body>
11     The current time is #{timeBean.time}
12   </h:body>
13 </html>
```

refresh page
JSF expression

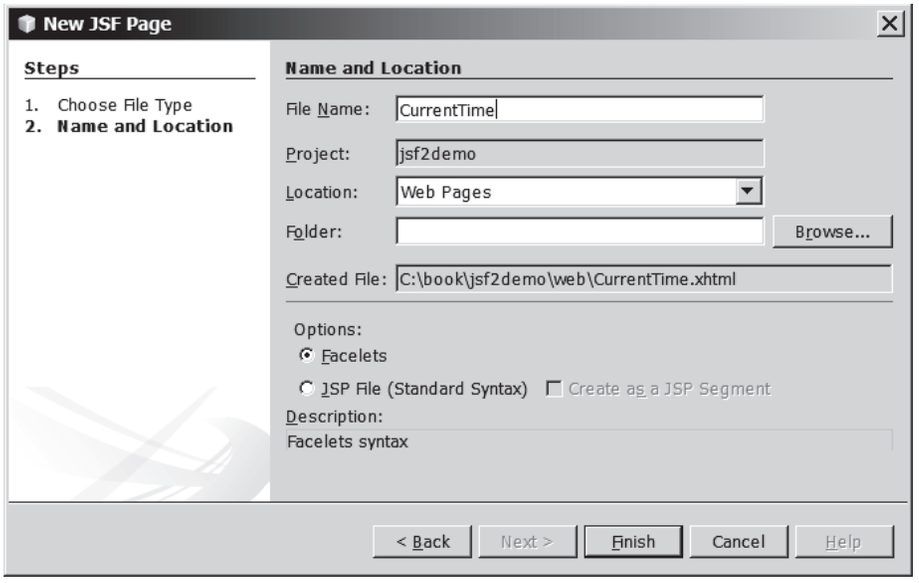


FIGURE 39.9 The New JSF Page dialog is used to create a JSF page.

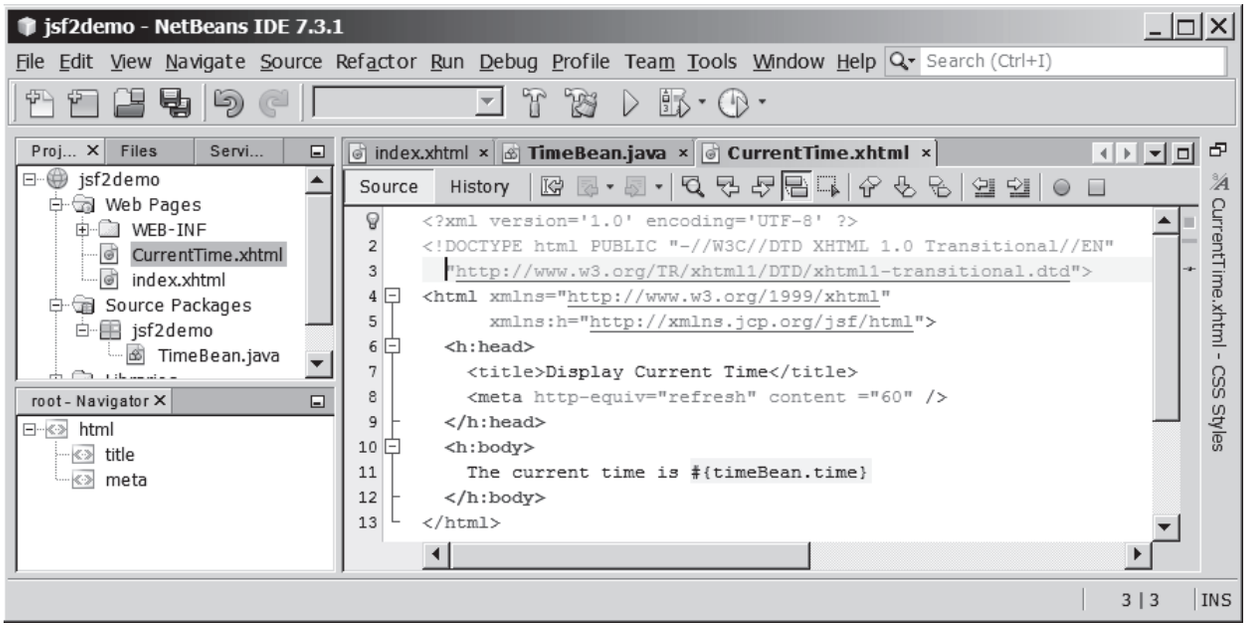


FIGURE 39.10 A New JSF page CurrentTime was created.

Line 11 defines a `meta` tag inside the `h:head` tag to tell the browser to refresh every 60 seconds. This line can also be written as

```
<meta http-equiv="refresh" content ="60"></ meta>
```

An element is called an *empty element* if there are no contents between the start tag and the end tag. In an empty element, data are typically specified as attributes in the start tag. You can close an empty element by placing a slash immediately preceding the start tag's right angle bracket, as shown in line 8, for brevity.

empty element

Line 8 uses a JSF expression `#{timeBean.time}` to obtain the current time. `timeBean` is an object of the `TimeBean` class. The object name can be changed in the `@Named` annotation (line 6 in Listing 39.2) using the following syntax:

```
@Named(name = "anyObjectName")
```

By default, the object name is the class name with the first letter in lowercase.

Note that `time` is a JavaBeans property because the `getTime()` method is defined in TimeBeans. The JSF expression can either use the property name or invoke the method to obtain the current time. So the following two expressions are fine.

```
#{timeBean.time}
#{timeBean.getTime()}
```

The syntax of a JSF expression is

```
#{expression}
```

JSF expressions bind JavaBeans objects with facelets. You will see more use of JSF expressions in the upcoming examples in this chapter.

39.2.1 What is JSF?

39.2.2 How do you create a JSF project in NetBeans?

39.2.3 How do you create a JSF page in a JSF project?

39.2.4 What is a facelet?

39.2.5 What is the file extension name for a facelet?

39.2.6 What is a managed bean?

39.2.7 What is the `@Named` annotation for?

39.2.8 What is the `@RequestScope` annotation for?

39.3 JSF GUI Components

JSF provides many elements for displaying GUI components.

Table 39.1 lists some of the commonly used elements. The tags with the `h` prefix are in the JSF HTML Tag library. The tags with the `f` prefix are in the JSF Core Tag library.

Listing 39.4 is an example that uses some of these elements to display a student registration form, as shown in Figure 39.11.

LISTING 39.4 StudentRegistrationForm.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html"
6        xmlns:f="http://xmlns.jcp.org/jsf/core">
7    <h:head>
8      <title>Student Registration Form</title>
9    </h:head>
10   <h:body>
11     <h:form>
12       <!-- Use h:graphicImage -->
13       <h3>Student Registration Form
14          <h:graphicImage name="usIcon.gif" library="image"/>
15       </h3>
```

jsf core namespace

graphicImage

TABLE 39.1 JSF GUI Form Elements

JSF Tag	*Description*
`h:form`	inserts an XHTML form into a page.
`h:panelGroup`	similar to a JavaFX FlowPane.
`h:panelGrid`	similar to a JavaFX GridPane.
`h:inputText`	displays a textbox for entering input.
`h:outputText`	displays a textbox for displaying output.
`h:inputTextArea`	displays a textarea for entering input.
`h:inputSecret`	displays a textbox for entering password.
`h:outputLabel`	displays a label.
`h:outputLink`	displays a hypertext link.
`h:selectOneMenu`	displays a combo box for selecting one item.
`h:selectOneRadio`	displays a set of radio button.
`h:selectManyCheckbox`	displays checkboxes.
`h:selectOneListbox`	displays a list for selecting one item.
`h:selectManyListbox`	displays a list for selecting multiple items.
`f:selectItem`	specifies an item in an `h:selectOneMenu`, `h:selectOneRadio`, or `h:selectManyListbox`.
`h:message`	displays a message for validating input.
`h:dataTable`	displays a data table.
`h:column`	specifies a column in a data table.
`h:graphicImage`	displays an image.

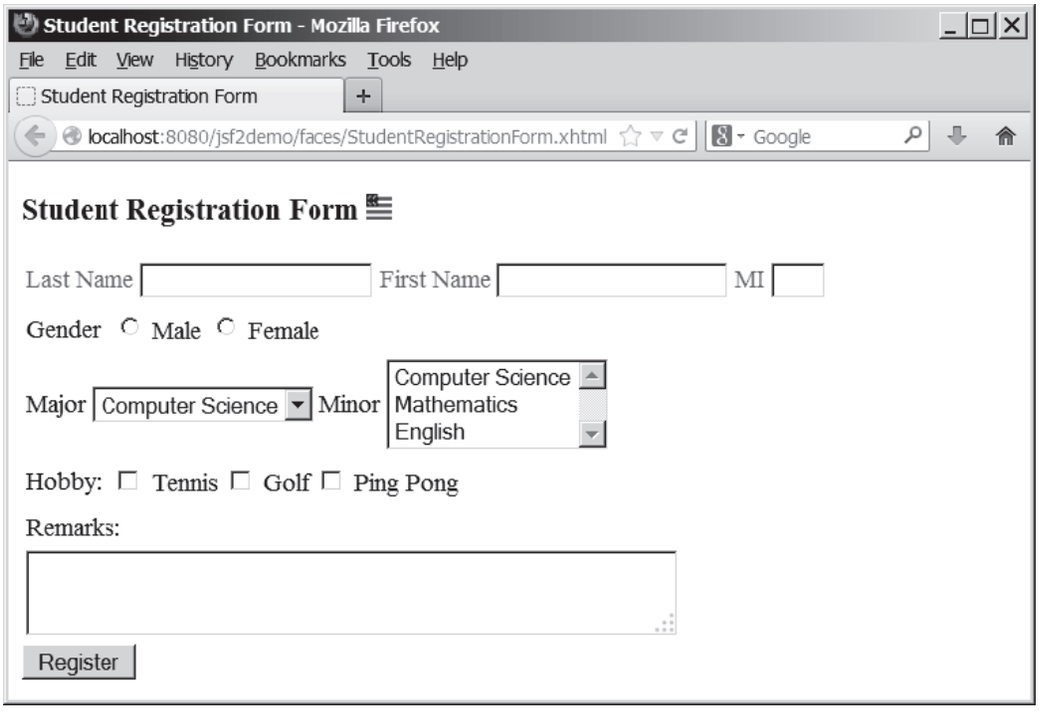


FIGURE 39.11 A student registration form is displayed using JSF elements.

```
                    16
                    17        <!-- Use h:panelGrid -->
h:panelGrid         18        <h:panelGrid columns="6" style="color:green">
h:outputLabel       19          <h:outputLabel value="Last Name"/>
h:inputText         20          <h:inputText id="lastNameInputText" />
                    21          <h:outputLabel value="First Name" />
                    22          <h:inputText id="firstNameInputText" />
                    23          <h:outputLabel value="MI" />
                    24          <h:inputText id="miInputText" size="1" />
                    25        </h:panelGrid>
                    26
                    27        <!-- Use radio buttons -->
                    28        <h:panelGrid columns="2">
                    29          <h:outputLabel>Gender </h:outputLabel>
h:selectOneRadio    30          <h:selectOneRadio id="genderSelectOneRadio">
f:selectItem        31            <f:selectItem itemValue="Male"
                    32                          itemLabel="Male"/>
                    33            <f:selectItem itemValue="Female"
                    34                          itemLabel="Female"/>
                    35          </h:selectOneRadio>
                    36        </h:panelGrid>
                    37
                    38        <!-- Use combo box and list -->
                    39        <h:panelGrid columns="4">
                    40          <h:outputLabel value="Major "/>
h:selectOneMenu     41          <h:selectOneMenu id="majorSelectOneMenu">
                    42            <f:selectItem itemValue="Computer Science"/>
                    43            <f:selectItem itemValue="Mathematics"/>
                    44          </h:selectOneMenu>
                    45          <h:outputLabel value="Minor "/>
h:selectManyListBox 46          <h:selectManyListbox id="minorSelectManyListbox">
                    47            <f:selectItem itemValue="Computer Science"/>
                    48            <f:selectItem itemValue="Mathematics"/>
                    49            <f:selectItem itemValue="English"/>
                    50          </h:selectManyListbox>
                    51        </h:panelGrid>
                    52
                    53        <!-- Use check boxes -->
                    54        <h:panelGrid columns="4">
                    55          <h:outputLabel value="Hobby: "/>
h:selectManyCheckbox 56         <h:selectManyCheckbox id="hobbySelectManyCheckbox">
                    57            <f:selectItem itemValue="Tennis"/>
                    58            <f:selectItem itemValue="Golf"/>
                    59            <f:selectItem itemValue="Ping Pong"/>
                    60          </h:selectManyCheckbox>
                    61        </h:panelGrid>
                    62
                    63        <!-- Use text area -->
                    64        <h:panelGrid columns="1">
                    65          <h:outputLabel>Remarks:</h:outputLabel>
h:inputTextarea     66          <h:inputTextarea id="remarksInputTextarea"
                    67                           style="width:400px; height:50px;" />
                    68        </h:panelGrid>
                    69
                    70        <!-- Use command button -->
h:commandButton     71        <h:commandButton value="Register" />
                    72      </h:form>
                    73    </h:body>
                    74  </html>
```

The tags with prefix `f` are in the JSF core tag library. Line 6

jsf core xmlns

```
xmlns:f="http://xmlns.jcp.org/jsf/core">
```

locates the library for these tags.

The `h:graphicImage` tag displays an image in the file usIcon.gif (line 14). The file is located in the /resources/image folder. In JSF 2.0, all resources (image files, audio files, and CCS files) should be placed under the `resources` folder under the **Web Pages** node. You can create these folders as follows:

h:graphicImage

Step 1: Right-click the Web Pages node in the project pane to display a context menu and choose *New*, *Folder* to display the New Folder dialog box. (If *Folder* is not in the context menu, choose *Other* to locate it.)

Step 2: Enter `resources` as the Folder Name and click *Finish* to create the `resources` folder, as shown in Figure 39.12.

Step 3: Right-click the `resources` node in the project pane to create the image folder under resources. You can now place usIcon.gif under the image folder.

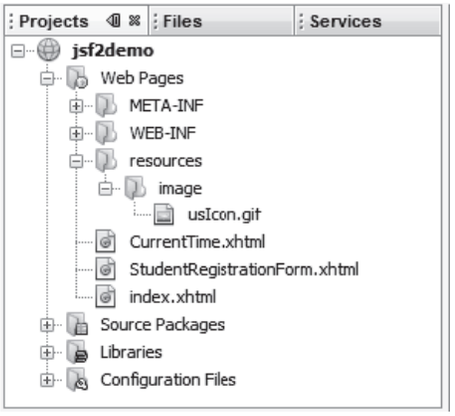


FIGURE 39.12 The resources folder was created.

