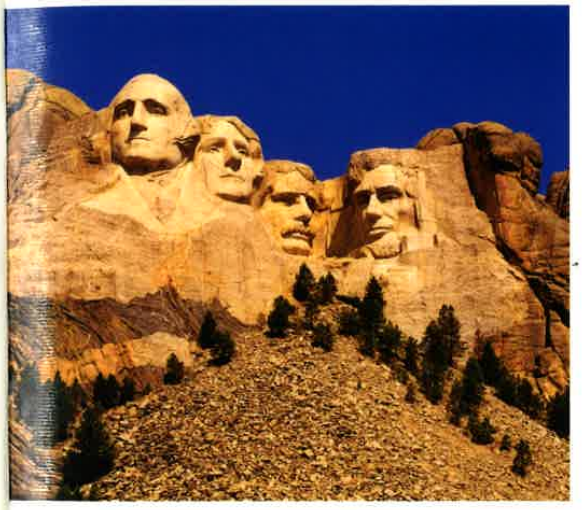
core JavaServer™ Faces



- Features the expertise of Expert Group Member David Geary and the unique insights of Cay Horstmann, the best-selling author of *Core Java*
- Explains exactly how to get things done with the JavaServer Faces 1.0 framework from the developer's point of view—practical workarounds, no punting
- Covers integration with Tiles, databases, wireless services, and more—with real working examples, no toy code

Sun microsystems

DAVID GEARY · CAY HORSTMANN

Java™ 2 Platform, Enterprise Edition Series

core JavaServer™ Faces

DAVID GEARY
CAY HORSTMANN

PRENTICE HALL PTR

Prentice Hall PTR, Upper Saddle River, NJ 07458
www.phptr.com

Sun Microsystems Press
A Prentice Hall Title

Prentice Hall PTR offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact U.S. Corporate and Government Sales, 1-800-382-3419, corpsales@pearsontechgroup.com. For sales outside of the U.S., please contact International Sales, 1-317-581-3793, international@pearsontechgroup.com.

Acquisitions Editor: *Gregory G. Doench*
Editorial Assistant: *Raquel Kaplan*
Production Supervision: *Patti Guerrieri*
Cover Design Director: *Jerry Votta*
Cover Designer: *Anthony Gemmellaro*
Art Director: *Gail Cocker-Bogusz*
Manufacturing Manager: *Alexis R. Heydt*
Marketing Manager: *Chris Guzikowski*

Sun Microsystems Press Publisher: *Myrna Rivera*

First Printing

ISBN 0-13-146305-5

Sun Microsystems Press
A Prentice Hall Title

About Prentice Hall Professional Technical Reference

With origins reaching back to the industry's first computer science publishing program in the 1960s, and formally launched as its own imprint in 1986, Prentice Hall Professional Technical Reference (PH PTR) has developed into the leading provider of technical books in the world today. Our editors now publish over 200 books annually, authored by leaders in the fields of computing, engineering, and business.

Our roots are firmly planted in the soil that gave rise to the technical revolution. Our bookshelf contains many of the industry's computing and engineering classics: Kernighan and Ritchie's *C Programming Language*, Nemeth's *UNIX System Administration Handbook*, Horstmann's *Core Java*, and Johnson's *High-Speed Digital Design*.

PH PTR acknowledges its auspicious beginnings while it looks to the future for inspiration. We continue to evolve and break new ground in publishing by providing today's professionals with tomorrow's solutions.

PRENTICE HALL PTR

Contents

MANAGED BEANS

Topics in This Chapter

Chapter 2

A central theme of web application design is the separation of presentation and business logic. JSF uses *beans* to achieve this separation. JSF pages refer to bean properties, and the program logic is contained in the bean implementation code. Because beans are so fundamental to JSF programming, we discuss them in detail in this chapter.

The first half of the chapter discusses the essential features of beans that every JSF developer needs to know. We then present an example program that puts these essentials to work. The remaining sections cover more technical aspects about bean configuration and value binding expressions. You can safely skip these sections when you first read this book, and return to them when the need arises.

Definition of a Bean

According to the JavaBeans specification (available at http://java.sun.com/products/javabeans/), a Java Bean is "a reusable software component that can be manipulated in a builder tool." That is a pretty broad definition, and indeed, as you will see in this chapter, beans are used for a wide variety of purposes.

At first glance, a bean seems to be similar to an object. However, beans serve a different purpose. Objects are created and manipulated inside a Java program when the program calls constructors and invokes methods. However, beans can be configured and manipulated *without programming*.

NOTE: You may wonder where the term "bean" comes from. Well, Java is a synonym for coffee (at least in the United States), and coffee is made from beans that encapsulate its flavor. You may find the analogy cute or annoying, but the term has stuck.

The "classic" application for JavaBeans is a user-interface builder. A palette window in the builder tool contains component beans such as text fields, sliders, check boxes, and so on. Instead of writing Swing code, a user-interface designer drags and drops component beans into a form and customizes them, by selecting property values from a dialog (see Figure 2–1).

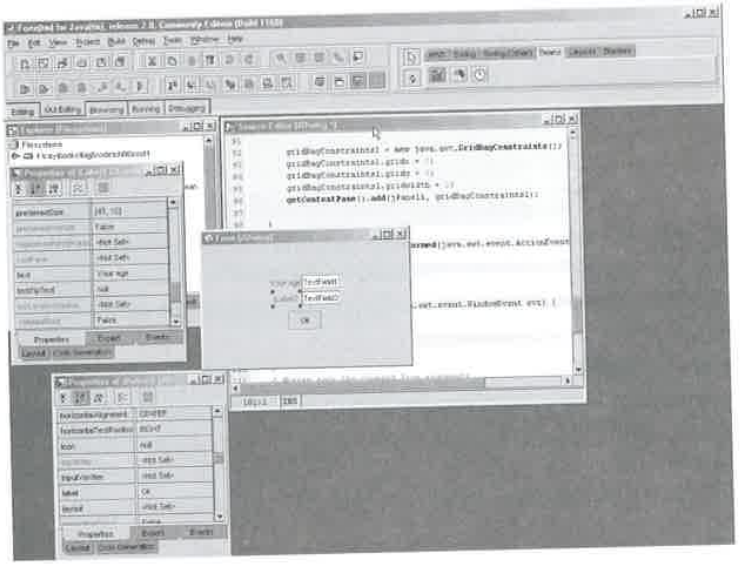


Figure 2–1 Customizing a Bean in a GUI Builder

In the context of JavaServer Faces, beans go beyond user interface components. You use beans whenever you need to wire up Java classes with web pages or configuration files.

Consider the `login` application in Chapter 1. A `UserBean` instance is configured in the `faces-config.xml` file:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```


This means: Construct an object of the class `com.corejsf.UserBean`, give it the name `user`, and keep it alive for the duration of the *session*, that is, for all requests that originate from the same client.

Once the bean has been defined, it can be accessed by JSF components. For example, this input field reads and updates the `password` property of the `user` bean.

```
<h:inputSecret value="#{user.password}"/>
```

As you can see, the JSF developer does not need to write any code to construct and manipulate the `user` bean.

In a JSF application, beans are commonly used for the following purposes:

- For user interface components (traditional user interface beans)
- For tying together the behavior of a web form (called "backing beans")
- For business objects whose properties are displayed on web pages
- For services such as external data sources that need to be configured when an application is assembled

Because beans are so ubiquitous, we now turn to a review of those parts of the JavaBeans specification that are relevant to JSF programmers.

Bean Properties

Bean classes need to follow specific programming conventions in order to expose features that tools can use. We discuss these conventions in this section.

The most important features of a bean are the properties that it exposes. A *property* is any attribute of a bean that has

- a name
- a type
- methods for getting and/or setting the property value

For example, the `UserBean` class of the preceding chapter has a property with name `password` and type `String`. The methods `getPassword` and `setPassword` access the property value.

Some programming languages, in particular Visual Basic and C#, have direct support for properties. However, in Java, a bean is simply a class that follows certain coding conventions.

The JavaBeans specification puts a single demand on a bean class: It must have a default constructor, that is, a constructor without parameters. However, in order to define properties, a bean must either use a *naming pattern* for property getters and setters, or it must define property descriptors. The latter approach is quite tedious and not commonly used, and we will not discuss it here. See *Horstmann & Cornell, Core Java vol. 2 ch. 8, Sun Microsystems Press 2003* for more information.

Defining properties with naming patterns is straightforward. Consider the following pair of methods:

```
T getFoo()
void setFoo(T newValue)
```

The pair corresponds to a read-write property with type `T` and name `foo`. If you only have the first method, then the property is read-only. If you only have the second method, then the property is write-only.