JSF provides `h:panelGrid` and `h:panelGroup` elements to contain and layout subelements. `h:panelGrid` places the elements in a grid like the JavaFX `GridPane`. `h:panelGrid` places the elements in a grid with the specified number of columns. Lines 18–25 place six elements (labels and input texts) that are in an `h:panelGrid`. The `columns` attribute specifies that each row in the grid has `6` columns. The elements are placed into a row from left to right in the order they appear in the facelet. When a row is full, a new row is created to hold the elements. We used `h:panelGrid` in this example. You may replace it with `h:panelGroup` to see how the elements would be arranged.

h:panelGrid

You may use the `style` attribute with a JSF html tag to specify the CSS style for the element and its subelements. The `style` attribute in line 18 specifies color green for all elements in this `h:panelGrid` element.

the `style` attribute

The `h:outputLabel` element is for displaying a label (line 19). The `value` attribute specifies the label's text.

h:outputLabel

The `h:inputText` element is for displaying a text input box for the user to enter a text (line 20). The `id` attribute is useful for other elements or the server program to reference this element.

h:inputText

The `h:selectOneRadio` element is for displaying a group of radio buttons (line 30). Each radio button is defined using an `f:selectItem` element (lines 31–34).

h:selectOneRadio

h:selectOneMenu

The `h:selectOneMenu` element is for displaying a combo box (line 41). Each item in the combo box is defined using an `f:selectItem` element (lines 42 and 43).

h:selectManyListbox

The `h:selectManyListbox` element is for displaying a list for the user to choose multiple items in a list (line 46). Each item in the list is defined using an `f:selectItem` element (lines 47–49).

h:selectManyCheckbox

The `h:selectManyCheckbox` element is for displaying a group of check boxes (line 56). Each item in the check box is defined using an `f:selectItem` element (lines 57–59).

h:selectTextarea

The `h:selectTextarea` element is for displaying a text area for multiple lines of input (line 66). The `style` attribute is used to specify the width and height of the text area (line 67).

h:commandButton

The `h:commandButton` element is for displaying a button (line 71). When the button is clicked, an action is performed. The default action is to request the same page from the server. The next section shows how to process the form.

39.3.1 What is the name space for JSF tags with prefix `h` and prefix `f`?

39.3.2 Describe the use of the following tags?

`h:form`, `h:panelGroup`, `h:panelGrid`, `h:inputText`, `h:outputText`, `h:inputTextArea`, `h:inputSecret`, `h:outputLabel`, `h:outputLink`, `h:selectOneMenu`, `h:selectOneRadio`, `h:selectBooleanCheckbox`, `h:selectOneListbox`, `h:selectManyListbox`, `h:selectItem`, `h:message`, `h:dataTable`, `h:columm`, `h:graphicImage`

39.4 Processing the Form

Processing forms is a common task for Web programming. JSF provides tools for processing forms.

The preceding section introduced how to display a form using common JSF elements. This section shows how to obtain and process the input.

To obtain input from the form, simply bind each input element with a property in a managed bean. We now define a managed bean named `registration` as shown in Listing 39.5.

LISTING 39.5 RegistrationJSFBean.java

managed bean
request scope
property `lastName`

```
1  package jsf2demo;
2
3  import javax.enterprise.context.RequestScoped;
4  import javax.inject.Named;
5
6  @Named(value = "registration")
7  @RequestScoped
8  public class RegistrationJSFBean {
9    private String lastName;
10   private String firstName;
11   private String mi;
12   private String gender;
13   private String major;
14   private String[] minor;
15   private String[] hobby;
16   private String remarks;
17
18   public RegistrationJSFBean() {
19   }
20
21   public String getLastName() {
22     return lastName;
23   }
```

```
24
25    public void setLastName(String lastName) {
26      this.lastName = lastName;
27    }
28
29    public String getFirstName() {
30      return firstName;
31    }
32
33    public void setFirstName(String firstName) {
34      this.firstName = firstName;
35    }
36
37    public String getMi() {
38      return mi;
39    }
40
41    public void setMi(String mi) {
42      this.mi = mi;
43    }
44
45    public String getGender() {
46      return gender;
47    }
48
49    public void setGender(String gender) {
50      this.gender = gender;
51    }
52
53    public String getMajor() {
54      return major;
55    }
56
57    public void setMajor(String major) {
58      this.major = major;
59    }
60
61    public String[] getMinor() {
62      return minor;
63    }
64
65    public void setMinor(String[] minor) {
66      this.minor = minor;
67    }
68
69    public String[] getHobby() {
70      return hobby;
71    }
72
73    public void setHobby(String[] hobby) {
74      this.hobby = hobby;
75    }
76
77    public String getRemarks() {
78      return remarks;
79    }
80
81    public void setRemarks(String remarks) {
82      this.remarks = remarks;
83    }
```

getResponse()

```
84
85    public String getResponse() {
86      if (lastName == null)
87        return ""; // Request has not been made
88      else {
89        String allMinor = "";
90        for (String s: minor) {
91          allMinor += s + " ";
92        }
93
94        String allHobby = "";
95        for (String s: hobby) {
96          allHobby += s + " ";
97        }
98
99        return "<p style=\"color:red\">You entered <br />" +
100         "Last Name: " + lastName + "<br />" +
101         "First Name: " + firstName + "<br />" +
102         "MI: " + mi + "<br />" +
103         "Gender: " + gender + "<br />" +
104         "Major: " + major + "<br />" +
105         "Minor: " + allMinor + "<br />" +
106         "Hobby: " + allHobby + "<br />" +
107         "Remarks: " + remarks + "</p>";
108     }
109   }
110 }
```

bean properties

The `RegistrationJSFBean` class is a managed bean that defines the properties `lastName`, `firstName`, `mi`, `gender`, `major`, `minor`, and `remarks`, which will be bound to the elements in the JSF registration form.

The registration form can now be revised as shown in Listing 39.6. Figure 39.13 shows that new JSF page displays the user input upon clicking the *Register* button.

LISTING 39.6 ProcessStudentRegistrationForm.xhtml

jsf core namespace

bind lastName

bind firstName

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html"
6        xmlns:f="http://xmlns.jcp.org/jsf/core">
7    <h:head>
8      <title>Student Registration Form</title>
9    </h:head>
10   <h:body>
11     <h:form>
12       <!-- Use h:graphicImage -->
13       <h3>Student Registration Form
14         <h:graphicImage name="usIcon.gif" library="image"/>
15       </h3>
16
17       <!-- Use h:panelGrid -->
18       <h:panelGrid columns="6" style="color:green">
19         <h:outputLabel value="Last Name"/>
20         <h:inputText id="lastNameInputText"
21                      value="#{registration.lastName}"/>
22         <h:outputLabel value="First Name" />
23         <h:inputText id="firstNameInputText"
24                      value="#{registration.firstName}"/>
```

```
25          <h:outputLabel value="MI" />
26          <h:inputText id="miInputText" size="1"
27                       value="#{registration.mi}"/>                    bind mi
28          </h:panelGrid>
29
30          <!-- Use radio buttons -->
31          <h:panelGrid columns="2">
32            <h:outputLabel>Gender </h:outputLabel>
33            <h:selectOneRadio id="genderSelectOneRadio"
34                              value="#{registration.gender}">           bind gender
35            <f:selectItem itemValue="Male"
36                          itemLabel="Male"/>
37            <f:selectItem itemValue="Female"
38                          itemLabel="Female"/>
39          </h:selectOneRadio>
40        </h:panelGrid>
41
42        <!-- Use combo box and list -->
43        <h:panelGrid columns="4">
44          <h:outputLabel value="Major "/>
45          <h:selectOneMenu id="majorSelectOneMenu"
46                           value="#{registration.major}">               bind major
47            <f:selectItem itemValue="Computer Science"/>
48            <f:selectItem itemValue="Mathematics"/>
49          </h:selectOneMenu>
50          <h:outputLabel value="Minor "/>
51          <h:selectManyListbox id="minorSelectManyListbox"
52                               value="#{registration.minor}">           bind minor
53            <f:selectItem itemValue="Computer Science"/>
54            <f:selectItem itemValue="Mathematics"/>
55            <f:selectItem itemValue="English"/>
56          </h:selectManyListbox>
57        </h:panelGrid>
58
59        <!-- Use check boxes -->
60        <h:panelGrid columns="4">
61          <h:outputLabel value="Hobby: "/>
62          <h:selectManyCheckbox id="hobbySelectManyCheckbox"
63                                value="#{registration.hobby}">           bind hobby
64            <f:selectItem itemValue="Tennis"/>
65            <f:selectItem itemValue="Golf"/>
66            <f:selectItem itemValue="Ping Pong"/>
67          </h:selectManyCheckbox>
68        </h:panelGrid>
69
70        <!-- Use text area -->
71        <h:panelGrid columns="1">
72          <h:outputLabel>Remarks:</h:outputLabel>
73          <h:inputTextarea id="remarksInputTextarea"
74                           style="width:400px; height:50px;"
75                           value="#{registration.remarks}"/>            bind remarks
76        </h:panelGrid>
77
78        <!-- Use command button -->
79        <h:commandButton value="Register" />
80        <br />
81        <h:outputText escape="false" style="color:red"
82                      value="#{registration.response}" />               bind response
83      </h:form>
84    </h:body>
85  </html>
```

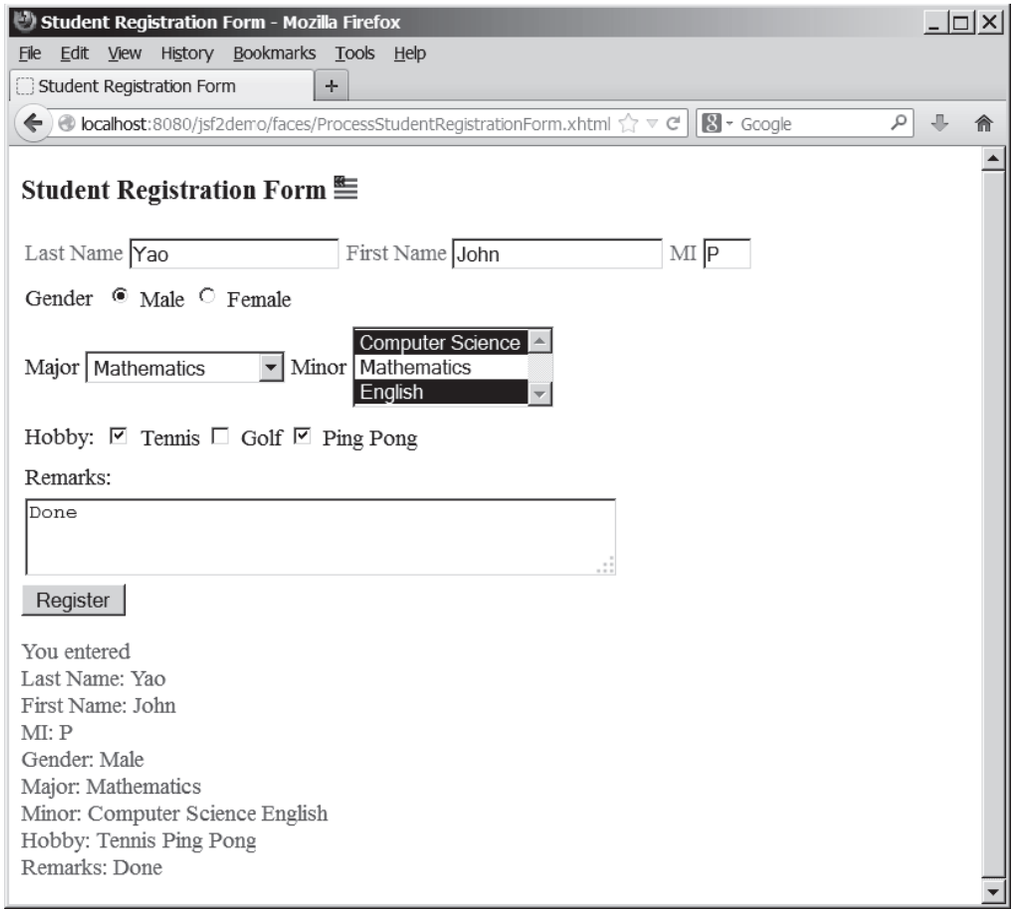


FIGURE 39.13 The user input is collected and displayed after clicking the Register button.

binding input texts

The new JSF form in this listing binds the `h:inputText` element for last name, first name, and mi with the properties `lastName`, `firstName`, and `mi` in the managed bean (lines 21, 24, and 27). When the *Register* button is clicked, the page is sent to the server, which invokes the setter methods to set the properties in the managed bean.

binding radio buttons

The `h:selectOneRadio` element is bound to the `gender` property (line 34). Each radio button has an `itemValue`. The selected radio button's `itemValue` is set to the `gender` property in the bean when the page is sent to the server.

binding combo box

The `h:selectOneMenu` element is bound to the `major` property (line 46). When the page is sent to the server, the selected item is returned as a string and is set to the `major` property.

binding list box

The `h:selectManyListbox` element is bound to the `minor` property (line 52). When the page is sent to the server, the selected items are returned as an array of strings and set to the `minor` property.

binding check boxes

The `h:selectManyCheckbox` element is bound to the `hobby` property (line 63). When the page is sent to the server, the checked boxes are returned as an array of `itemValues` and set to the `hobby` property.

binding text area

The `h:selectTextarea` element is bound to the `remarks` property (line 75). When the page is sent to the server, the content in the text area is returned as a string and set to the `remarks` property.

binding response

The `h:outputText` element is bound to the `response` property (line 82). This is a read-only property in the bean. It is `""` if `lastName` is `null` (lines 86 and 87 in Listing 39.5). When the page is returned to the client, the `response` property value is displayed in the output text element (line 82).

The `h:outputText` element's `escape` attribute is set to `false` (line 81) to enable the contents to be displayed in HTML. By default, the `escape` attribute is `true`, which indicates the contents are considered regular text.

escape attribute

39.4.1 In the `h:outputText` tag, what is the `escape` attribute for?

39.4.2 Does every GUI component tag in JSF have the style attribute?

Check Point

39.5 Case Study: Calculator

This section gives a case study on using GUI elements and processing forms.

Key Point

This section uses JSF to develop a calculator to perform addition, subtraction, multiplication, and division, as shown in Figure 39.14.

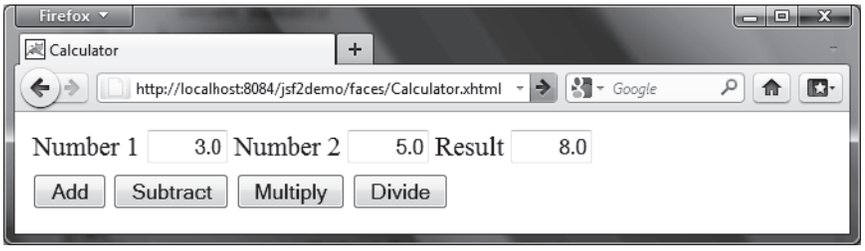


FIGURE 39.14 This JSF application enables you to perform addition, subtraction, multiplication, and division.

Here are the steps to develop this project:

Step 1. Create a new managed bean named `calculator` with the request scope as shown in Listing 39.7, CalculatorJSFBean.java.

create managed bean

Step 2. Create a JSF facelet in Listing 39.8, Calculator.xhtml.

create JSF facelet

LISTING 39.7 CalculatorJSFBean.java

```
1  package jsf2demo;
2
3  import javax.inject.Named;
4  import javax.enterprise.context.RequestScoped;
5
6  @Named(value = "calculator")
7  @RequestScoped
8  public class CalculatorJSFBean {
9    private Double number1;
10   private Double number2;
11   private Double result;
12
13   public CalculatorJSFBean() {
14   }
15
16   public Double getNumber1() {
17     return number1;
18   }
19
20   public Double getNumber2() {
21     return number2;
22   }
23
```

property `number1`
property `number2`
property `result`

```
24    public Double getResult() {
25      return result;
26    }
27
28    public void setNumber1(Double number1) {
29      this.number1 = number1;
30    }
31
32    public void setNumber2(Double number2) {
33      this.number2 = number2;
34    }
35
36    public void setResult(Double result) {
37      this.result = result;
38    }
39
40    public void add() {                                    add
41      result = number1 + number2;
42    }
43
44    public void subtract() {                               subtract
45      result = number1 - number2;
46    }
47
48    public void divide() {                                 divide
49      result = number1 / number2;
50    }
51
52    public void multiply() {                               multiply
53      result = number1 * number2;
54    }
55  }
```

The managed bean has three properties `number1`, `number2`, and `result` (lines 9–38). The methods `add()`, `subtract()`, `divide()`, and `multiply()` add, subtract, multiply, and divide `number1` with `number2` and assigns the result to `result` (lines 40–54).

LISTING 39.8 Calculator.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html">
6    <h:head>
7      <title>Calculator</title>
8    </h:head>
9    <h:body>
10     <h:form>
11       <h:panelGrid columns="6">
12         <h:outputLabel value="Number 1"/>
13         <h:inputText id="number1InputText" size ="4"
14                      style="text-align: right"                 right align
15                      value="#{calculator.number1}"/>           bind text input
16         <h:outputLabel value="Number 2" />
17         <h:inputText id="number2InputText" size ="4"
18                      style="text-align: right"
19                      value="#{calculator.number2}"/>
20         <h:outputLabel value="Result" />
21         <h:inputText id="resultInputText" size ="4"
```

```
22                      style="text-align: right"
23                      value="#{calculator.result}"/>
24        </h:panelGrid>
25
26        <h:panelGrid columns="4">
27          <h:commandButton value="Add"
28                           action ="#{calculator.add}"/>                    action
29          <h:commandButton value="Subtract"
30                           action ="#{calculator.subtract}"/>
31          <h:commandButton value="Multiply"
32                           action ="#{calculator.multiply}"/>
33          <h:commandButton value="Divide"
34                           action ="#{calculator.divide}"/>
35        </h:panelGrid>
36      </h:form>
37    </h:body>
38  </html>
```

Three text input components along with their labels are placed in the grid panel (lines 11–24). Four button components are placed in the grid panel (lines 26–35).

The bean property `number1` is bound to the text input for Number 1 (line 15). The CSS style `text-align: right` (line 14) specifies that the text is right aligned in the input box.

The `action` attribute for the *Add* button is set to the `add` method in the calculator bean (line 28). When the *Add* button is clicked, the `add` method in the bean is invoked to add `number1` with `number2` and assign the result to `result`. Since the `result` property is bound to the Result input text (line 23), the new result is now displayed in the text input field.

39.6 Session Tracking

You can create a managed bean at the application scope, session scope, view scope, or request scope.

scope
request scope
view scope
session scope
application scope

JSF supports session tracking for managed beans at the application scope, session scope, view scope, and request scope. The *scope* is the lifetime of a bean. A *request*-scoped bean is alive in a single HTTP request. After the request is processed, the bean is no longer alive. A *view*-scoped bean lives as long as you are in the same JSF page. A *session*-scoped bean is alive for the entire Web session between a client and the server. An *application*-scoped bean lives as long as the Web application runs. In essence, a request-scoped bean is created once for a request; a view-scoped bean is created once for the view; a session-scoped bean is created once for the entire session; and an application-scoped bean is created once for the entire application. A managed bean with a session scope must be serializable because the system may need to free resources during and session and stores the bean to a file if the bean is not used for a while. When the bean is used again, the system will restore the bean to the memory.

Consider the following example that prompts the user to guess a number. When the page starts, the program randomly generates a number between `0` and `99`. This number is stored in a bean. When the user enters a guess, the program checks the guess with the random number in the bean and tells the user whether the guess is too high, too low, or just right, as shown in Figure 39.15.

Here are the steps to develop this project:

Step 1. Create a new managed bean named `guessNumber` with the view scope as shown in Listing 39.9, GuessNumberJSFBean.java. — create managed bean

Step 2. Create a JSF facelet in Listing 39.10, GuessNumber.xhtml. — create JSF facelet

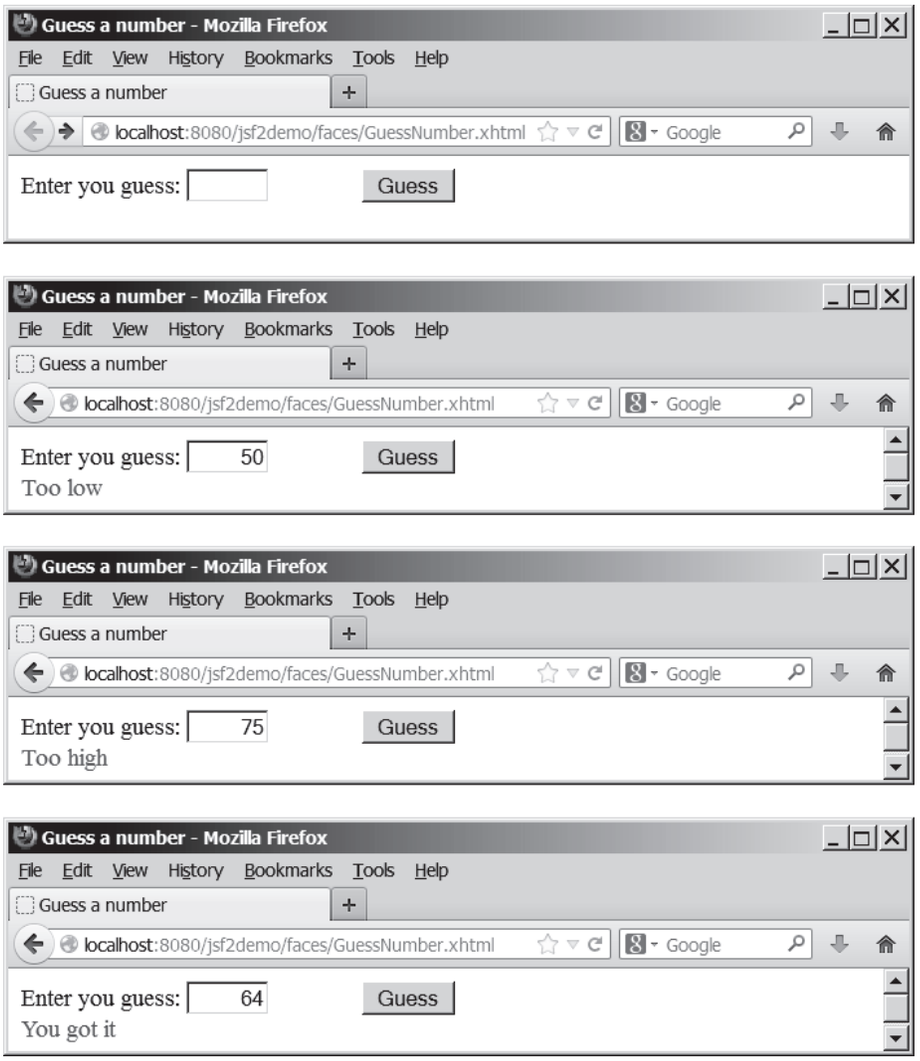


FIGURE 39.15 The user enters a guess and the program displays the result.

LISTING 39.9 GuessNumberJSFBean.java

```
1  package jsf2demo;
2
3  import javax.inject.Named;
4  import javax.faces.view.ViewScoped;
5
6  @Named(value = "guessNumber")
7  @ViewScoped
8  public class GuessNumberJSFBean {
9    private int number;
10   private String guessString;
11
12   public GuessNumberJSFBean() {
13    number = (int)(Math.random() * 100);
14   }
15
16   public String getGuessString() {
17     return guessString;
18   }
```

view scope

random number
guess by user

create random number

getter method

```
19
20    public void setGuessString(String guessString) {                    setter method
21      this.guessString = guessString;
22    }
23
24    public String getResponse() {                                       get response
25      if (guessString == null)
26        return ""; // No user input yet
27
28      int guess = Integer.parseInt(guessString);                        check guess
29      if (guess < number)
30        return "Too low";
31      else if (guess == number)
32        return "You got it";
33      else
34        return "Too high";
35    }
36  }
```

The managed bean uses the `@ViewScope` annotation (line 7) to set up the view scope for the bean. The view scope is most appropriate for this project. The bean is alive as long as the view is not changed. The bean is created when the page is displayed for the first time. A random number between `0` and `99` is assigned to `number` (line 13) when the bean is created. This number will not change as long as the bean is alive in the same view.

The `getResponse` method converts `guessString` from the user input to an integer (line 28) and determines if the guess is too low (line 30), too high (line 34), and just right (line 32).

LISTING 39.10 GuessNumber.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html">
6    <h:head>
7      <title>Guess a number</title>
8    </h:head>
9    <h:body>
10     <h:form>
11       <h:outputLabel value="Enter you guess: "/>
12       <h:inputText style="text-align: right; width: 50px"
13                    id="guessInputText"
14                    value="#{guessNumber.guessString}"/>                bind text input
15       <h:commandButton style="margin-left: 60px" value="Guess" />
16       <br />
17       <h:outputText style="color: red"
18                     value="#{guessNumber.response}" />                 bind text output
19     </h:form>
20   </h:body>
21 </html>
```

The bean property `guessString` is bound to the text input (line 14). The CSS style `text-align: right` (line 13) specifies that the text is right aligned in the input box.

The CSS style `margin-left: 60px` (line 15) specifies that the command button has a left margin of 60 pixels.

The bean property `response` is bound to the text output (line 18). The CSS style `color: red` (line 17) specifies that the text is displayed in red in the output box.

The project uses the **view** scope. What happens if the scope is changed to the request scope? Every time the page is refreshed, JSF creates a new bean with a new random number. What happens if the scope is changed to the **session** scope? The bean will be alive as long as the scope

browser is alive. What happens if the scope is changed to the **application** scope? The bean will be created once when the application is launched from the server. So every client will use the same random number.

39.6.1 What is a scope? What are the available scopes in JSF? Explain request scope, view scope, session scope, and application scope. How do you set a request scope, view scope, session scope, and application scope in a managed bean?

39.6.2 What happens if the bean scope in Listing 39.9 GuessNumberJSFBean.java is changed to request?

39.6.3 What happens if the bean scope in Listing 39.9 GuessNumberJSFBean.java is changed to session?

39.6.4 What happens if the bean scope in Listing 39.9 GuessNumberJSFBean.java is changed to application?

39.7 Validating Input

JSF provides tools for validating user input.

In the preceding `GuessNumber` page, an error would occur if you entered a noninteger in the input box before clicking the *Guess* button. One way to fix the problem is to check the text field before processing any event. But a better way is to user the validators. You can use the standard validator tags in the JSF Core Tag Library or create custom validators. Table 39.2 lists some JSF input validator tags.

TABLE 39.2 JSF Input Validator Tags

JSF Tag	*Description*
`f:validateLength`	validates the length of the input.
`f:validateDoubleRange`	validates whether numeric input falls within acceptable range of double values.
`f:validateLongRange`	validates whether numeric input falls within acceptable range of long values.
`f:validateRequired`	validates whether a field is not empty.
`f:validateRegex`	validates whether the input matches a regualar expression.
`f:validateBean`	invokes a custom method in a bean to perform custom validation.

Consider the following example that displays a form for collecting user input as shown in Figure 39.16. All text fields in the form must be filled. If not, error messages are displayed. The SSN must be formatted correctly. If not, an error is displayed. If all input are correct, clicking *Submit* displays the result in an output text, as shown in Figure 39.17.

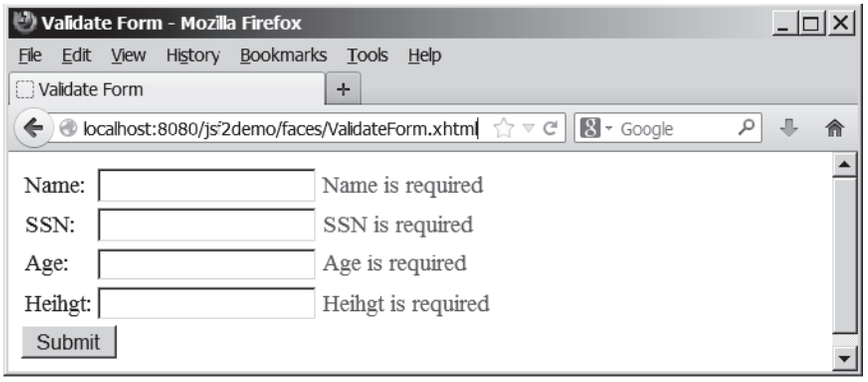
Here are the steps to create this project.

Step 1. Create a new page in Listing 39.11, ValidateForm.xhtml.

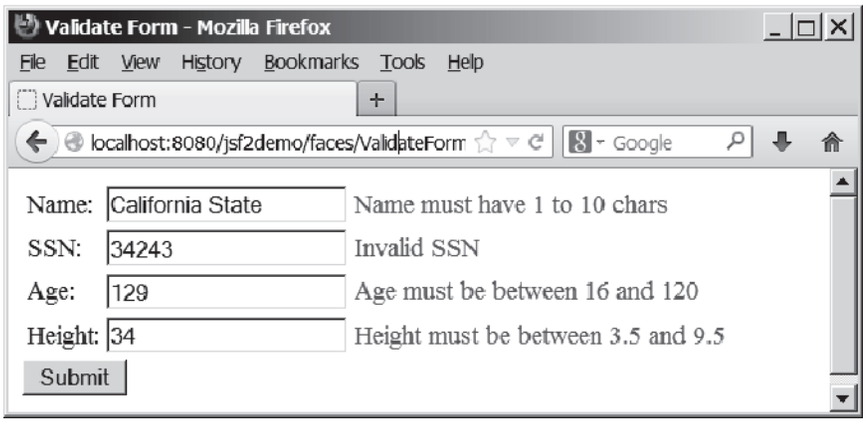
Step 2. Create a new managed bean named `validateForm`, as shown in Listing 39.12.

LISTING 39.11 ValidateForm.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html"
6        xmlns:f="http://xmlns.jcp.org/jsf/core">
7    <h:head>
```



(a) *The required messages are displayed if input is required, but empty.*



(b) *Error messages are displayed if input is incorrect.*

FIGURE 39.16 The input fields are validated.