The method names and signatures must match the pattern precisely. The method name must start with `get` or `set`. A `get` method must have no parameters. A `set` method must have one parameter and no return value. A bean class can have other methods, but they do not yield bean properties.

Note that the name of the property is the "decapitalized" form of the part of the method name that follows the `get` or `set` prefix. For example, `getFoo` gives rise to a property named `foo`, with the first letter turned into lower case. However, if the first *two* letters after the prefix are upper case, then the first letter stays unchanged. For example, the method name `getURL` defines a property `URL`, and not `uRL`.

For properties of type `boolean`, you have a choice of prefixes for the method that reads the property. Both

```
boolean isConnected()
```

and

```
boolean getConnected()
```

are valid names for the reader of the `connected` property.

NOTE: The JavaBean specification also defines indexed properties, specified by method sets such as the following:

```
T[] getFoo()
T getFoo(int index)
void setFoo(T[] newArray)
void setFoo(int index, T newValue)
```

However, JSF provides no support for accessing the indexed values.

The JavaBeans specification is silent on the *behavior* of the getter and setter methods. In many situations, these methods will simply manipulate an instance field. But they may equally well carry out more sophisticated operations, such as database lookups, data conversion, validation, and so on.

A bean class may have other methods beyond property getters and setters. Of course, those methods do not give rise to bean properties.

Value Binding Expressions

Many JSF user interface components have an attribute `value` that lets you specify either a value or a *binding* to a value that is obtained from a bean property. For example, you can specify a direct value.

```
<h:outputText value="Hello, World!"/>
```

Or you can specify a value binding.

```
<h:outputText value="#{user.name}"/>
```

In most situations, a value binding expression such as `#{user.name}` describes a property. Note that the binding can be used both for reading and writing when it is used in an input component, such as

```
<h:inputText value="#{user.name}"/>
```

The property getter is invoked when the component is rendered. The property setter is invoked when the user response is processed.

We will discuss the syntax of value binding expressions in detail starting on page 60.

NOTE: JSF value binding expressions are different from the JSTL/JSP 2.0 expression language. A JSTL expression always invokes property getters. For that reason, JSF uses the `#{...}` delimiters instead of the JSTL `${...}` syntax.

Message Bundles

When you implement a web application, it is a good idea to collect all message strings in a central location. This process makes it easier to keep messages consistent and, crucially, makes it easier to localize your application for other locales

JSF simplifies this process. First, you collect your message strings in a file in the time-honored "properties" format:

```
currentScore=Your current score is:
guessNext=Guess the next number in the sequence!
```

NOTE: Look into the API documentation of the `load` method of the `java.util.Properties` class for a precise description of the file format.

Save the file together with your classes, for example, in `WEB-INF/classes/com/corejsf/messages.properties`. You can choose any directory path and file name, but you must use the extension `.properties`.

Add the `f:loadBundle` element to your JSF page, like this:

```
<f:loadBundle basename="com.corejsf.messages" var="msgs"/>
```

This element loads the messages in the bundle into a map variable with the name `msgs`, and stores that variable in request scope. (The base name looks like a class name, and indeed the properties file is loaded by the class loader.)

You can now use value binding expressions to access the message strings:

```
<h:outputText value="#{msgs.guessNext}"/>
```

That's all there is to it! When you are ready to localize your application for another locale, you simply supply localized bundle files.

When you localize a bundle file, you need to add a locale suffix to the file name: an underscore followed by the lowercase two-letter ISO-639 language code. For example, German strings would be in `com/corejsf/messages_de.properties`.

NOTE: You can find a listing of all two- and three-letter ISO-639 language codes at http://www.loc.gov/standards/iso639-2/.

As part of the internationalization support in Java, the bundle that matches the current locale is automatically loaded. The default bundle without a locale prefix is used as a fallback when the appropriate localized bundle is not available. (See Chapter 10 of *Horstmann & Cornell, Core Java vol. 2* for a detailed description of Java internationalization.)

NOTE: When you prepare translations, keep one oddity in mind: message bundle files are not encoded in UTF-8. Instead, Unicode characters beyond 127 are encoded as `\uxxxx` escape sequences. The Java SDK utility `native2ascii` can create these files.

You can have multiple bundles for a particular locale. For example, you may want to have separate bundles for commonly used error messages.

Once you have prepared your message bundles, you need to decide how to set the locale of your application. You have three choices:

- You can add a `locale` attribute to the `f:view` element, for example,

  ```
  <f:view locale="de">
  ```

- You can set the default and supported locales in `WEB-INF/faces-config.xml` (or another application configuration resource):


```
<faces-config>
   <application>
      <locale-config>
         <default-locale>en</default-locale>
         <supported-locale>de</supported-locale>
      </locale-config>
  </application>
</faces-config>
```

When a browser connects to your application, it usually includes an `Accept-Language` value in the HTTP header (see http://www.w3.org/International/questions/qa-accept-lang-locales.html). JSF reads the header and finds the best match among the supported locales. You can test this feature by setting the preferred language in your browser—see Figure 2–2.

- You can call the `setLocale` method of the `UIViewRoot` object:

```
UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
viewRoot.setLocale(new Locale("de"));
```

See chapter 7 for more information.

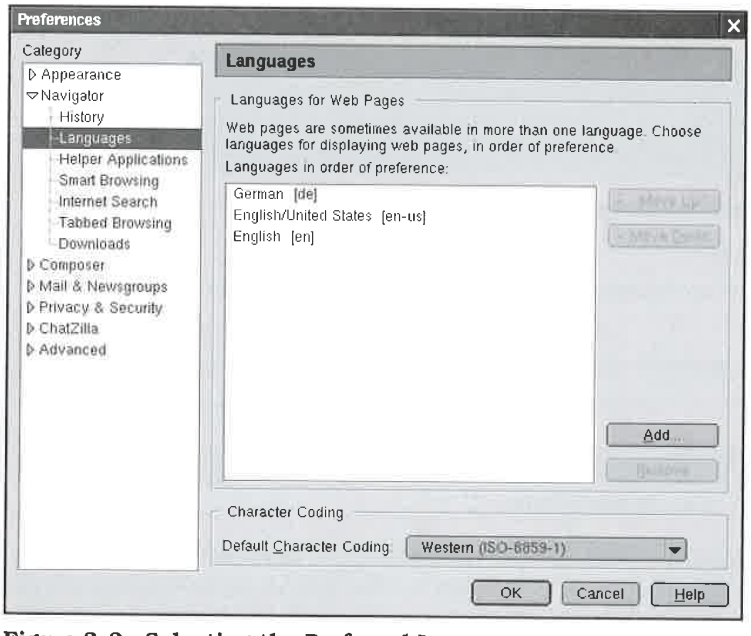


Figure 2–2 Selecting the Preferred Language

A Sample Application

After all these rather abstract rules and regulations, it is time for a concrete example. The application presents a series of quiz questions. Each question displays a sequence of numbers and asks the participant to guess the next number of the sequence.

For example, Figure 2–3 asks for the next number in the sequence

```
3 1 4 1 5
```

You often find puzzles of this kind in tests that purport to measure intelligence. To solve the puzzle, you need to find the pattern. In this case, we have the first digits of π.

Type in the next number in the sequence (9), and the score goes up by one.

NOTE: There is a Java-compatible mnemonic for the digits of π: "Can I have a small container of coffee?" Count the letters in each word, and you get 3 1 4 1 5 9 2 6. See http://dir.yahoo.com/Science/Mathematics/Numerical_Analysis/Numbers/Specific_Numbers/Pi/Mnemonics/ for more elaborate memorization aids.

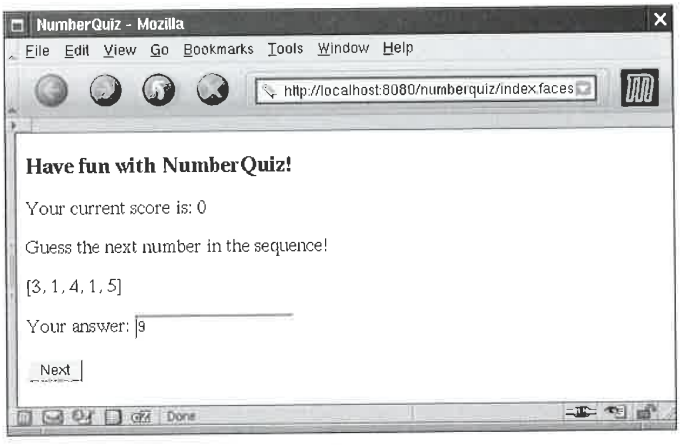


Figure 2–3 The Number Quiz

In this example, we place the quiz questions in the `faces-config.xml` file. Of course in a real application, you would be more likely to store this information in a database, but the purpose of the example is to demonstrate how to configure beans that have complex structure.