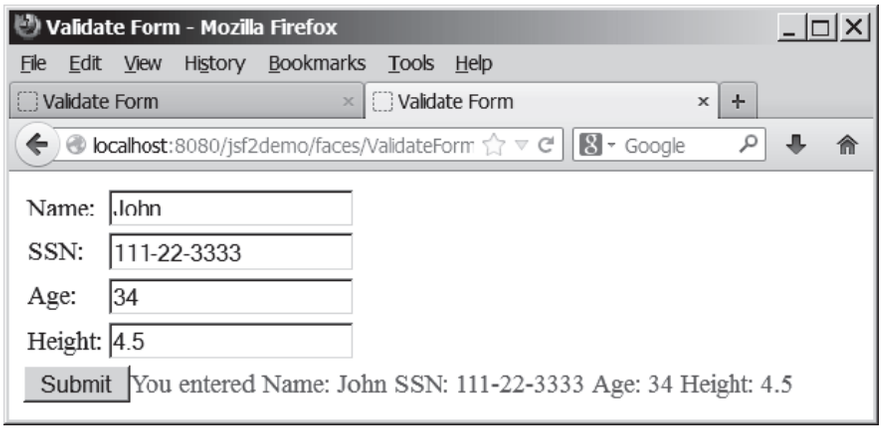


FIGURE 39.17 The correct input values are displayed.

```
 8      <title>Validate Form</title>
 9    </h:head>
10    <h:body>
11      <h:form>
12        <h:panelGrid columns="3">
13          <h:outputLabel value="Name:"/>
```

required input
required message
validator message

validate length

message element

validate regex

validate integer range

validate double range

```
14          <h:inputText id="nameInputText" required="true"
15            requiredMessage="Name is required"
16            validatorMessage="Name must have 1 to 10 chars"
17            value="#{validateForm.name}">
18            <f:validateLength minimum="1" maximum="10" />
19          </h:inputText>
20          <h:message for="nameInputText" style="color:red"/>
21
22          <h:outputLabel value="SSN:" />
23          <h:inputText id="ssnInputText" required="true"
24            requiredMessage="SSN is required"
25            validatorMessage="Invalid SSN"
26            value="#{validateForm.ssn}">
27            <f:validateRegex pattern="[\d]{3}-[\d]{2}-[\d]{4}"/>
28          </h:inputText>
29          <h:message for="ssnInputText" style="color:red"/>
30
31          <h:outputLabel value="Age:" />
32          <h:inputText id="ageInputText" required="true"
33            requiredMessage="Age is required"
34            validatorMessage="Age must be between 16 and 120"
35            value="#{validateForm.ageString}">
36            <f:validateLongRange minimum="16" maximum="120"/>
37          </h:inputText>
38          <h:message for="ageInputText" style="color:red"/>
39
40          <h:outputLabel value="Height:" />
41          <h:inputText id="heightInputText" required="true"
42            requiredMessage="Height is required"
43            validatorMessage="Height must be between 3.5 and 9.5"
44            value="#{validateForm.heightString}">
45            <f:validateDoubleRange minimum="3.5" maximum="9.5"/>
46          </h:inputText>
47          <h:message for="heightInputText" style="color:red"/>
48        </h:panelGrid>
49
50        <h:commandButton value="Submit" />
51
52        <h:outputText style="color:red"
53                      value="#{validateForm.response}" />
54      </h:form>
55    </h:body>
56  </html>
```

`required` attribute
`requiredMessage`

For each input text field, set its `required` attribute `true` (lines 14, 23, 32, and 41) to indicate that an input value is required for the field. When a required input field is empty, the `requiredMessage` is displayed (lines 15, 24, 33, and 42).

`validatorMessage`
`f:validateLength`

The `validatorMessage` attribute specifies a message to be displayed if the input field is invalid (line 16). The `f:validateLength` tag specifies the minimum or maximum length of the input (line 18). JSF will determine whether the input length is valid.

`h:message`

The `h:message` element displays the `validatorMessage` if the input is invalid. The element's `for` attribute specifies the `id` of the element for which the message will be displayed (line 20).

`f:validateRegex`

The `f:validateRegex` tag specifies a regular expression for validating the input (line 27). For information on regular expression, see Appendix H.

f:validateLongRange

The `f:validateLongRange` tag specifies a range for an integer input using the `minimum` and `maximum` attributes (line 45). In this project, a valid age value is between `16` and `120`.

f:validateDoubleRange

The `f:validateDoubleRange` tag specifies a range for a double input using the `minimum` and `maximum` attributes (line 36). In this project, a valid height value is between `3.5` and `9.5`.

LISTING 39.12 ValidateFormJSFBean.java

```
1  package jsf2demo;
2
3  import javax.enterprise.context.RequestScoped;
4  import javax.inject.Named;
5
6  @Named(value = "validateForm")
7  @RequestScoped
8  public class ValidateFormJSFBean {
9    private String name;
10   private String ssn;
11   private String ageString;
12   private String heightString;
13
14   public String getName() {
15     return name;
16   }
17
18   public void setName(String name) {
19     this.name = name;
20   }
21
22   public String getSsn() {
23     return ssn;
24   }
25
26   public void setSsn(String ssn) {
27     this.ssn = ssn;
28   }
29
30   public String getAgeString() {
31     return ageString;
32   }
33
34   public void setAgeString(String ageString) {
35     this.ageString = ageString;
36   }
37
38   public String getHeightString() {
39     return heightString;
40   }
41
42   public void setHeightString(String heightString) {
43     this.heightString = heightString;
44   }
45
46   public String getResponse() {
47     if (name == null || ssn == null || ageString == null
48             || heightString == null) {
49       return "";                                          some input not set
50     }
51     else {
52       return "You entered " +
53         " Name: " + name +
54         " SSN: " + ssn +
55         " Age: " + ageString +
56         " Height: " + heightString;
57     }
58   }
59 }
```

If an input is invalid, its value is not set to the bean. So only when all input are correct, the `getResponse()` method will return all input values (lines 46–58).

Check Point

39.7.1 Write a tag that validates an input text with minimal length of `2` and maximum `12`.

39.7.2 Write a tag that validates an input text for SSN using a regular expression.

39.7.3 Write a tag that validates an input text for a double value with minimal `4.5` and maximum `19.9`.

39.7.4 Write a tag that validates an input text for an integer value with minimal `4` and maximum `20`.

39.7.5 Write a tag that makes an input text required.

Key Point

39.8 Binding Database with Facelets

You can bind a database in JSF applications.

Often you need to access a database from a Web page. This section gives examples of building Web applications using databases.

Consider the following example that lets the user choose a course, as shown in Figure 39.18. After a course is selected in the combo box, the students enrolled in the course are displayed in the table, as shown in Figure 39.19. In this example, all the course titles in the `Course` table are bound to the combo box and the query result for the students enrolled in the course is bound to the table.

Here are the steps to create this project:

Managed bean

Step 1. Create a managed bean named `courseName` with application scope, as shown in Listing 39.13.

JSF page

Step 2. Create a JSF in Listing 39.14, DisplayStudent.xhtml.

style sheet

Step 3. Create a cascading style sheet for formatting the table as follows:

Step 3.1. Right-click the `resources` node to choose *New*, *Others* to display the New File dialog box, as shown in Figure 39.20.

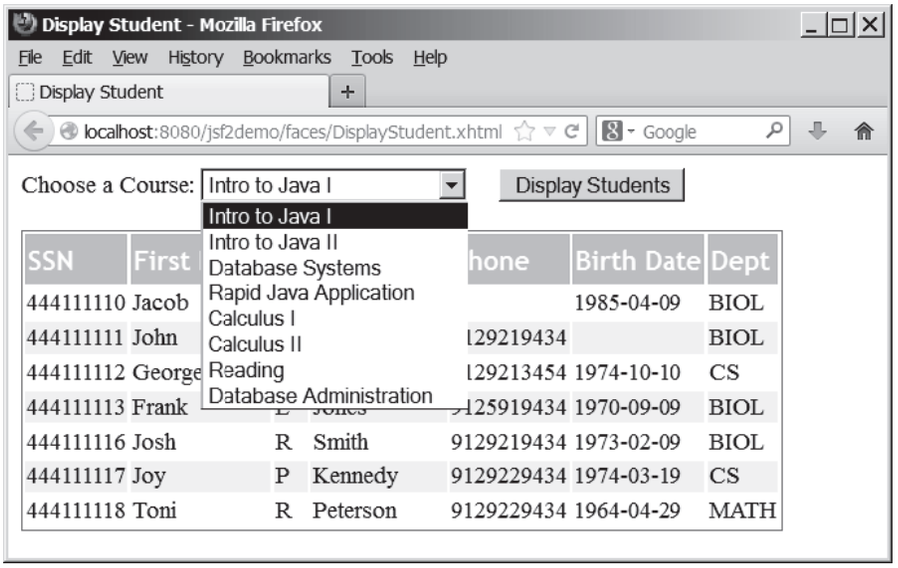


FIGURE 39.18 You need to choose a course and display the students enrolled in the course.

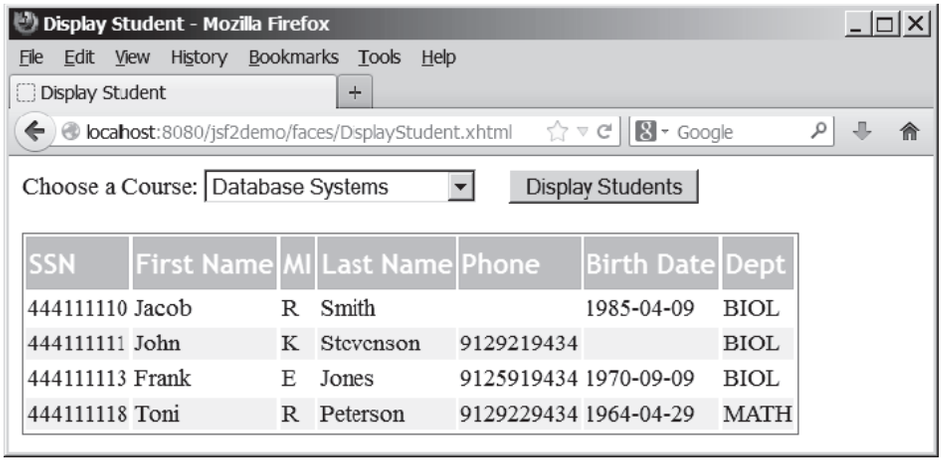


FIGURE 39.19 The table displays the students enrolled in the course.

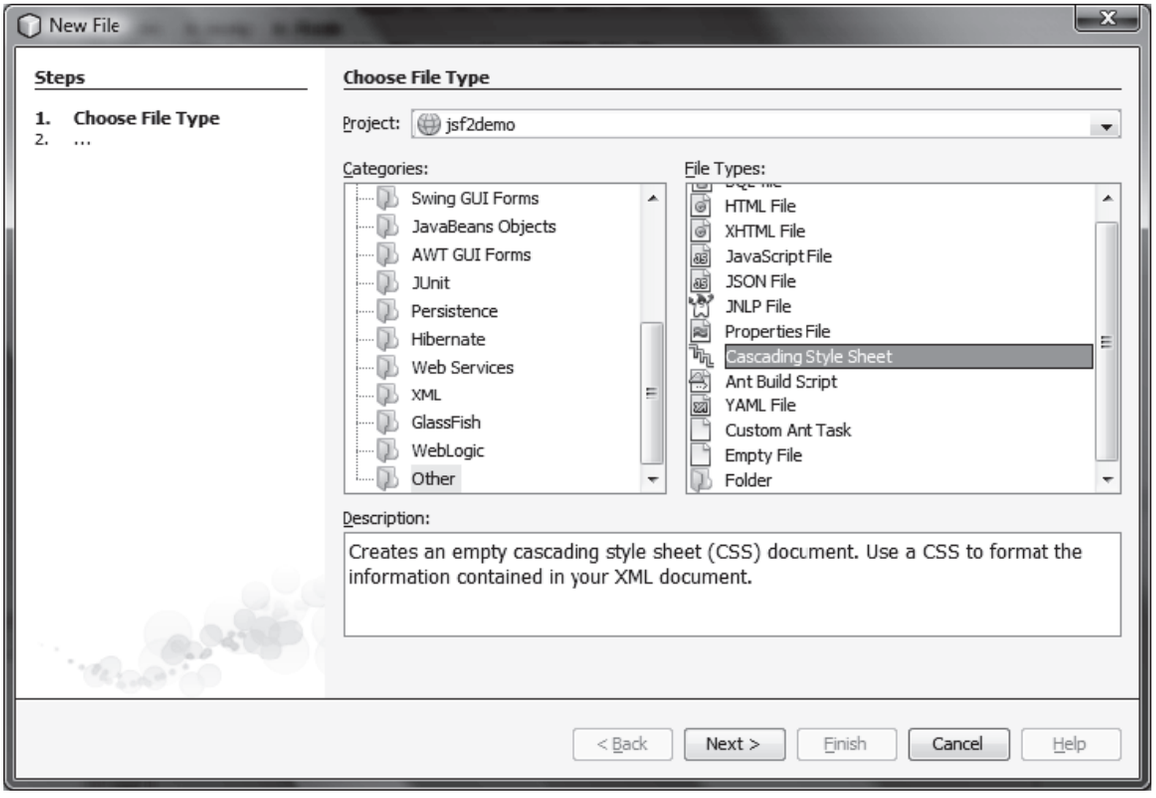


FIGURE 39.20 You can create CSS files for Web project in NetBenas.

Step 3.2. Choose `Other` in the Categories section and `Cascading Style Sheet` in the File Types section to display the New Cascading Style Sheet dialog box, as shown in Figure 39.21.

Step 3.3. Enter `tablestyle` as the File Name and click *Finish* to create tablestyle.css under the resources node.

Step 3.4. Define the CSS style as shown in Listing 39.15.

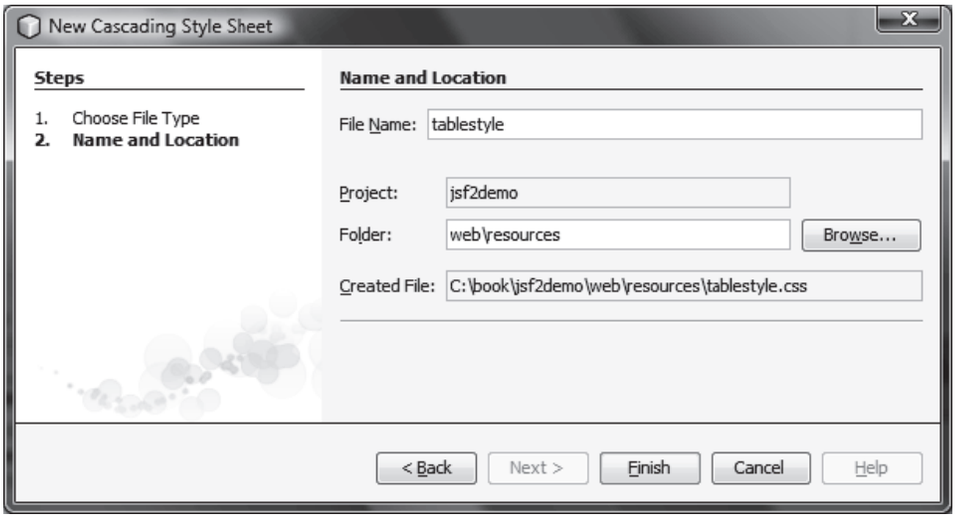


FIGURE 39.21 The New Cascading Style Sheet dialog box creates a new style sheet file.