We start out with a ProblemBean class. A ProblemBean has two properties: solution, of type int, and sequence, of type ArrayList—see Listing 2–1.

Listing 2–1 numberquiz/WEB-INF/classes/com/corejsf/ProblemBean.java

```
1. package com.corejsf;
2. import java.util.ArrayList;
3.
4. public class ProblemBean {
5.    private ArrayList sequence;
6.    private int solution;
7.
8.    public ProblemBean() {}
9.
10.   public ProblemBean(int[] values, int solution) {
11.      sequence = new ArrayList();
12.      for (int i = 0; i < values.length; i++)
13.         sequence.add(new Integer(values[i]));
14.      this.solution = solution;
15.   }
16.
17.   // PROPERTY: sequence
18.   public ArrayList getSequence() { return sequence; }
19.   public void setSequence(ArrayList newValue) { sequence = newValue; }
20.
21.   // PROPERTY: solution
22.   public int getSolution() { return solution; }
23.   public void setSolution(int newValue) { solution = newValue; }
24. }
```

Next, we define a bean for the quiz with the following properties:

- problems: a write-only property to set the quiz problems
- score: a read-only property to get the current score
- current: a read-only property to get the current quiz problem
- answer: a property to get and set the answer that the user provides

The problems property is unused in this sample program—we initialize the problem set in the QuizBean constructor. However, on page 57, you will see how to set up the problem set inside faces-config.xml, without having to write any code.

The current property is used to display the current problem. However, the value of the current property is a ProblemBean object, and we cannot directly display that object in a text field. We make a second property access to get the number sequence:


```
<h:outputText value="#{quiz.current.sequence}"/>
```

The value of the sequence property is an ArrayList. When it is displayed, it is converted to a string by a call to the toString method. The result is a string of the form

```
[3, 1, 4, 1, 5]
```

Finally, we do a bit of dirty work with the answer property. We tie the answer property to the input field.

```
<h:inputText value="#{quiz.answer}"/>
```

When the input field is displayed, the getter is called, and we define the getAnswer method to return an empty string.

When the form is submitted, the setter is called with the value that the user typed into the input field. We define setAnswer to check the answer, update the score, and advance to the next problem.

```
public void setAnswer(String newValue) {
   try {
      int answer = Integer.parseInt(newValue.trim());
      if (getCurrent().getSolution() == answer) score++;
      currentIndex = (currentIndex + 1) % problems.size();
   }
   catch (NumberFormatException ex) {
   }
}
```

Strictly speaking, it is a bad idea to put code into a property setter that is unrelated to the task of setting the property. Updating the score and advancing to the next problem should really be contained in a handler for the button action. However, we have not yet discussed how to react to button actions, so we use the flexibility of the setter method to our advantage.

Another weakness of our sample application is that we haven't yet covered how to stop at the end of the quiz. Instead, we just wrap around to the beginning, letting the user rack up a higher score. You will learn in the next chapter how to do a better job. Remember—the point of this application is to show you how to configure and use beans.

Finally, note that we use message bundles for internationalization. Try switching your browser language to German, and the program will appear as in Figure 2–4.

This finishes our sample application. Figure 2–5 shows the directory structure. The remaining code is in Listings 2–2 through 2–6.

Figure 2–4 Viel Spaß mit dem Zahlenquiz!

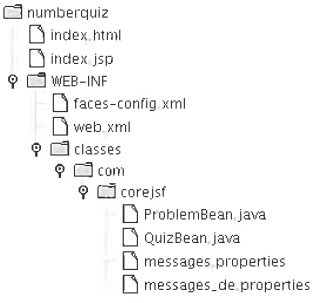


Figure 2–5 The Directory Structure of the Number Quiz Example

Listing 2–2 numberquiz/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
4.
5.    <f:view>
6.       <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.       <head>
8.          <title><h:outputText value="#{msgs.title}"/></title>
9.       </head>
```


Listing 2–2 numberquiz/index.jsp (cont.)

```
10.     <body>
11.        <h:form>
12.           <h3>
13.              <h:outputText value="#{msgs.heading}"/>
14.           </h3>
15.           <p>
16.              <h:outputText value="#{msgs.currentScore}"/>
17.              <h:outputText value="#{quiz.score}"/>
18.           </p>
19.           <p>
20.              <h:outputText value="#{msgs.guessNext}"/>
21.           </p>
22.           <p>
23.              <h:outputText value="#{quiz.current.sequence}"/>
24.           </p>
25.           <p>
26.              <h:outputText value="#{msgs.answer}"/>
27.              <h:inputText value="#{quiz.answer}"/></p>
28.           <p>
29.              <h:commandButton value="#{msgs.next}" action="next"/>
30.           </p>
31.        </h:form>
32.     </body>
33.  </f:view>
34. </html>
```

Listing 2–3 numberquiz/WEB-INF/classes/com/corejsf/QuizBean.java

```
1. package com.corejsf;
2. import java.util.ArrayList;
3.
4. public class QuizBean {
5.    private ArrayList problems = new ArrayList();
6.    private int currentIndex;
7.    private int score;
8.
9.    public QuizBean() {
10.      problems.add(
11.         new ProblemBean(new int[] { 3, 1, 4, 1, 5 }, 9)); // pi
12.      problems.add(
13.         new ProblemBean(new int[] { 1, 1, 2, 3, 5 }, 8)); // fibonacci
14.      problems.add(
15.         new ProblemBean(new int[] { 1, 4, 9, 16, 25 }, 36)); // squares
16.      problems.add(
```


Listing 2-3 numberquiz/WEB-INF/classes/com/corejsf/QuizBean.java (cont.)

```
17.         new ProblemBean(new int[] { 2, 3, 5, 7, 11 }, 13)); // primes
18.      problems.add(
19.         new ProblemBean(new int[] { 1, 2, 4, 8, 16 }, 32)); // powers of 2
20.   }
21.
22.   // PROPERTY: problems
23.   public void setProblems(ArrayList newValue) {
24.      problems = newValue;
25.      currentIndex = 0;
26.      score = 0;
27.   }
28.
29.   // PROPERTY: score
30.   public int getScore() { return score; }
31.
32.   // PROPERTY: current
33.   public ProblemBean getCurrent() {
34.      return (ProblemBean) problems.get(currentIndex);
35.   }
36.
37.   // PROPERTY: answer
38.   public String getAnswer() { return ""; }
39.   public void setAnswer(String newValue) {
40.      try {
41.         int answer = Integer.parseInt(newValue.trim());
42.         if (getCurrent().getSolution() == answer) score++;
43.         currentIndex = (currentIndex + 1) % problems.size();
44.      }
45.      catch (NumberFormatException ex) {
46.      }
47.   }
48. }
```

Listing 2-4 quizbean/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2.
3. <!DOCTYPE faces-config PUBLIC
4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
6.
7. <faces-config>
8.    <application>
9.       <locale-config>
10.         <default-locale>en</default-locale>
```

Listing 2–4 quizbean/WEB-INF/faces-config.xml (cont.)

```
11.         <supported-locale>de</supported-locale>
12.      </locale-config>
13.   </application>
14.
15.   <navigation-rule>
16.      <from-view-id>/index.faces</from-view-id>
17.      <navigation-case>
18.         <from-outcome>next</from-outcome>
19.         <to-view-id>/index.faces</to-view-id>
20.      </navigation-case>
21.   </navigation-rule>
22.
23.   <managed-bean>
24.      <managed-bean-name>quiz</managed-bean-name>
25.      <managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
26.      <managed-bean-scope>session</managed-bean-scope>
27.   </managed-bean>
28. </faces-config>
```

Listing 2–5 quizbean/WEB-INF/classes/com/corejsf/messages.properties

```
1. title=NumberQuiz
2. heading=Have fun with NumberQuiz!
3. currentScore=Your current score is:
4. guessNext=Guess the next number in the sequence!
5. answer=Your answer:
6. next=Next
```

Listing 2–6 quizbean/WEB-INF/classes/com/corejsf/messsages_de.properties

```
1. title=Zahlenquiz
2. heading=Viel Spa\u00df mit dem Zahlenquiz!
3. currentScore=Ihre Punktzahl:
4. guessNext=Raten Sie die n\u00e4chste Zahl in der Folge!
5. answer=Ihre Antwort:
6. next=Weiter
```

Backing Beans

Sometimes, it is convenient to design a bean that contains some or all component objects of a web form. Such a bean is called a *backing bean* for the web form. For example, we can turn the `QuizBean` into a backing bean by adding properties for the component on the form:


```
public class QuizBean {
   private UIOutput scoreComponent;
   private UIInput answerComponent;

   // PROPERTY: scoreComponent
   public UIOutput getScoreComponent() { return scoreComponent; }
   public void setScoreComponent(UIOutput newValue) { scoreComponent = newValue; }

   // PROPERTY: answerComponent
   public UIInput getAnswerComponent() { return answerComponent; }
   public void setAnswerComponent(UIInput newValue) { answerComponent = newValue; }
   ...
}
```

Output components belong to the `UIOutput` class and input components belong to the `UIInput` class. We will discuss these classes in greater detail in Chapter 9.