LISTING 39.13 CourseNameJSFBean.java

```
1  package jsf2demo;
2
3  import java.sql.*;
4  import java.util.ArrayList;
5  import javax.enterprise.context.ApplicationScoped;
6  import javax.inject.Named;
7
8  @Named(value = "courseName")
9  @ApplicationScoped
10 public class CourseNameJSFBean {
11   private PreparedStatement studentStatement = null;
12   private String choice; // Selected course
13   private String[] titles; // Course titles
14
15   /** Creates a new instance of CourseName */
16   public CourseNameJSFBean() {
17     initializeJdbc();
18   }
19
20   /** Initialize database connection */
21   private void initializeJdbc() {
22     try {
23       Class.forName("com.mysql.jdbc.Driver");
24       System.out.println("Driver loaded");
25
26       // Connect to the sample database
27       Connection connection = DriverManager.getConnection(
28         "jdbc:mysql://localhost/javabook", "scott", "tiger");
29
30       // Get course titles
31       PreparedStatement statement = connection.prepareStatement(
32         "select title from course");
33
34       ResultSet resultSet = statement.executeQuery();
35
36       // Store resultSet into array titles
37       ArrayList<String> list = new ArrayList<>();
```

application scope

initialize JDBC

connect to database

get course titles

execute SQL

```
38        while (resultSet.next()) {
39          list.add(resultSet.getString(1));
40        }
41        titles = new String[list.size()]; // Array for titles              titles array
42        list.toArray(titles); // Copy strings from list to array
43
44        // Define a SQL statement for getting students
45        studentStatement = connection.prepareStatement(
46          "select Student.ssn, "
47          + "student.firstName, Student.mi, Student.lastName, "
48          + "Student.phone, Student.birthDate, Student.street, "
49          + "Student.zipCode, Student.deptId "
50          + "from Student, Enrollment, Course "
51          + "where Course.title = ? "
52          + "and Student.ssn = Enrollment.ssn "
53          + "and Enrollment.courseId = Course.courseId;");
54      }
55      catch (Exception ex) {
56        ex.printStackTrace();
57      }
58    }
59
60    public String[] getTitles() {
61      return titles;
62    }
63
64    public String getChoice() {
65      return choice;
66    }
67
68    public void setChoice(String choice) {
69      this.choice = choice;
70    }
71
72    public ResultSet getStudents() throws SQLException {                   get students
73      if (choice == null) {
74        if (titles.length == 0)
75          return null;
76        else
77          studentStatement.setString(1, titles[0]);                        set a default course
78      }
79      else {
80        studentStatement.setString(1, choice); // Set course title         set a course
81      }
82
83      // Get students for the specified course
84      return studentStatement.executeQuery();                              return students
85    }
86  }
```

We use the same MySQL database **javabook** created in Chapter 34, "Java Database Programming." The scope for this managed bean is **application**. The bean is created when the project is launched from the server. The `initializeJdbc` method loads the JDBC driver for MySQL (lines 23 and 24), connects to the MySQL database (lines 27 and 28), creates statement for obtaining course titles (lines 31 and 32), and creates a statement for obtaining the student information for the specified course (lines 45–53). Lines 31–42 execute the statement for obtaining course titles and store them in array `titles`.

The `getStudents()` method returns a `ResultSet` that consists of all students enrolled in the specified course (lines 72–85). The choice for the title is set in the statement to obtain the

student for the specified title (line 80). If choice is `null`, the first title in the titles array is set in the statement (line 77). If no titles in the course, `getStudents()` returns `null` (line 75).

add MySQL in the Libraries node

TIP

In order to use the MySQL database from this project, you have to add the MySQL JDBC driver from the Libraries node in the Project pane in NetBeans.

LISTING 39.14 DisplayStudent.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html"
6        xmlns:f="http://xmlns.jcp.org/jsf/core">
7    <h:head>
8      <title>Display Student</title>
9      <h:outputStylesheet name="tablestyle.css"/>
10   </h:head>
11   <h:body>
12     <h:form>
13       <h:outputLabel value="Choose a Course: " />
14       <h:selectOneMenu value="#{courseName.choice}">
15         <f:selectItems value="#{courseName.titles}" />
16       </h:selectOneMenu>
17
18       <h:commandButton style="margin-left: 20px"
19                        value="Display Students" />
20
21       <br /> <br />
22       <h:dataTable value="#{courseName.students}" var="student"
23                    rowClasses="oddTableRow, evenTableRow"
24                    headerClass="tableHeader"
25                    styleClass="table">
26         <h:column>
27           <f:facet name="header">SSN</f:facet>
28           #{student.ssn}
29         </h:column>
30
31         <h:column>
32           <f:facet name="header">First Name</f:facet>
33           #{student.firstName}
34         </h:column>
35
36         <h:column>
37           <f:facet name="header">MI</f:facet>
38           #{student.mi}
39         </h:column>
40
41         <h:column>
42           <f:facet name="header">Last Name</f:facet>
43           #{student.lastName}
44         </h:column>
45
46         <h:column>
47           <f:facet name="header">Phone</f:facet>
48           #{student.phone}
49         </h:column>
50
51         <h:column>
```

style sheet

bind choice titles

display button

bind result set rowClasses headerClass styleClass

ssn column

firstName column

mi column

lastName column

phone column

```
52            <f:facet name="header">Birth Date</f:facet>
53            #{student.birthDate}                                    birthDate column
54          </h:column>
55
56          <h:column>
57            <f:facet name="header">Dept</f:facet>
58            #{student.deptId}                                       deptId column
59          </h:column>
60        </h:dataTable>
61      </h:form>
62    </h:body>
63  </html>
```

Line 9 specifies that the style sheet tablestyle.css created in Step 3 is used in this XMTHL file. The `rowClasses = "oddTableRow, evenTableRow"` attribute specifies the style applied to the rows alternately using `oddTableRow` and `evenTableRow` (line 23). The `headerClasses = "tableHeader"` attribute specifies that the `tableHeader` class is used for header style (line 24). The `styleClasses = "table"` attribute specifies that the `table` class is used for the style of all other elements in the table (line 25).

Line 14 binds the `choice` property in the `courseName` bean with the combo box. The selection values in the combo box are bound with the `titles` array property (line 15).

Line 22 binds the table value with a database result set using the attribute `value="#{courseName.students}"`. The `var="student"` attribute associates a row in the result set with `student`. Lines 26–59 specify the column values using `student.ssn` (line 28), `student.firstName` (line 33), `student.mi` (line 38), `student.lastName` (line 33), `student.phone` (line 48), `student.birthDate` (line 53), and `student.deptId` (line 58).

LISTING 39.15 tablestyle.css

```
1  /* Style for table */
2  .tableHeader {                                                     tableHeader
3    font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
4    border-collapse:collapse;
5    font-size:1.1em;
6    text-align:left;
7    padding-top:5px;
8    padding-bottom:4px;
9    background-color:#A7C942;
10   color:white;
11   border:1px solid #98bf21;
12 }
13
14 .oddTableRow {                                                     oddTableRow
15   border:1px solid #98bf21;
16 }
17
18 .evenTableRow {                                                    evenTableRow
19   background-color: #eeeeee;
20   font-size:1em;
21
22   padding:3px 7px 2px 7px;
23
24   color:#000000;
25   background-color:#EAF2D3;
26 }
27
28 .table {                                                           table
29   border:1px solid green;
30 }
```

The style sheet file defines the style classes `tableHeader` (line 2) for table header style, `oddTableRow` for odd table rows (line 14), `evenTableRow` for even table rows (line 18), and table for all other table elements (line 28).

39.9 Opening New JSF Pages

You can open new JSF pages from the current JSF pages.

All the examples you have seen so far use only one JSF page in a project. Suppose you want to register student information to the database. The application first displays the page as shown in Figures 39.22 to collect student information. After the user enters the information and clicks the *Submit* button, a new page is displayed to ask the user to confirm the input, as shown in Figure 39.23. If the user clicks the *Confirm* button, the data are stored into the database and the status page is displayed, as shown in Figure 39.24. If the user clicks the *Go Back* button, it goes back to the first page.

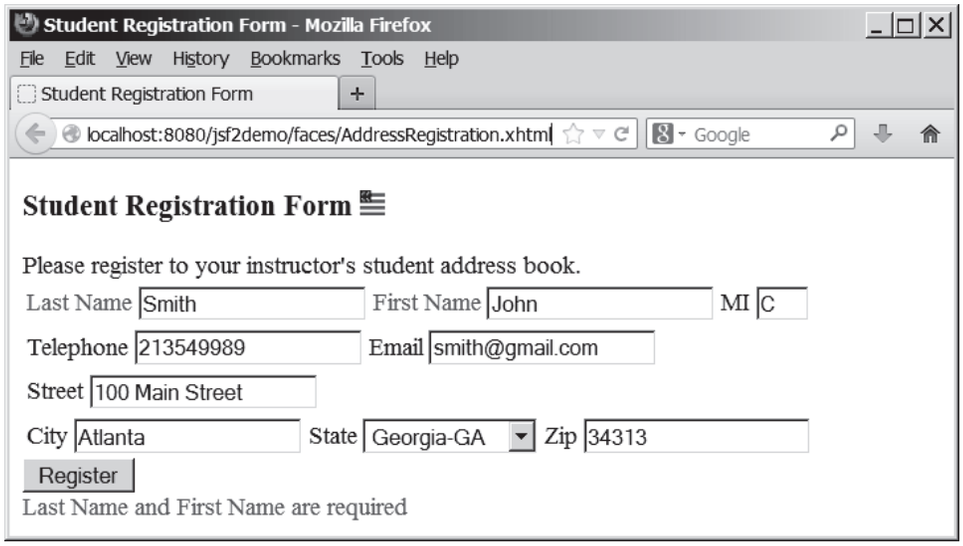


FIGURE 39.22 This page lets the user enter input.

For this project, you need to create three JSF pages named AddressRegistration.xhtml, ConfirmAddress.xhtml, and AddressStoredStatus.xhtml in Listings 39.16–39.18. The project starts with AddressRegistration.xhtml. When clicking the *Submit* button, the action for the button returns "ConfirmAddress" if the last name and first name are not empty, which causes ConfirmAddress.xhtml to be displayed. When clicking the *Confirm* button, the status page AddressStoredStatus is displayed. When clicking the *Go Back* button, the first page Address-Registration is now displayed.

LISTING 39.16 AddressRegistration.xhtml

jsf core namespace

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html"
6        xmlns:f="http://xmlns.jcp.org/jsf/core">
7    <h:head>
8      <title>Student Registration Form</title>
```

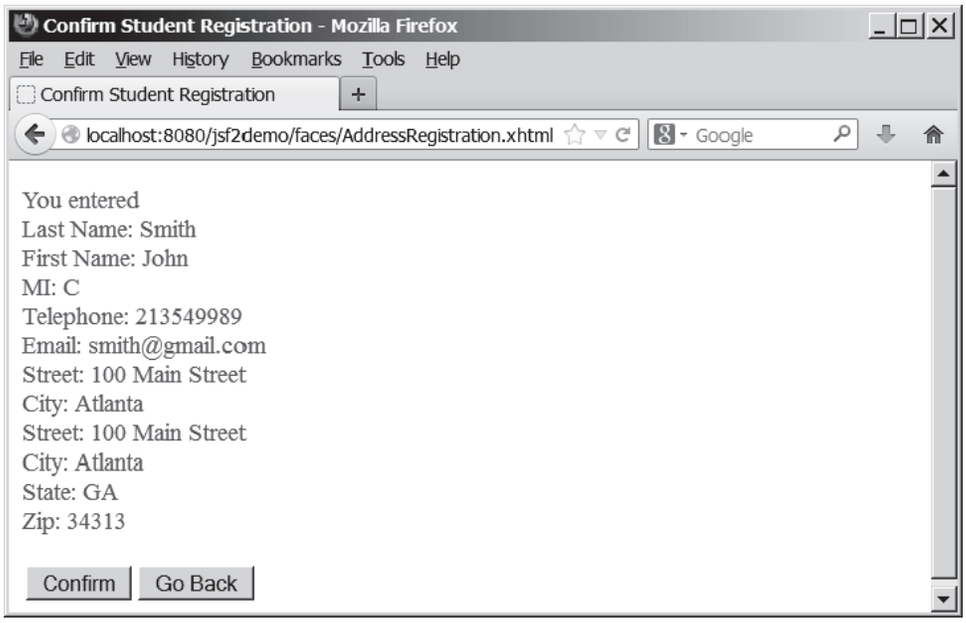


FIGURE 39.23 This page lets the user confirm the input.

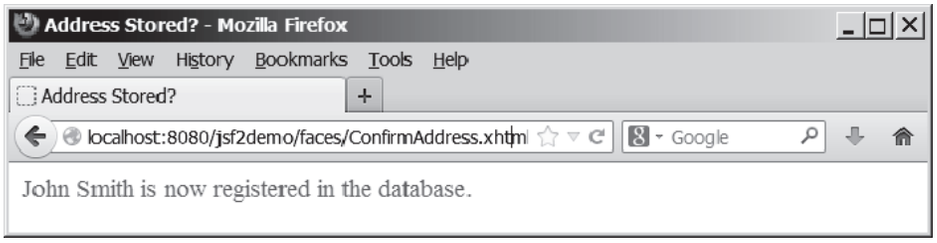


FIGURE 39.24 This page displays the status of the user input.

```
 9    </h:head>
10    <h:body>
11      <h:form>
12        <!-- Use h:graphicImage -->
13        <h3>Student Registration Form
14          <h:graphicImage name="usIcon.gif" library="image"/>
15        </h3>
16
17        Please register to your instructor's student address book.
18        <!-- Use h:panelGrid -->
19        <h:panelGrid columns="6">
20          <h:outputLabel value="Last Name" style="color:red"/>
21          <h:inputText id="lastNameInputText"
22                       value="#{addressRegistration.lastName}"/>          bind lastName
23          <h:outputLabel value="First Name" style="color:red"/>
24          <h:inputText id="firstNameInputText"
25                       value="#{addressRegistration.firstName}"/>         bind firstName
26          <h:outputLabel value="MI" />
27          <h:inputText id="miInputText" size="1"
28                       value="#{addressRegistration.mi}"/>                bind mi
29        </h:panelGrid>
30
```

```
31        <h:panelGrid columns="4">
32          <h:outputLabel value="Telephone"/>
33          <h:inputText id="telephoneInputText"
34                       value="#{addressRegistration.telephone}"/>
35          <h:outputLabel value="Email"/>
36          <h:inputText id="emailInputText"
37                       value="#{addressRegistration.email}"/>
38        </h:panelGrid>
39
40        <h:panelGrid columns="4">
41          <h:outputLabel value="Street"/>
42          <h:inputText id="streetInputText"
43                       value="#{addressRegistration.street}"/>
44        </h:panelGrid>
45
46        <h:panelGrid columns="6">
47          <h:outputLabel value="City"/>
48          <h:inputText id="cityInputText"
49                       value="#{addressRegistration.city}"/>
50          <h:outputLabel value="State"/>
51          <h:selectOneMenu id="stateSelectOneMenu"
52                           value="#{addressRegistration.state}">
53            <f:selectItem itemLabel="Georgia-GA" itemValue="GA" />
54            <f:selectItem itemLabel="Oklahoma-OK" itemValue="OK" />
55            <f:selectItem itemLabel="Indiana-IN" itemValue="IN"/>
56          </h:selectOneMenu>
57          <h:outputLabel value="Zip"/>
58          <h:inputText id="zipInputText"
59                       value="#{addressRegistration.zip}"/>
60        </h:panelGrid>
61
62        <!-- Use command button -->
63        <h:commandButton value="Register"
64                         action="#{addressRegistration.processSubmit()}"/>
65        <br />
66        <h:outputText escape="false" style="color:red"
67                      value="#{addressRegistration.requiredFields}" />
68      </h:form>
69    </h:body>
70  </html>
```

bind telephone
bind email
bind street
bind city
bind state
bind zip
process register

LISTING 39.17 ConfirmAddress.xhtml

```
1   <?xml version='1.0' encoding='UTF-8' ?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4   <html xmlns="http://www.w3.org/1999/xhtml"
5         xmlns:h="http://xmlns.jcp.org/jsf/html">
6     <h:head>
7       <title>Confirm Student Registration</title>
8     </h:head>
9     <h:body>
10      <h:form>
11        <h:outputText escape="false" style="color:red"
12                      value="#{registration1.input}" />
13        <h:panelGrid columns="2">
14        <h:commandButton value="Confirm"
15           action = "#{registration1.storeStudent()}"/>
16        <h:commandButton value="Go Back"
17           action = "AddressRegistration"/>
```

process confirm
go to AddressRegistration page

```
18        </h:panelGrid>
19      </h:form>
20    </h:body>
21  </html>
```

LISTING 39.18 AddressStoredStatus.xhtml

```
 1  <?xml version='1.0' encoding='UTF-8' ?>
 2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4  <html xmlns="http://www.w3.org/1999/xhtml"
 5        xmlns:h="http://xmlns.jcp.org/jsf/html">
 6    <h:head>
 7      <title>Address Stored?</title>
 8    </h:head>
 9    <h:body>
10      <h:form>
11        <h:outputText escape="false" style="color:green"
12                      value="#{registration1.status}" />                display status
13      </h:form>
14    </h:body>
15  </html>
```

LISTING 39.19 AddressRegistrationJSFBean.java

```
 1  package jsf2demo;
 2
 3  import javax.inject.Named;
 4  import javax.enterprise.context.SessionScoped;
 5  import java.sql.*;
 6  import java.io.Serializable;
 7
 8  @Named(value = "addressRegistration")                                  managed bean
 9  @SessionScoped                                                         session scope
10  public class AddressRegistrationJSFBean implements Serializable {
11    private String lastName;                                             property lastName
12    private String firstName;
13    private String mi;
14    private String telephone;
15    private String email;
16    private String street;
17    private String city;
18    private String state;
19    private String zip;
20    private String status = "Nothing stored";
21    // Use a prepared statement to store a student into the database
22    private PreparedStatement pstmt;
23
24    public AddressRegistrationJSFBean() {
25      initializeJdbc();                                                  initialize database
26    }
27
28    public String getLastName() {
29      return lastName;
30    }
31
32    public void setLastName(String lastName) {
33      this.lastName = lastName;
34    }
35
```

```
36    public String getFirstName() {
37      return firstName;
38    }
39
40    public void setFirstName(String firstName) {
41      this.firstName = firstName;
42    }
43
44    public String getMi() {
45      return mi;
46    }
47
48    public void setMi(String mi) {
49      this.mi = mi;
50    }
51
52    public String getTelephone() {
53      return telephone;
54    }
55
56    public void setTelephone(String telephone) {
57      this.telephone = telephone;
58    }
59
60    public String getEmail() {
61      return email;
62    }
63
64    public void setEmail(String email) {
65      this.email = email;
66    }
67
68    public String getStreet() {
69      return street;
70    }
71
72    public void setStreet(String street) {
73      this.street = street;
74    }
75
76    public String getCity() {
77      return city;
78    }
79
80    public void setCity(String city) {
81      this.city = city;
82    }
83
84    public String getState() {
85      return state;
86    }
87
88    public void setState(String state) {
89      this.state = state;
90    }
91
92    public String getZip() {
93      return zip;
94    }
95
96    public void setZip(String zip) {
```

```
97      this.zip = zip;
98    }
99
100   private boolean isRquiredFieldsFilled() {
101     return !(lastName == null || firstName == null
102           || lastName.trim().length() == 0
103           || firstName.trim().length() == 0);
104   }
105
106   public String processSubmit() {
107     if (isRquiredFieldsFilled())
108       return "ConfirmAddress";                          go to a new page
109     else
110       return "";
111   }
112
113   public String getRequiredFields() {
114     if (isRquiredFieldsFilled())                        check required fields
115       return "";
116     else
117       return "Last Name and First Name are required";
118   }
119
120   public String getInput() {                            get input
121     return "<p style=\"color:red\">You entered <br />"
122             + "Last Name: " + lastName + "<br />"
123             + "First Name: " + firstName + "<br />"
124             + "MI: " + mi + "<br />"
125             + "Telephone: " + telephone + "<br />"
126             + "Email: " + email + "<br />"
127             + "Street: " + street + "<br />"
128             + "City: " + city + "<br />"
129             + "Street: " + street + "<br />"
130             + "City: " + city + "<br />"
131             + "State: " + state + "<br />"
132             + "Zip: " + zip + "</p>";
133   }
134
135   /** Initialize database connection */
136   private void initializeJdbc() {
137     try {
138       // Explicitly load a MySQL driver
139       Class.forName("com.mysql.jdbc.Driver");
140       System.out.println("Driver loaded");
141
142       // Establish a connection
143       Connection conn = DriverManager.getConnection(
144             "jdbc:mysql://localhost/javabook", "scott", "tiger");
145
146       // Create a Statement
147       pstmt = conn.prepareStatement("insert into Address (lastName,"
148             + " firstName, mi, telephone, email, street, city, "
149             + "state, zip) values (?, ?, ?, ?, ?, ?, ?, ?, ?)");
150     }
151     catch (Exception ex) {
152       System.out.println(ex);
153     }
154   }
155
156   /** Store an address to the database */
157   public String storeStudent() {                        store address
```

```
158       try {
159         pstmt.setString(1, lastName);
160         pstmt.setString(2, firstName);
161         pstmt.setString(3, mi);
162         pstmt.setString(4, telephone);
163         pstmt.setString(5, email);
164         pstmt.setString(6, street);
165         pstmt.setString(7, city);
166         pstmt.setString(8, state);
167         pstmt.setString(9, zip);
168         pstmt.executeUpdate();
169         status = firstName + " " + lastName
170                 + " is now registered in the database.";
171       }
172       catch (Exception ex) {
173         status = ex.getMessage();
174       }
175
176       return "AddressStoredStatus";
177     }
178
179     public String getStatus() {
180       return status;
181     }
182   }
```

update status

go to a new page

A session-scoped managed bean must implement the `java.io.Serializable` interface. So, the `AddressRegistration` class is defined as a subtype of `java.io.Serializable`.

The action for the *Register* button in the `AddressRegistration` JSF page is `processSubmit()` (line 64 in AddressRegistration.xhtml). This method checks if last name and first name are not empty (lines 106–111 in AddressRegistrationJSFBean.java). If so, it returns a string `"ConfirmAddress"`, which causes the `ConfirmAddress` JSF page to be displayed.

The `ConfirmAddress` JSF page displays the data entered from the user (line 12 in ConfirmAddress.xhtml). The `getInput()` method (lines 120–133 in AddressRegistrationJSFBean.java) collects the input.

The action for the *Confirm* button in the `ConfirmAddress` JSF page is `storeStudent()` (line 15 in ConfirmAddress.xhtml). This method stores the address in the database (lines 157–177 in AddressRegistrationJSFBean.java) and returns a string `"AddressStoredStatus"`, which causes the `AddressStoredStatus` page to be displayed. The status message is displayed in this page (line 12 in AddressStoredStatus.xhtml).

The action for the *Go Back* button in the `ConfirmAddress` page is `"AddressRegistration"` (line 17 in ConfirmAddress.xhtml). This causes the `AddressRegistration` page to be displayed for the user to reenter the input.

The scope of the managed bean is session (line 9 AddressRegistrationJSFBean.java) so the multiple pages can share the same bean.

Note that this program loads the database driver explicitly (line 139 AddressRegistrationJSFBean.java). Sometimes, an IDE such as NetBeans is not able to find a suitable driver. Loading a driver explicitly can avoid this problem.

39.10 Contexts and Dependency Injection

Contexts and dependency injection enables beans to be shared in multiple applications.

Contexts and dependency injection, short for *CDI*, allows multiple programs to share a bean. To illustrate the need for this, consider two simple Web pages and a server object named `track`. One page contains a button and a message that displays the number of times the button is clicked from the current IP address, as shown in Figure 39.25. When the button is clicked

for the first time, the user's IP address along with count value 1 is stored in a map with the IP address as the key. When the button is clicked again, the count value for the IP address is increased in the map. The other page simply displays the total count from each IP address, as shown in Figure 39.26. The `Track` class is defined as shown in Listing 39.20.

FIGURE 39.25 The count is updated when the Click Me button is clicked.

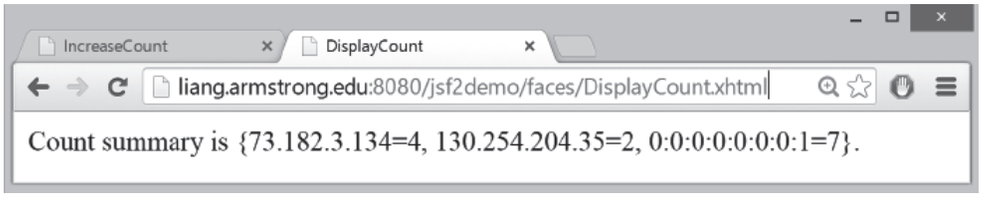


FIGURE 39.26 The count for each client IP Address is displayed.

LISTING 39.20 Track.java

```
1  package jsf2demo;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import javax.enterprise.context.ApplicationScoped;
6
7  @ApplicationScoped                                          application scope
8  public class Track {
9    private Map<String, Integer> map = new HashMap<>();       store counts
10
11   public void add(String ipAddress) {
12     map.put(ipAddress, map.containsKey(ipAddress) ?         add or update count
13       map.get(ipAddress) + 1 : 1);
14   }
15
16   public int getCount(String ipAddress) {                   return count
17     return map.containsKey(ipAddress) ? map.get(ipAddress) : 0;
18   }
19
20   public String getAllCount() {                             return all counts
21     return "Count summary is " + map;
22   }
23 }
```

A `Track` object uses a map to store an IP address and its count with IP address as a key (line 9). The `add` method (lines 11–14) adds an IP Address to the map. If the IP address is not in the map, a new entry is created for the IP Address with value `1`. Otherwise, the value for the IP address is incremented by `1` in the map. The `getCount` method (lines 16–18) returns the count for an IP address. If the IP address is not in the map, the method returns `0`. The

`getAllCount` method (lines 20–22) simply returns a string that describes the counts for all IP address in the map.

We now create a page named **IncreaseCount.xhtml** (Listing 39.21) with a button for displaying the number of times a button is clicked on the client, and create a page named **DisplayCount.xhtml** (Listing 39.22) for displaying the counts from all clients.

LISTING 39.21 IncreaseCount.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html">
6    <h:head>
7      <title>IncreaseCount</title>
8    </h:head>
9    <h:body>
10     <h:form>
11       <h:commandButton
12         action="#{increaseCount.click()}" value="Click Me"/>
13       <br>The current count is #{increaseCount.getCount()} and your
14         IP address is #{increaseCount.getIpAddress()}.</br>
15     </h:form>
16   </h:body>
17 </html>
```

process a click
obtain count
obtain IP address

LISTING 39.22 DisplayCount.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html">
6    <h:head>
7      <title>DisplayCount</title>
8    </h:head>
9    <h:body>
10     #{displayCount.getAllCount()}.
11   </h:body>
12 </html>
```

obtain all counts

The `IncreaseCount` page uses the `increasCount` bean to process the click action (line 12), obtain the click count (line 13), and the client's IP address (line 14). The `DisplayCount` page uses the `displayCount` bean to obtain the count from all clients (line 10). Both `increasCount` and `displayCount` need to access the same `track` object. How can you create a `Track` object to be used by different objects? JSF supports context dependency injection (CDI) for injecting an object into a class using the `@Inject` annotation. Listing 39.23 gives the implementation for IncreaseCount.java and Listing 39.24 for DisplayCount.java.

LISTING 39.23 IncreaseCount.java

```
1  package jsf2demo;
2
3  import javax.enterprise.context.SessionScoped;
4  import javax.inject.Named;
5  import javax.faces.context.FacesContext;
6  import javax.inject.Inject;
7  import javax.servlet.http.HttpServletRequest;
8
9  @Named(value = "increaseCount")
```

```
10  @SessionScoped                                                    session scope
11  public class IncreaseCount implements java.io.Serializable {
12    @Inject private Track track;                                    inject track
13    private String ipAddress;
14
15    public IncreaseCount() {                                        increase count
16      HttpServletRequest request = (HttpServletRequest)FacesContext obtain client's IP
17        .getCurrentInstance().getExternalContext().getRequest();
18      this.ipAddress = request.getRemoteAddr();
19    }
20
21    public void click() {
22      track.add(ipAddress);                                         add an IP address
23    }
24
25    public String getIpAddress() {
26      return ipAddress;
27    }
28
29    public int getCount() {
30      return track.getCount(ipAddress);                             count for an IP
31    }
32  }
```

Listing 39.24 DisplayCount.java

```
1   package jsf2demo;
2
3   import javax.enterprise.context.ApplicationScoped;
4   import javax.inject.Named;
5   import javax.inject.Inject;
6
7   @Named(value = "displayCount")
8   @ApplicationScoped                                                application scope
9   public class DisplayCount {
10    @Inject private Track track;                                    inject track
11
12    public String getAllCount() {                                   obtain all counts
13      return track.getAllCount();
14    }
15  }
```

The `@Inject` annotation in line 12 of **IncreaseCount.java** and line 10 of **DisplayCount.java** injects a `Track` object. This `Track` object is created by the Java server container. The `track` data fields in both classes refer to this object.

In **IncreaseCount.java**, the constructor obtains the IP address of a client (lines 16 and 17) and sets it in the data field `ipAddress` (line 18). The `click` method adds the `ipAddress` to the map in the track object (line 22).