Why would you want such a bean? As we show in Chapters 6 and 7, it is sometimes necessary for validators and event handlers to have access to the actual components on a form. Moreover, visual JSF development environments generally use backing beans. These environments automatically generate the property getters and setters for all components that are dragged onto a form.

When you use a backing bean, you need to wire up the components on the form to those on the bean. You use the `binding` attribute for this purpose:

```
<h:outputText binding="#{quiz.scoreComponent}"/>
```

When the component tree for the form is built, the `getScoreComponent` method of the backing bean is called, but it returns `null`. As a result, an output component is constructed and installed into the backing bean with a call to `setScoreComponent`.

Backing beans have their uses, but they can also be abused. You should not use the user interface components as a repository for business data. If you use backing beans for your forms, you should still use beans for business objects.

Bean Scopes

For the convenience of the web application programmer, a servlet container provides separate scopes, each of which manages a table of name/value bindings.

These scopes typically hold beans and other objects that need to be available in different components of a web application.

Request Scope

The *request scope* is short-lived. It starts when an HTTP request is submitted and ends when the response is sent back to the client. The `f:loadBundle` tag places the bundle variable in request scope. You would place an object into request

scope only if you wanted to forward it to another processing phase inside the current request.

NOTE: If a request is *forwarded* to another request, all name/value pairs stored in the request scope are carried over to the new request. On the other hand, if a request is *redirected*, the request data are lost.

Session Scope

Recall that the HTTP protocol is *stateless*. The browser sends a request to the server, the server returns a response, and then neither the browser nor the server has any obligation to keep any memory of the transaction. This simple arrangement works well for retrieving basic information, but it is unsatisfactory for server-side applications. For example, in a shopping application, you want the server to remember the contents of the shopping cart.

For that reason, servlet containers augment the HTTP protocol to keep track of a *session*, that is, repeated connections by the same client. There are various methods for session tracking. The simplest method uses *cookies*: name/value pairs that a server sends to a client, hoping to have them returned in subsequent requests (see Figure 2–6).

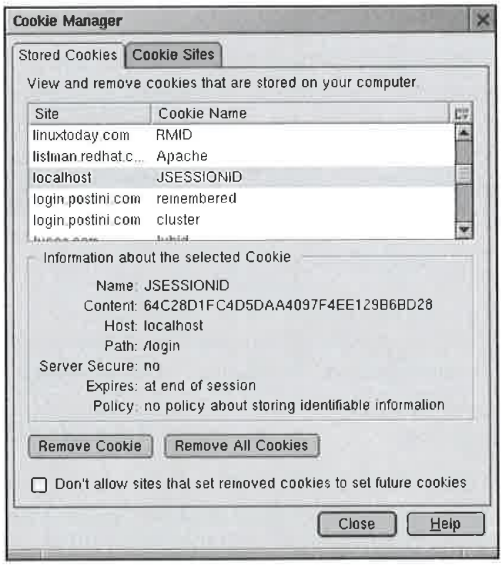


Figure 2–6 The Cookie Sent by a JSF Application

As long as the client doesn't deactivate cookies, the server receives a session identifier with each subsequent request.

Application servers use fallback strategies, such as *URL rewriting*, for dealing with those clients that don't return cookies. URL rewriting adds a session identifier to an URL, which looks somewhat like this:

```
http://corejsf.com/login/index.jsp;jsessionid=64C28D1FC...D28
```

Session tracking with cookies is completely transparent to the web developer, and the standard JSF tags automatically perform URL rewriting if a client does not use cookies.

The *session scope* persists from the time that a session is established until session termination. A session terminates if the web application invokes the `invalidate` method on the `HttpSession` object or if it times out.

Web applications typically place most of their beans into session scope.

For example, a `UserBean` can contain information about users that is accessible throughout the entire session. A `ShoppingCartBean` can be filled up gradually during the requests that make up a session.

Application Scope

Finally, the *application scope* persists for the entire duration of the web application. That scope is shared among all requests and all sessions.

You can see in Chapter 10 how to use the application scope for global beans such as LDAP directories.

Configuring Beans

This section describes how you can configure a bean in a configuration file. The details are rather technical. You may want to have a glance at this section and return to it when you need to configure beans with complex properties.

The most commonly used configuration file is `WEB-INF/faces-config.xml`. However, you can also place configuration information inside the following locations:

- Files named `META-INF/faces-config.ml` inside any JAR files loaded by the external context's class loader. (You use this mechanism if you deliver reusable components in a JAR file.)
- Files listed in the `javax.faces.CONFIG_FILES` initialization parameter inside `WEB-INF/web.xml`. For example,

```
<web-app>
   <context-param>
      <param-name>javax.faces.CONFIG_FILES</param-name>
```

```
        <param-value>WEB-INF/navigation.xml,WEB-INF/beans.xml</param-value>
    </context-param>
    ...
</web-app>
```

(This mechanism is attractive for builder tools because it separates navigation, beans, etc.)

For simplicity, we use `WEB-INF/faces-config.xml` in this chapter.

A bean is defined with a `managed-bean` element inside the top-level `faces-config` element. Minimally, you must specify the name, class, and scope of the bean.

```
<faces-config>
   <managed-bean>
      <managed-bean-name>user</managed-bean-name>
      <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>
</faces-config>
```

The scope can be `request`, `session`, `application`, or `none`. The `none` scope denotes an object that is not kept in one of the three scope maps. You use objects with scope `none` as building blocks when wiring up complex beans.

Setting Property Values

Let us start with a simple example. Here we customize a `UserBean` instance:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>name</property-name>
      <value>me</value>
   </managed-property>
   <managed-property>
      <property-name>password</property-name>
      <value>secret</value>
   </managed-property>
</managed-bean>
```

When the `user` bean is first looked up, it is constructed with the `UserBean()` default constructor. Then the `setName` and `setPassword` methods are executed.

To initialize a property with `null`, use a `null-value` element. For example,

```
<managed-property>
   <property-name>password</property-name>
   <null-value/>
</managed-property>
```


Initializing Lists and Maps

A special syntax initializes values that are of type `List` or `Map`. Here is an example of a list:

```
<list-entries>
   <value-class>java.lang.Integer</value.class>
   <value>3</value>
   <value>1</value>
   <value>4</value>
   <value>1</value>
   <value>5</value>
</list-entries>
```

Here we use the `java.lang.Integer` wrapper type since a `List` cannot hold values of primitive type.

The list can contain a mixture of `value` and `null-value` elements. The `value-class` is optional. If it is omitted, a list of `java.lang.String` objects is produced.

A map is more complex. You specify optional `key-class` and `value-class` elements (again, with a default of `java.lang.String`). Then you provide a sequence of `map-entry` elements, each of which has a `key` element followed by a `value` or `null-value` element.

Here is an example:

```
<map-entries>
   <key-class>java.lang.Integer</key-class>
   <map-entry>
      <key>1</key>
      <value>George Washington</value>
   </map-entry>
   <map-entry>
      <key>3</key>
      <value>Thomas Jefferson</value>
   </map-entry>
   <map-entry>
      <key>16</key>
      <value>Abraham Lincoln</value>
   </map-entry>
   <map-entry>
      <key>26</key>
      <value>Theodore Roosevelt</value>
   </map-entry>
</map-entries>
```

You can use `list-entries` and `map-entries` elements to initialize either a `managed-bean` or a `managed-property`, provided that the bean or property type is a `List` or `Map`.

Figure 2–7 shows a *syntax diagram* for the `managed-bean` element and all of its child elements. Simply follow the arrows to see which constructs are legal inside a `managed-bean` element. For example, the second graph tells you that a `managed-property` element starts with zero or more `description` elements, followed by zero or more `display-name` elements, zero or more `icons`, then a mandatory `property-name`, an optional `property-class`, and exactly one of the elements `value`, `null-value`, `values`, or `map-entries`.

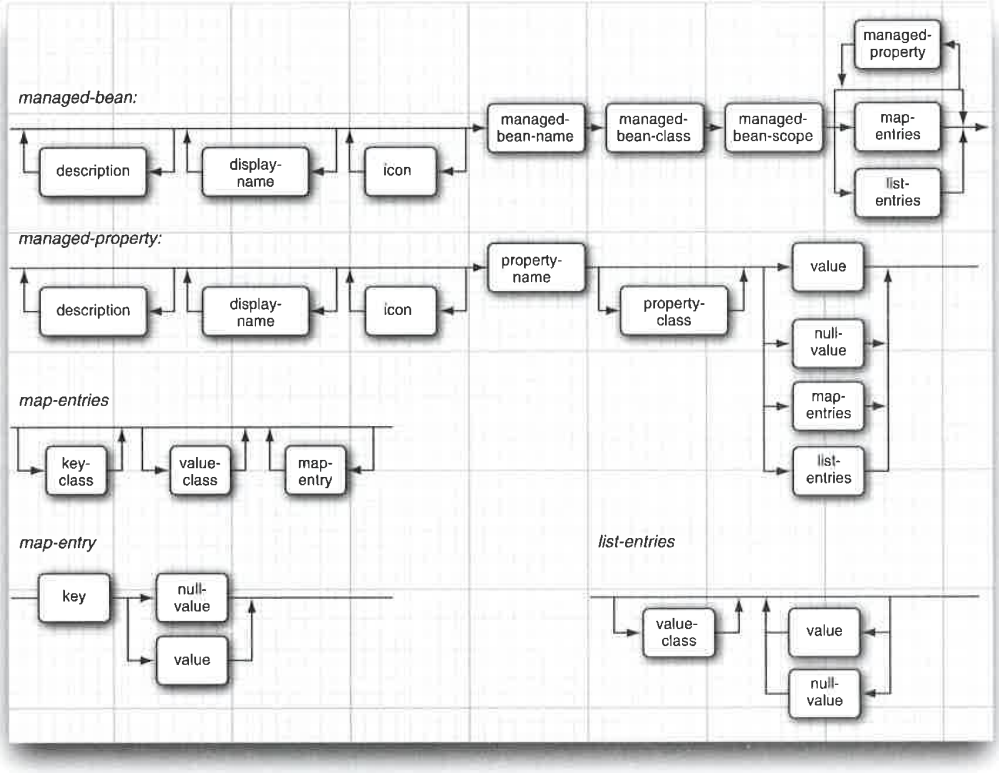


Figure 2–7 Syntax Diagram for `managed-bean` Elements

Chaining Bean Definitions

You can achieve more complex arrangements by using value binding expressions inside the `value` element to chain beans together. Consider the quiz bean in the `numberquiz` application.

The quiz contains a collection of problems, represented as the write-only `problems` property. You can configure it with the following instructions:

```
<managed-bean>
   <managed-bean-name>quiz</managed-bean-name>
   <managed-bean-class>com.corejsf.QuizBackingBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>problems</property-name>
      <list-entries>
         <value-class>com.corejsf.ProblemBean</value-class>
         <value>#{problem1}</value>
         <value>#{problem2}</value>
         <value>#{problem3}</value>
         <value>#{problem4}</value>
         <value>#{problem5}</value>
      </list-entries>
   </managed-property>
</managed-bean>
```

Of course, now we must define beans with names `problem1` through `problem5`, like this:

```
<managed-bean>
   <managed-bean-name>problem1</managed-bean-name>
   <managed-bean-class>
      com.corejsf.ProblemBean
   </managed-bean-class>
   <managed-bean-scope>none</managed-bean-scope>
      <managed-property>
         <property-name>sequence</property-name>
         <list-entries>
            <value-class>java.lang.Integer</value-class>
            <value>3</value>
            <value>1</value>
            <value>4</value>
            <value>1</value>
            <value>5</value>
         </list-entries>
      </managed-property>
```

```
    <managed-property>
      <property-name>solution</property-name>
      <value>9</value>
    </managed-property>
  </managed-bean>
```

When the quiz bean is requested, then the creation of the beans problem1 through problem5 is triggered automatically. You need not worry about the order in which you specify managed beans.

Note that the problem beans have scope none since they are never requested from a JSP page.

When you wire beans together, make sure that their scopes are compatible. Table 2–1 lists the permissible combinations.

Table 2–1 Compatible Bean Scopes

When defining a bean of this scope...	...you can use beans of these scopes
none	none
application	none, application
session	none, application, session
request	none, application, session, request

String Conversions

You specify property values and elements of lists or maps with a value element that contains a string. The enclosed string needs to be converted to the type of the property or element. For primitive types, this conversion is straightforward. For example, you can specify a boolean value with the string true or false.

For other property types, the JSF implementation attempts to locate a matching PropertyEditor. If a property editor exists, its setAsText method is invoked to convert strings to property values. Property editors are heavily used for client-side beans, to convert between property values and a textual or graphical representation that can be displayed in a property sheet (see Figure 2–8).

Defining a property editor is somewhat involved, and we refer the interested reader to *Horstmann & Cornell, Core Java, volume 2 chapter 8, Sun Microsystems Press 2002.*

Figure 2–8 A Property Sheet in a GUI Builder

Note that the rules are fairly restrictive. For example, if you have a property of type `URL`, you cannot simply specify the URL as a string, even though there is a constructor `URL(String)`. You would need to supply a property editor for the `URL` type or reimplement the property type as `String`.

Table 2–2 summarizes these conversion rules. They are identical to the rules for the `jsp:setProperty` action of the JSP specification.

Table 2–2 String Conversions

Target Type	Conversion
`int`, `byte`, `short`, `long`, `float`, `double`, or the corresponding wrapper type	The `valueOf` method of the wrapper type, or `0` if the string is empty.
`boolean` or `Boolean`	The result of `Boolean.valueOf`, or `false` if the string is empty.
`char` or `Character`	The first character of the string, or `(char) 0` if the string is empty
`String` or `Object`	A copy of the string; `new String("")` if the string is empty.
bean property	A type that calls the `setAsText` method of the property editor if it exists. If the property editor doesn't exist or it throws an exception, the property is set to `null` if the string is empty. An error occurs otherwise

NOTE: You now know how to use value binding expressions inside your JSF pages. Sometimes, you need to evaluate a value binding expression in your Java code. Use a sequence of statements such as the following:

```
FacesContext context = FacesContext.getCurrentInstance();
ValueBinding binding =
context.getApplication().createValueBinding("#{user.name}");
String name = (String) binding.getValue(context);
```

See Chapter 9 for more information.

The Syntax of Value Binding Expressions

In this section, we discuss the syntax for value binding expressions in gruesome detail. This section is intended for reference. Feel free to skip it at first reading.

Let us start with an expression of the form `a.b`. For now, we'll assume that we already know the object to which `a` refers. If `a` is an array, a list, or a map, then special rules apply—see the next subsection. If `a` is any other object, then `b` must be the name of a property of `a`. The exact meaning of `a.b` depends on whether the expression is used in *rvalue mode* or *lvalue mode*.

This terminology is used in the theory of programming languages to denote that an expression on the *right-hand side* of an assignment is treated differently from an expression on the *left-hand side*.

Consider the assignment

```
left = right;
```

A compiler generates different code for the `left` and `right` expressions. The `right` expression is evaluated in rvalue mode and yields a value. The `left` expression is evaluated in lvalue mode and stores a value in a location.

The same phenomenon happens when you use a value binding expression in a user interface component:

```
<h:inputText value="#{user.name}"/>
```

When the text field is rendered, the expression `user.name` is evaluated in rvalue mode, and the `getName` method is called. During decoding, the same expression is evaluated in lvalue mode, and the `setName` method is called.

In general, the expression `a.b` in rvalue mode is evaluated by calling the property getter, whereas `a.b` in lvalue mode calls the property setter.

Using Brackets

Just as in JavaScript, you can use brackets instead of the dot notation. That is, the following three expressions all have the same meaning:

```
a.b
a["b"]
a['b']
```

For example, `user.password`, `user["password"]`, and `user['password']` are equivalent expressions.

Why would anyone write `user["password"]` when `user.password` is much easier to type? There are a number of reasons.

- When you access an array or map, the `[]` notation is more intuitive.
- You can use the `[]` notation with strings that contain periods or dashes, for example, `msgs["error.password"]`.
- The `[]` notation allows you to dynamically compute a property: `a[b.propname]`.

TIP: Use single quotes in value binding expressions if you delimit attributes with double quotes: `value="#{user['password']}"`. Alternatively, you can switch single and double quotes: `value='#{user["password"]}'`.

Map and List Expressions

The value binding expression language goes beyond bean property access. For example, let `m` be an object of any class that implements the `Map` interface. Then `m["key"]` (or the equivalent `m.key`) is a binding to the associated value. In rvalue mode, the value