Note that the scope for `Track` and `DisplayCount` is `ApplicationScoped` since these two objects are created once for the entire application. However, the scope for `IncreaseCount` is `SessionScoped` since each session has its own IP Address.

Key Terms

application scope 39-21
contexts and dependency injection (CDI) 39-40
JavaBean 39-5
request scope 39-21
scope 39-21
session scope 39-21
view scope 39-21

Chapter Summary

1. JSF enables you to completely separate Java code from HTML.

2. A `facelet` is an XHTML page that mixes JSF tags with XHTML tags.

3. JSF applications are developed using the Model-View-Controller (MVC) architecture, which separates the application's data (contained in the model) from the graphical presentation (the view).

4. The controller is the JSF framework that is responsible for coordinating interactions between view and the model.

5. In JSF, the facelets are the view for presenting data. Data are obtained from Java objects. Objects are defined using Java classes.

6. In JSF, the objects that are accessed from a facelet are JavaBeans objects.

7. The JSF expression can either use the property name or invoke the method to obtain the current time.

8. JSF provides many elements for displaying GUI components. The tags with the `h` prefix are in the JSF HTML Tag library. The tags with the `f` prefix are in the JSF Core Tag library.

9. You can specify the JavaBeans objects at the application scope, session scope, view scope, or request scope.

10. The view scope keeps the bean alive as long as you stay on the view. The view scope is between session and request scopes.

11. JSF provides several convenient and powerful ways for input validation. You can use the standard validator tags in the JSF Core Tag Library or create custom validators.

Quiz

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

MyProgrammingLab™

Programming Exercises

***39.1** *(Factorial table in JSF)* Write a JSF page that displays a factorial page as shown in Figure 39.27. Display the table in an `h:outputText` component. Set its `escape` property to `false` to display it as HTML contents.

***39.2** (*Multiplication table*) Write a JSF page that displays a multiplication table as shown in Figure 39.28.

***39.3** (*Calculate tax*) Write a JSF page to let the user enter taxable income and filing status, as shown in Figure 39.29a. Clicking the *Compute Tax* button computes and displays the tax, as shown in Figure 39.29b. Use the `computeTax` method introduced in Listing 3.5, ComputeTax.java, to compute tax.

***39.4** (*Calculate loan*) Write a JSF page that lets the user enter loan amount, interest rate, and number of years, as shown in Figure 39.30a. Click the *Compute Loan Payment* button to compute and display the monthly and total loan payments, as shown in Figure 39.30b. Use the `Loan` class given in Listing 10.2, Loan.java, to compute the monthly and total payments.

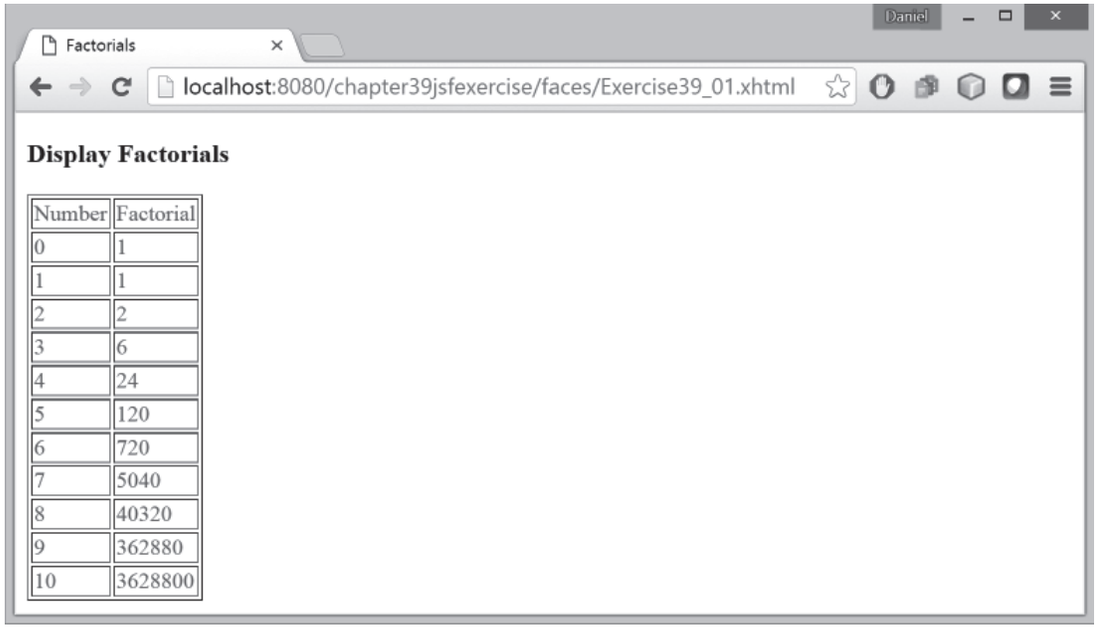


FIGURE 39.27 The JSF page displays factorials for the numbers from 0 to 10 in a table.

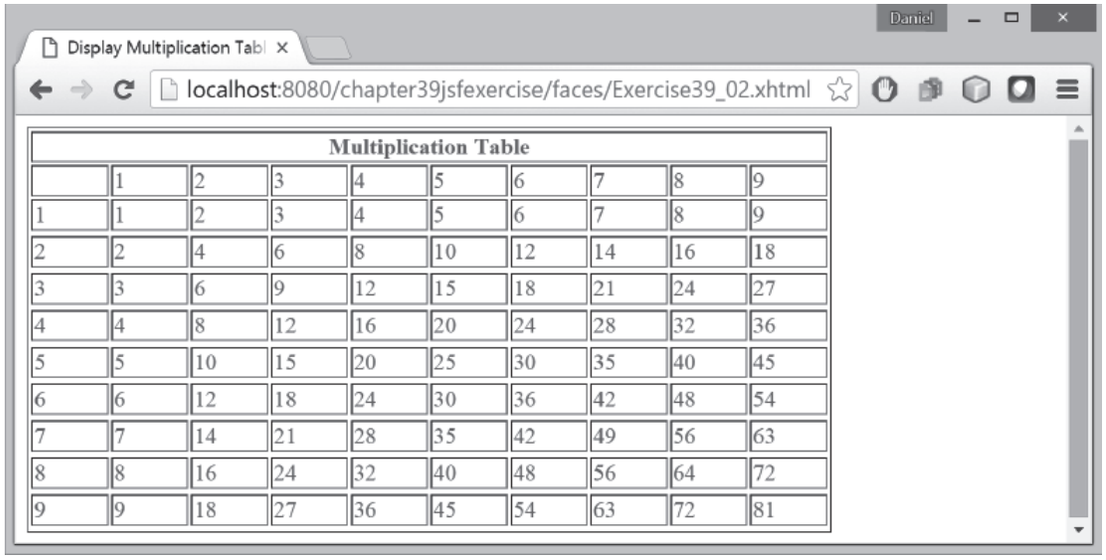
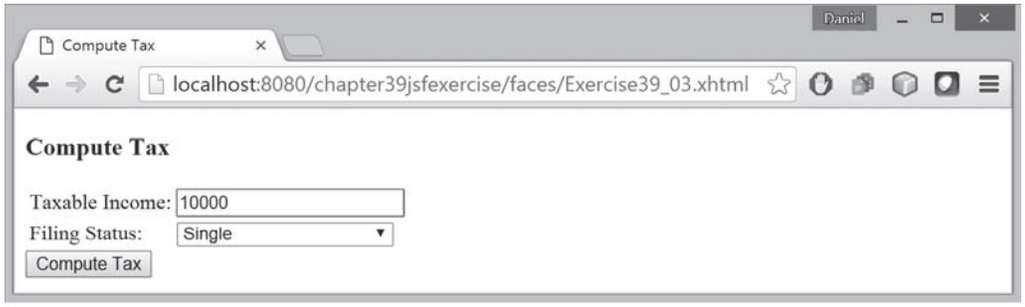


FIGURE 39.28 The JSF page displays the multiplication table.

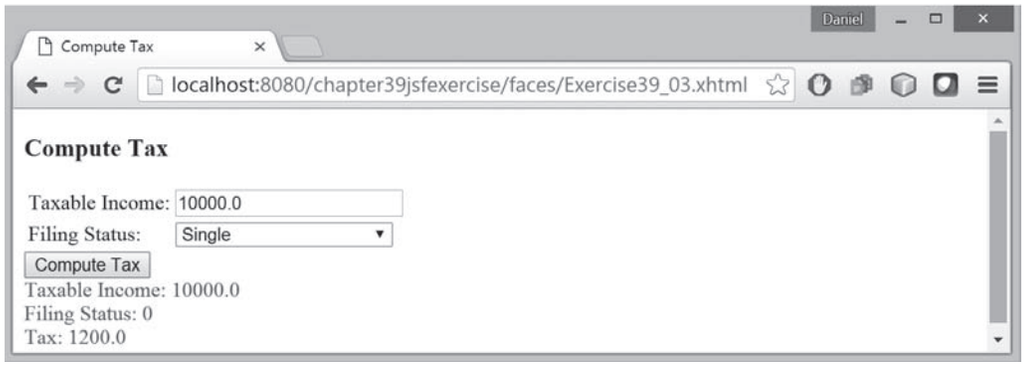
***39.5** (*Addition quiz*) Write a JSF program that generates addition quizzes randomly, as shown in Figure 39.31a. After the user answers all questions, it displays the result, as shown in Figure 39.31b.

***39.6** (*Large factorial*) Rewrite Exercise 39.1 to handle large factorial as shown in Figure 39.32. Use the `BigInteger` class introduced in Section 10.9.

***39.7** (*Guess birthday*) Listing 4.3, GuessBirthday.java, gives a program for guessing a birthday. Write a JSF program that displays five sets of numbers, as shown in Figure 39.33a. After the user checks the appropriate boxes and clicks the *Guess Birthday* button, the program displays the birthday, as shown in Figure 39.33b.

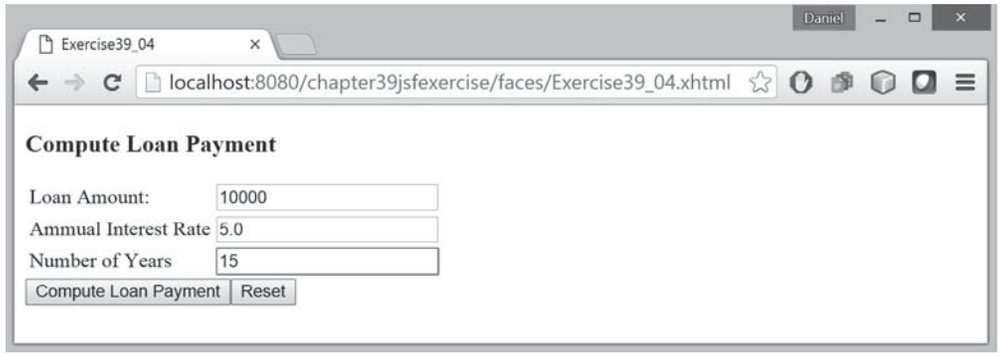


(a)



(b)

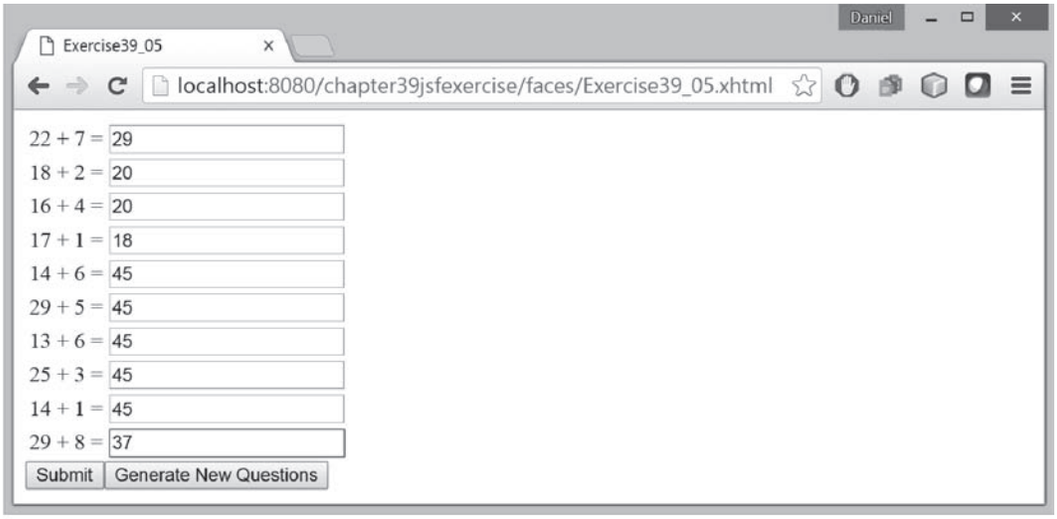
FIGURE 39.29 The JSF page computes the tax.



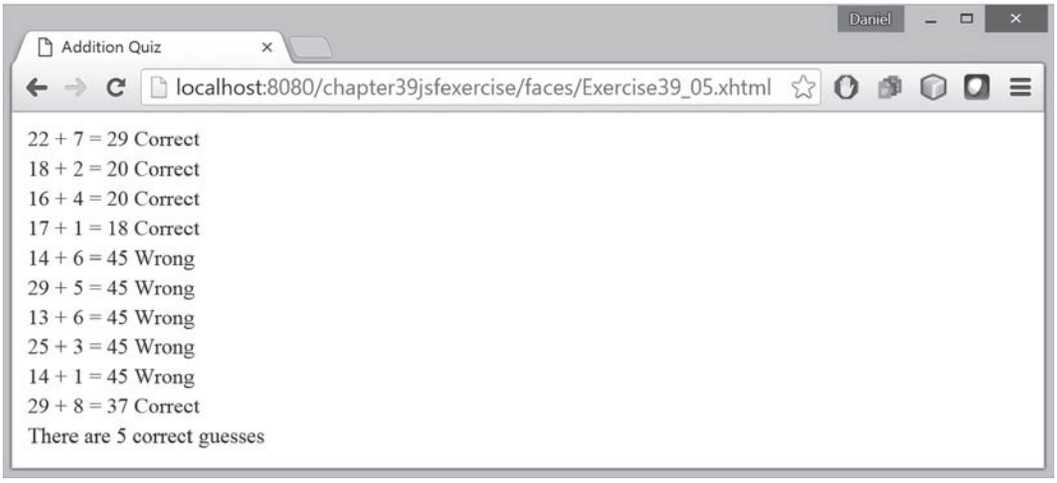
(a)

(b)

FIGURE 39.30 The JSF page computes the loan payment.



(a)



(b)

FIGURE 39.31 The program displays addition questions in (a) and answers in (b).

***39.8** (*Guess capitals*) Write a JSF that prompts the user to enter a capital for a state, as shown in Figure 39.34a. Upon receiving the user input, the program reports whether the answer is correct, as shown in Figure 39.34b. 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 8.37. Create a list from the array and apply the `shuffle` method to reorder the list so the questions will appear in random order.

***39.9** (*Access and update a `Staff` table*) Write a JSF program that views, inserts, and updates staff information stored in a database, as shown in Figure 39.35. The view button displays a record with a specified ID. The `Staff` table is created as follows:

```
create table Staff (
  id char(9) not null,
  lastName varchar(15),
  firstName varchar(15),
```

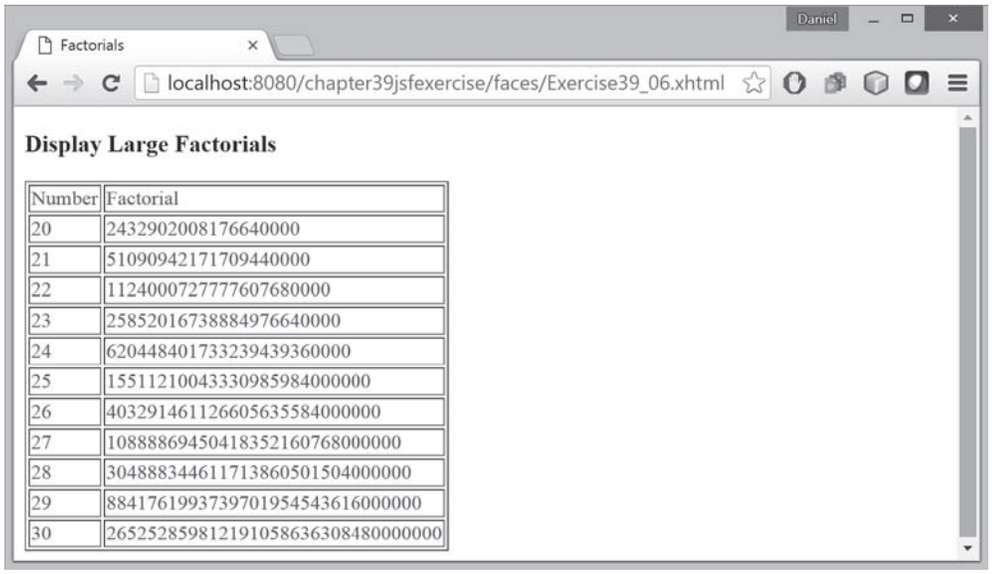
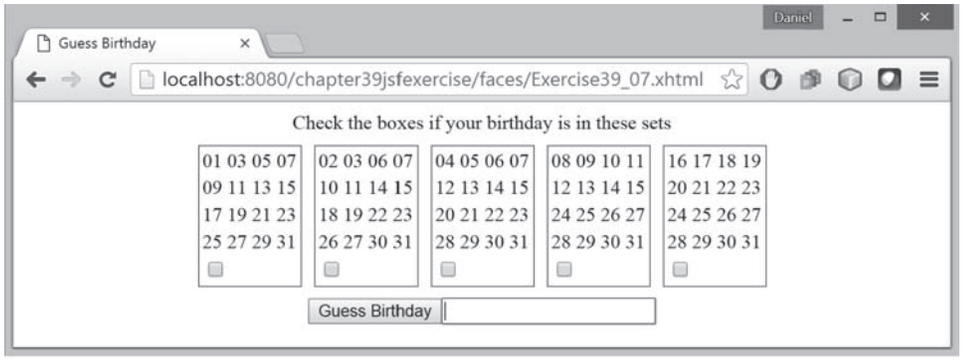
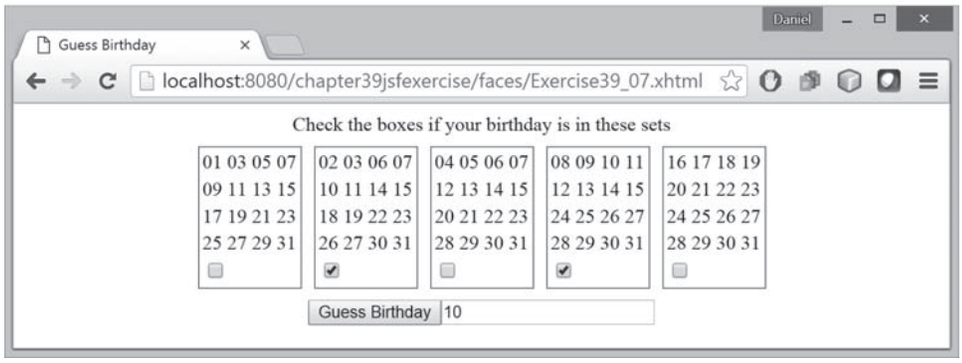


FIGURE 39.32 The JSF page displays factorials for the numbers from 10 to 20 in a table.



(a)



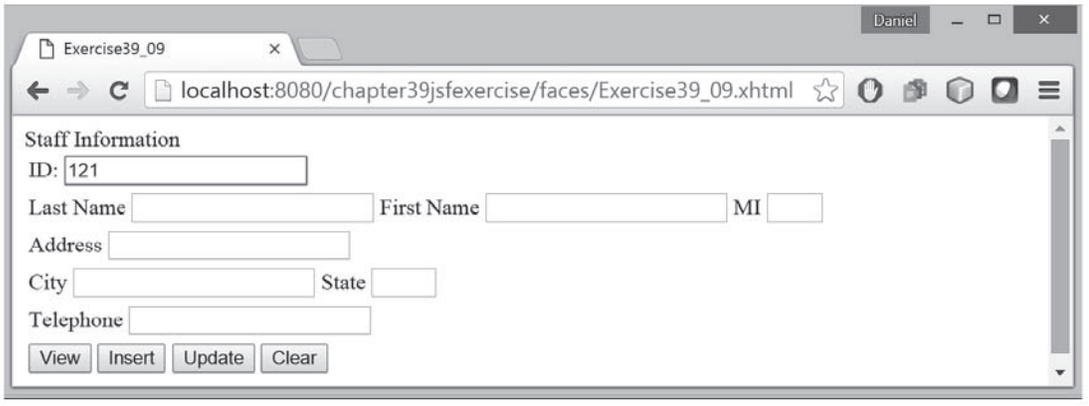
(b)

FIGURE 39.33 (a) The program displays five sets of numbers for the user to check the boxes. (b) The program displays the date.

(a)

(b)

FIGURE 39.34 (a) The program displays a question. (b) The program displays the answer to the question.



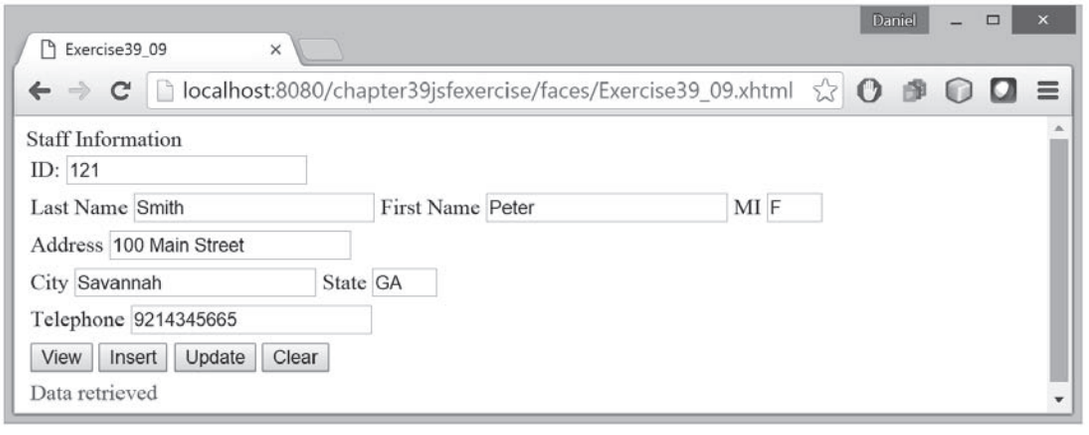


FIGURE 39.35 The web page lets you view, insert, and update staff information.

```
  mi char(1),
  address varchar(20),
  city varchar(20),
  state char(2),
  telephone char(10),
  email varchar(40),
  primary key (id)
);
```

***39.10** (*Random cards*) Write a JSF that displays four random cards from a deck of 52 cards, as shown in Figure 39.36. When the user clicks the *Refresh* button, four new random cards are displayed.

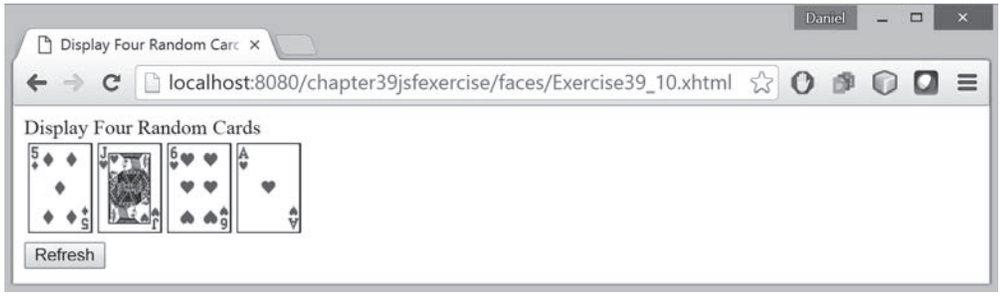


FIGURE 39.36 This JSF application displays four random cards.

*****39.11** (*Game: the 24-point card game*) Rewrite Exercise 20.13 using JSF, as shown in Figure 39.37. Upon clicking the *Refresh* button, the program displays four random cards and displays an expression if a 24-point solution exists. Otherwise, it displays `No solution`.

FIGURE 39.37 The JSF application solves a 24-Point card game.

***39.12 (*Game: the 24-point card game*) Rewrite Exercise 20.17 using JSF, as shown in Figure 39.38. The program lets the user enter four card values and finds a solution upon clicking the *Find a Solution* button.

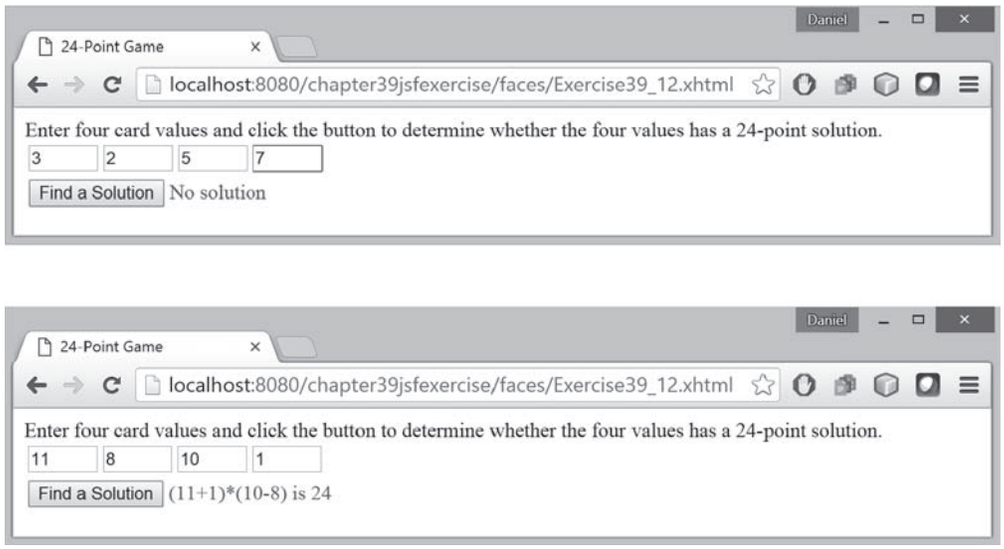


FIGURE 39.38 The user enters four numbers and the program finds a solution.

*39.13 (*Day of week*) Write a program that displays the day of the week for a given day, month, and year, as shown in Figure 39.39. The program lets the user select a day, month, and year, and click the *Get Day of Week* button to display the day of week. The Time field displays Future if it is a future day or Past otherwise. Use the Zeller's congruence to find the day of the week (see Programming Exercise 3.21).

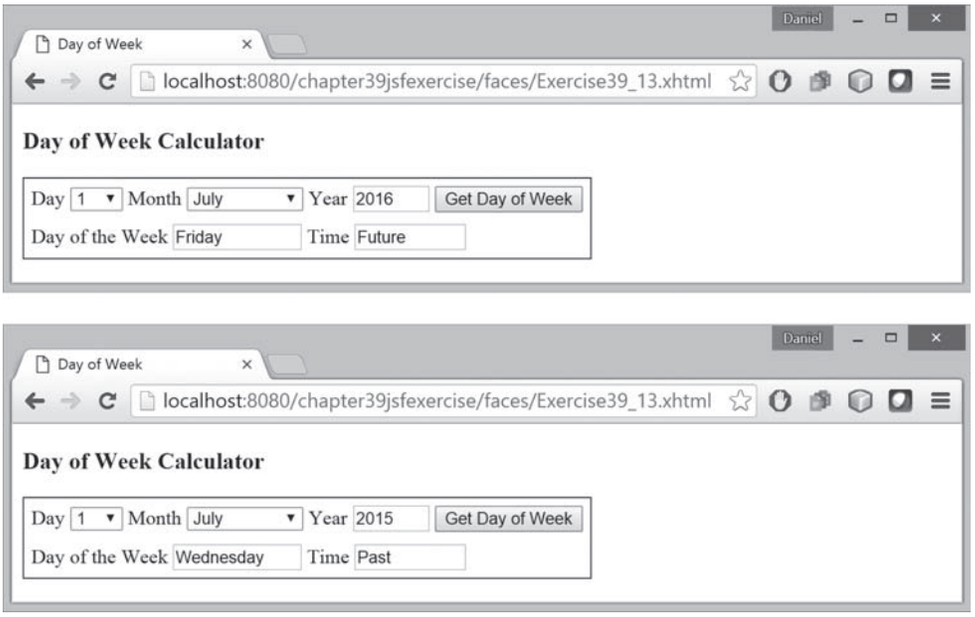


FIGURE 39.39 The user enters a day, month, and year and the program finds the day of the week.

***39.14** (*Display total count*) Revise Listing 39.22 DisplayCount.xhmt to display the total count of the button clicks form all clients and display the client's IP address and counts in increasing order of the counts, as shown in Figure 39.40.

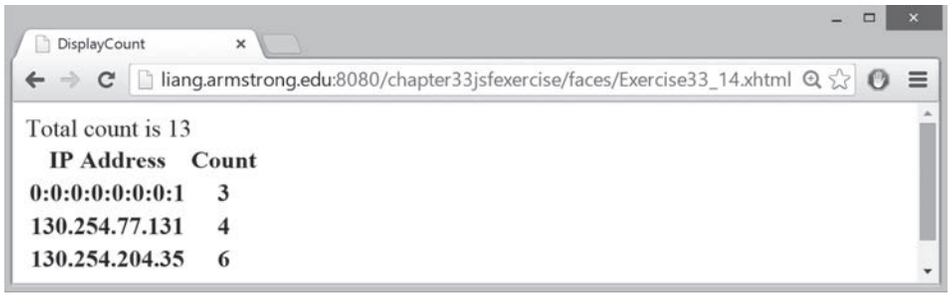


FIGURE 39.40 The total counts and individual client counts are displayed.