```
m.get("key")
```

is fetched. In lvalue mode, the statement

```
m.put("key", right);
```

is executed. Here, `right` is the "right-hand side" value that is assigned to `m.key`.

You can also access a value of any object of a class that implements the `List` interface (such as an `ArrayList`). You specify an integer index for the list position. For example, `a[i]` (or, if you prefer, `a.i`) binds the `i`th element of the list `a`. Here `i` can be an integer or a string that can be converted to an integer. The same rule applies for array types. As always, index values start at zero.

Table 2–3 summarizes these evaluation rules.

Table 2–3 Evaluating the Value Binding Expression `a.b`

Type of a	Type of b	lvalue mode	rvalue mode
`null`	any	error	`null`
any	`null`	error	`null`
`Map`	any	`a.put(b, right)`	`a.get(b)`
`List`	convertible to `int`	`a.set(b, right)`	`a.get(b)`
array	convertible to `int`	`a[b]`	`a[b]`
bean	any	call setter of property with name `b.toString()`	call getter of property with name `b.toString()`

CAUTION: Unfortunately, value bindings do not work for indexed properties. If `p` is an indexed property of a bean `b` and `i` is an integer, then `b.p[i]` does not access the `i`th value of the property. It is simply a syntax error. This deficiency is inherited from the JSTL expression language.

Resolving the Initial Term

Now you know how an expression of the form `a.b` is resolved. The rules can be applied repetitively to expressions such as `a.b.c.d` (or, of course, `a['b'].c["d"]`). We still need to discuss the meaning of the initial term `a`.

In the examples, you have seen so far, the initial term referred to a bean that was configured in the `faces-config.xml` file, or to a message bundle map. Those are indeed the most common situations. But it is also possible to specify other names.

There are a number of predefined objects. Table 2–4 shows the complete list. For example,

```
header['User-Agent']
```

is the value of the `User-Agent` parameter of the HTTP request that identifies the user's browser.

If the initial term is not one of the predefined objects, the JSF implementation looks for it in the *request*, *session*, and *application scopes*, in that order. Those scopes are map objects that are managed by the servlet container. For example, when you define a managed bean, its name and value are added to the appropriate scope map.

Table 2–4 Predefined Objects in the Value Binding Expression Language

Variable Name	Meaning
header	a Map of HTTP header parameters, containing only the first value for each name
headerValues	a Map of HTTP header parameters, yielding a String[]array of all values for a given name
param	a Map of HTTP request parameters, containing only the first value for each name
paramValues	a Map of HTTP request parameters, yielding a String[]array of all values for a given name
cookie	a Map of the cookie names and values of the current request
initParam	a Map of the initialization parameters of this web application. Initialization parameters are discussed in Chapter 10.
requestScope	a Map of all request scope attributes
sessionScope	a Map of all session scope attributes
applicationScope	a Map of all application scope attributes
facesContext	The FacesContext instance of this request. This class is discussed in Chapter 6
view	The UIViewRoot instance of this request. This class is discussed in Chapter 7

Finally, if the name is still not found, it is passed to the VariableResolver of the JSF application. The default variable resolver looks up managed-bean elements in a configuration resource, typically the faces-config.xml file.

Consider, for example, the expression

```
#{user.password}
```

The term user is not one of the predefined objects. When it is encountered for the first time, it is not an attribute name in request, session, or application scope. Therefore, the variable resolver processes the faces-config.xml entry.

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

It calls the default constructor of the class com.corejsf.UserBean. Next, it adds an association to the sessionScope map. Finally, it returns the object as the result of the lookup.

When the term user needs to be resolved again in the same session, it is located in the session scope.

Composite Expressions

You can use a limited set of operators inside value binding expressions:

- arithmetic operators + - * / %. The last two operators have alphabetic variants div and mod.
- relational operators < <= > >= == != and their alphabetic variants lt le gt ge eq ne. (The first four variants are required for XML safety.)
- logical operators && || ! and their alphabetic variants and or not. (The first variant is required for XML safety.)
- the empty operator. The expression empty a is true if a is null, an array or String of length 0, or a Collection or Map of size 0.
- the ternary ?: selection operator

Operator precedence follows the same rules as in Java. The empty operator has the same precedence as the unary - and ! operators.

Generally, you don't want to do a lot of expression computation in web pages—that would violate the separation of presentation and business logic. However, occasionally the presentation layer can benefit from operators. For example, suppose you want to hide a component when the hide property of a bean is true. To hide a component, you set its rendered attribute to false. Inverting the bean value requires the ! (or not) operator:

```
<h:inputText rendered="#{!bean.hide}" ... />
```

Finally, you can concatenate plain strings and value binding expressions, simply by placing them next to each other. Consider, for example,

```
<h:outputText value="#{messages.greeting}, #{user.name}!"/>
```

The statement concatenates four strings: the string returned from #{messages.greeting}, the string consisting of a comma and a space, the string returned from #{user.name}, and the string "!".

You have now seen all the rules that are applied to resolve value binding expressions. Of course, in practice, most expressions are simply of the form #{bean.property}. Just come back to this section when you need to tackle a more complex expression.

Method Binding Expressions

A *method binding expression* denotes an object together with a method that can be applied to it.

For example, here is a typical use of a method binding expression.

```
<h:commandButton action="#{user.checkPassword}"/>
```

We assume that `user` is a value of type `UserBean` and `checkPassword` is a method of that class. The method binding expression is simply a convenient way of describing a method invocation that needs to be carried out at some future time.

When the expression is evaluated, the method is applied to the object.

In our example, the command button component will call `user.checkPassword()` and pass the returned string to the navigation handler.

Syntax rules for method binding expressions are similar to those of value binding expressions. All but the last component are used to determine an object. The last component must be the name of a method that can be applied to that object.

Four component attributes can take a method binding expression:

- `action` (see Chapter 3)
- `validator` (see Chapter 6)
- `actionListener` (see Chapter 7)
- `valueChangeListener` (see Chapter 7)

The parameter and return types of the method depend on the context in which the method binding is used. For example, an `action` must be bound to a method with no parameters and return type `String`, whereas an `actionListener` is bound to a method with one parameter of type `ActionEvent` and return type `void`. The code that invokes the method binding is responsible for supplying parameter values and processing the return value.

Subviews and Tiles

Topics in This Chapter

Chapter 8

User interfaces are typically the most volatile aspect of web applications during development, so it's crucial to create flexible and extensible interfaces. This chapter shows you how to achieve that flexibility and extensibility by including common content. First we discuss standard JSP mechanisms—JSP includes and JSTL imports—you can use to include common content in a JSF application. Next we explore the use of Struts's Tiles package—which lets you encapsulate layout in addition to content, among other handy features—with JSF.

Common Layouts

Many popular web sites, such as nytimes.com, java.sun.com, or amazon.com, use a common layout for their web pages. For example, all three of the web sites listed above use a header-menu-content layout as depicted in Figure 8–1.

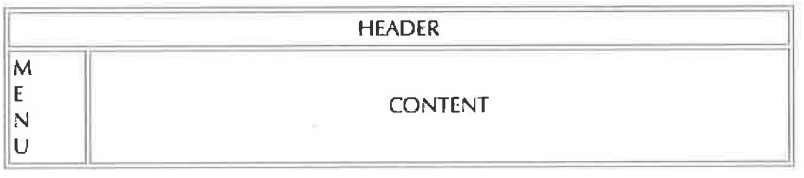


Figure 8–1 A Typical Web Page Layout

You can use HTML frames to achieve the layout shown in Figure 8–1, but frames are undesirable for several reasons. For example, frames make it hard for users to bookmark pages. Frames also generate separate requests, which can be problematic for web applications. Including content, which is the focus of this chapter, is generally preferred over frames.

A Book Viewer and a Library

To illustrate implementing layouts, including common content, and using Tiles, we discuss two applications in this chapter: a book viewer and a library. Those applications are shown in Figure 8–2 and Figure 8–3, respectively.

Figure 8–2 The Book Viewer

Figure 8–3 The Library

The book viewer is intuitive. If you click on a chapter link, that chapter is shown in the content region of the web page. The library is an extension of the book viewer that lets you view more than one book. You can select books from the menu at the top of the web page.

The book viewer addresses the following topics:

- "Monolithic JSF Pages" on page 320
- "Common Content Inclusion" on page 325
- "Looking at Tiles" on page 330

- "Parameterizing Tiles" on page 334
- "Extending Tiles" on page 335

The library illustrates these Tiles features:

- "Nested Tiles" on page 340
- "Tile Controllers" on page 340

Coverage of the book viewer begins in the next section. The library is discussed in "The Library" on page 338.

NOTE: For the examples in this chapter, we downloaded Alice in Wonderland and Peter Pan from the Project Gutenberg web site—http://promo.net/pg/—chopped them up into chapters and converted them to HTML.

The Book Viewer

The book viewer is rather limited in scope. It supports only a single book, which is a bean that we define in the faces configuration file. The name of that bean is `book`.

The `book` bean has these properties:

- `titleKey`
- `image`
- `numChapters`
- `chapterKeys`

The `titleKey` property represents a key in a resource bundle for the book's title. In the book viewer's properties file we have the key/value pair `titleKey=Alice in Wonderland`. When we display the book's title, we use the `titleKey` property like this:

```
<h:outputText value="#{msgs[book.titleKey]}"/>
```

The `image` property is a string. The application interprets that string as a URL and loads it in the book viewer's header like this:

```
<h:graphicImage url="#{book.image}"/>
```

The `chapterKeys` property is a read-only list of keys, one for each chapter. The book viewer populates the book viewer's menu with corresponding values from a resource bundle:


```
<h:dataTable value="#{book.chapterKeys}" var="chapterKey">
    <h:commandLink>
        <h:outputText value="#{msgs[chapterKey]}"/>
        ...
    </h:commandLink>
</h:dataTable>
```

The Book class uses the numChapters property to compute the chapter keys.

The implementation of the Book class is rather mundane. You can see it in Listing 8–3 on page 323. Here's how we define an instance of the Book class in faces-config.xml:

```
<faces-config>
   <!-- The book -->
   <managed-bean>
      <managed-bean-name>book</managed-bean-name>
      <managed-bean-class>com.corejsf.Book</managed-bean-class>
      <managed-bean-scope>request</managed-bean-scope>

      <managed-property>
        <property-name>titleKey</property-name>
        <value>aliceInWonderland</value>
      </managed-property>

      <managed-property>
        <property-name>image</property-name>
        <value>cheshire.jpg</value>
      </managed-property>

      <managed-property>
        <property-name>numChapters</property-name>
        <property-class>java.lang.Integer</property-class>
        <value>12</value>
      </managed-property>
   </managed-bean>
</faces-config>
```

There are many ways to implement a header-menu-content layout, as shown in Figure 8–1 on page 315. In this section we look at three options: a monolithic JSF page, inclusion of common content, and Tiles.

NOTE: We don't set the book's chapterKeys property in faces-config.xml. That's because the Book class creates that list of chapter keys for us. All we have to do is define the numChapters property.

Monolithic JSF Pages

A monolithic JSF page is perhaps the quickest way to implement the book viewer shown in Figure 8–2; for example, here's a naive implementation:

```
<%-- A panel grid, which resides in a form, for the entire page --%>
<h:panelGrid columns="2" styleClass="book"
    columnClasses="menuColumn, chapterColumn">

    <%-- The header, containing an image, title and horizontal rule --%>
    <f:facet name="header">
        <h:panelGrid columns="1" styleClass="bookHeader">
            <h:graphicImage value="#{book.image}"/>
            <h:outputText value="#{msgs[book.titleKey]}" styleClass='bookTitle'/>
            <f:verbatim><hr></f:verbatim>
        </h:panelGrid>
    </f:facet>

    <%-- Column 1: The menu, which consists of chapter links --%>
    <h:dataTable value="#{book.chapterKeys}" var="chapterKey"
                 styleClass="links" columnClasses="linksColumn">
        <h:column>
            <h:commandLink>
                <h:outputText value="#{msgs[chapterKey]}"/>
                <f:param name="chapter" value="#{chapterKey}"/>
            </h:commandLink>
        </h:column>
    </h:dataTable>

    <%-- Column 2: The chapter content --%>
    <f:verbatim>
        <c:import url="${param.chapter}.html"/>
    </f:verbatim>
</h:panelGrid>
```

The book viewer is implemented with a panel grid with two columns. The header region is populated with an image, text, and HTML horizontal rule. Besides the header, the panel grid has only one row—the menu occupies the left column and the current chapter is displayed in the right column.

The menu is composed of chapter links. By default, `Book.getChapterKeys()` returns a list of strings that look like this:

```
chapter1
chapter2
...
chapterN
```

Chapter N represents the last chapter in the book. In the book viewer's resource bundle, we define values for those keys:


```
chapter1=Chapter 1
chapter2=Chapter 2
...
```

To create chapter links, we use `h:dataTable` to iterate over the book's chapter keys. For every chapter, we create a link whose text corresponds to the chapter key's value with this expression: `#{msgs[chapterKey]}`. So we wind up with Chapter 1 … Chapter12 displayed in the menu when the number of chapters is 12.

The right column is reserved for chapter content. That content is included with JSTL's `c:import` tag.

The directory structure for the book viewer is shown in Figure 8–4. The monolithic JSF page version of the book viewer is listed in Listing 8–1 through Listing 8–5.

NOTE: Notice the `f:param` tag inside `h:commandLink`. The JSF framework turns that parameter into a request parameter—named `chapter`—when the link is activated. When the page is reloaded, that request parameter is used to load the chapter's content like this: `<c:import url="${param.chapter}"/>`

NOTE: When we import book chapters, we place the `c:import` tag in the body of an `f:verbatim` tag.

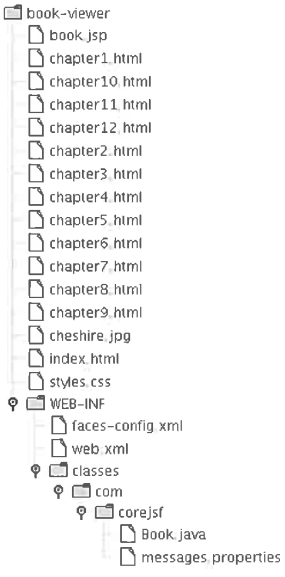


Figure 8–4 The book-viewer Directory Structure

Listing 8–1 book-viewer-monolith/book.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/core"  prefix="f" %>
4.    <%@ taglib uri="http://java.sun.com/jsf/html"  prefix="h" %>
5.    <f:view>
6.       <head>
7.          <link href="styles.css" rel="stylesheet" type="text/css"/>
8.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
9.          <title><h:outputText value="#{msgs.bookWindowTitle}"/></title>
10.      </head>
11.      <body>
12.         <h:form>
13.            <h:panelGrid columns="2" styleClass="book"
14.                  columnClasses="menuColumn, chapterColumn">
15.               <f:facet name="header">
16.                  <h:panelGrid columns="1" styleClass="bookHeader">
17.                     <h:graphicImage value="#{book.image}"/>
18.                     <h:outputText value="#{msgs[book.titleKey]}"
19.                              styleClass='bookTitle'/>
20.
21.                     <f:verbatim><hr/></f:verbatim>
22.                  </h:panelGrid>
23.               </f:facet>
24.
25.               <h:dataTable value="#{book.chapterKeys}" var="chapterKey"
26.                        styleClass="links" columnClasses="linksColumn">
27.                  <h:column>
28.                     <h:commandLink>
29.                        <h:outputText value="#{msgs[chapterKey]}"/>
30.                        <f:param name="chapter" value="#{chapterKey}"/>
31.                     </h:commandLink>
32.                  </h:column>
33.               </h:dataTable>
34.
35.               <f:verbatim>
36.                  <c:import url="${param.chapter}.html"/>
37.               </f:verbatim>
38.            </h:panelGrid>
39.         </h:form>
40.      </body>
41.   </f:view>
42. </html>
```

Listing 8–2 book-viewer-monolith/WEB-INF/faces-config.txt

```
1. <?xml version="1.0"?>
2.
3. <!DOCTYPE faces-config PUBLIC
4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
6.
7. <faces-config>
8.    <!-- The book -->
9.    <managed-bean>
10.       <managed-bean-name>book</managed-bean-name>
11.       <managed-bean-class>com.corejsf.Book</managed-bean-class>
12.       <managed-bean-scope>request</managed-bean-scope>
13.
14.       <managed-property>
15.          <property-name>titleKey</property-name>
16.          <value>aliceInWonderland</value>
17.       </managed-property>
18.
19.       <managed-property>
20.          <property-name>image</property-name>
21.          <value>cheshire.jpg</value>
22.       </managed-property>
23.
24.       <managed-property>
25.          <property-name>numChapters</property-name>
26.          <property-class>java.lang.Integer</property-class>
27.          <value>12</value>
28.       </managed-property>
29.    </managed-bean>
30. </faces-config>
```

Listing 8–3 book-viewer-monolith/WEB-INF/classes/com/corejsf/Book.java

```
1. package com.corejsf;
2.
3. import java.util.LinkedList;
4. import java.util.List;
5.
6. public class Book {
7.    private String titleKey;
8.    private String image;
9.    private int numChapters;
10.   private List chapterKeys = null;
```


Listing 8–3 book-viewer-monolith/WEB-INF/classes/com/corejsf/Book.java (cont.)

```
11.
12.    // PROPERTY: titleKey
13.    public void setTitleKey(String titleKey) { this.titleKey = titleKey; }
14.    public String getTitleKey() { return titleKey; }
15.
16.    // PROPERTY: image
17.    public void setImage(String image) { this.image = image; }
18.    public String getImage() { return image; }
19.
20.    // PROPERTY: numChapters
21.    public void setNumChapters(int numChapters) { this.numChapters = numChapters;}
22.    public int getNumChapters() { return numChapters; }
23.
24.    // PROPERTY: chapterKeys
25.    public List getChapterKeys() {
26.       if(chapterKeys == null) {
27.          chapterKeys = new LinkedList();
28.          for(int i=1; i <= numChapters; ++i)
29.             chapterKeys.add("chapter" + i);
30.       }
31.       return chapterKeys;
32.    }
33. }
```

Listing 8–4 book-viewer-monolith/WEB-INF/classes/com/corejsf/messages.properties

```
1. bookWindowTitle=Welcome to Alice in Wonderland
2. aliceInWonderland=Alice in Wonderland
3.
4. chapter1=Chapter 1
5. chapter2=Chapter 2
6. chapter3=Chapter 3
7. chapter4=Chapter 4
8. chapter5=Chapter 5
9. chapter6=Chapter 6
10. chapter7=Chapter 7
11. chapter8=Chapter 8
12. chapter9=Chapter 9
13. chapter10=Chapter 10
14. chapter11=Chapter 11
15. chapter12=Chapter 12
16. chapter13=Chapter 13
17. chapter14=Chapter 14
18. chapter15=Chapter 15
```


Listing 8–5 book-viewer-monolith/styles.css

```
.bookHeader {
   width: 100%;
   text-align: center;
   background-color: #eee;
   padding: 0 px;
   border: thin solid CornflowerBlue;
}
.bookTitle {
   text-align: center;
   font-style: italic;
   font-size: 1.3em;
   font-family: Helvetica;
}
.book {
   vertical-align: top;
   width: 100%;
   height: 100%;
}
.menuColumn {
   vertical-align: top;
   background-color: #eee;
   width: 100px;
   border: thin solid #777;
}
.chapterColumn {
   vertical-align: top;
   text-align: left;
   width: *;
}
```

Common Content Inclusion

A monolithic JSF page is a poor choice for the book viewer because the JSF page is difficult to modify. Also, realize that our monolithic JSF page represents two things: layout and content.

Layout is implemented with an `h:panelGrid` tag, and content is represented by various JSF tags, such as `h:graphicImage`, `h:outputText`, `h:commandLink`, and the book chapters. Realize that *with a monolithic JSF page, we cannot reuse content or layout.*

In the next section we concentrate on including content. In "Looking at Tiles" on page 330, we discuss including layout.

Content Inclusion in JSP-Based Applications

Instead of cramming a bunch of code into a monolithic JSF page, as we did in Listing 8–1 on page 322, it's better to include common content so you can reuse that content in other JSF pages. With JSP, you have three choices for including content:

- `<%@ include file="header.jsp"% >`
- `<jsp:include page="header.jsp"/>`
- `<c:import url="header.jsp"/>`

The first choice listed above—the JSP `include` directive—includes the specified file before the enclosing JSF page is compiled to a servlet. However, the `include` directive suffers from an important limitation: If the included file's content changes after the enclosing page was first processed, those changes are not reflected in the enclosing page. That means you must manually update the enclosing pages—whether the including pages changed or not—whenever included content changes.

The last two choices listed above include the content of a page at runtime and merge the included content with the including JSF page. Because the inclusion happens at runtime, changes to included pages are always reflected when the enclosing page is redisplayed. For that reason, `jsp:include` and `c:import` are usually preferred to the `include` directive.

The `c:import` tag works just like `jsp:include`, but it has more features; for example, `c:import` can import resources from another web application, whereas `jsp:include` cannot. Also, prior to JSP 2.0, you cannot use JSP expressions for `jsp:include` attributes, whereas you can with `c:import`. Remember that you must import the JSTL core tag library to use `c:import`.

Throughout this chapter, we use `c:import` for consistency. You can use either `jsp:include` or `c:import` to dynamically include content. If you don't need `c:import`'s extra features, then it's ever-so-slightly easier to use `jsp:include` because you don't need to import the JSTL core tag library.

JSF-Specific Considerations

Regardless of whether you include content with the `include` directive, `jsp:include`, or `c:import`, there are two special considerations that you must take into account when you include content in a JavaServer Faces application.

1. You must wrap included JSF tags in an `f:subview` tag.
2. Included JSF tags cannot contain `f:view` tags.

The first rule applies to included content that contains JSF tags. For example, the book viewer should encapsulate header content in its own JSF page so that we can reuse that content:


```
<%-- This is header .jsp --%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<h:panelGrid columns="1" styleClass="header">
    <h:graphicImage value="books/book/cheshire.jpg"/>
    <h:outputText value="#{msgs.bookTitle}" styleClass="bookTitle"/>
    ...
</h:panelGrid>
```

Now we can include that content from the original JSF page:

```
<%-- This is from the original JSF page --%>
<f:view>
    ...
    <f:subview id="header">
        <c:import url="header.jsp"/>
    </f:subview>
    ...
</f:view>
```

You must assign an ID to each subview. The standard convention for including content is to name the subview after the imported JSF page.

JSF views, which are normally web pages, can contain an unlimited number of subviews. But there can be only one view. Because of that restriction, included JSF tags—which must be wrapped in a subview—cannot contain `f:view` tags.

CAUTION: The `book-viewer-include` application maps the Faces servlet to `*.faces`. That means you can start the application with this URL: http://www.localhost:8080/book-viewer-include/book.faces. The Faces servlet maps `books.faces` to `books.jsp`. However, you can't use the `faces` suffix when you use `c:import`. If you use `c:import`, you must use the `jsp` suffix.

Content Inclusion in the Book Viewer

To include content in the book viewer, we split our monolithic JSF page into four files: the original JSF page, `/header.jsp`, `/menu.jsp`, and `/content.jsp`. We include the header, menu, and content in the original JSF page:

```
<h:panelGrid columns="2" styleClass="book"
    columnClasses="menuColumn, contentColumn">

    <f:facet name="header">
        <f:subview id="header">
            <c:import url="header.jsp"/>
        </f:subview>
```

```
        </f:facet>

        <f:subview id="menu">
            <c:import url="menu.jsp"/>
        </f:subview>
                    <f:subview id="content">
            <c:import url="content.jsp"/>
        </f:subview>
    </h:panelGrid>
    ...
```

This code is much cleaner than the original JSF page listed in Listing 8–1 on page 322, so it's easier to understand, maintain, and modify. But more importantly, we are now free to reuse the header, menu, and content for other views. For example, to use the book-viewer with another book, all we have to do is change the book's `titleKey`, `image`, and `numChapters` properties in `faces-config.xml`.

The directory structure for the book viewer with includes example is shown in Figure 8–5. Listing 8–6 through Listing 8–9 show the JSF pages for the book, its header, menu, and content.

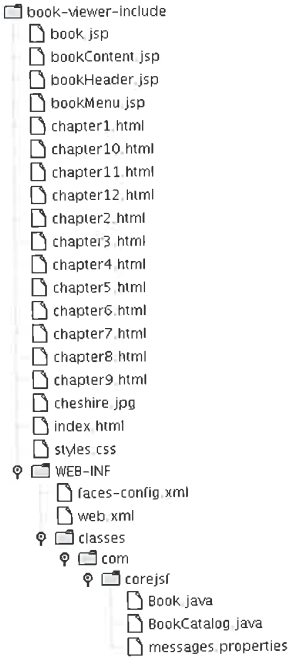


Figure 8–5 The Directory Structure of the Book Viewer with Includes

Listing 8–6 book-viewer-include/book.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/core"  prefix="f" %>
4.    <%@ taglib uri="http://java.sun.com/jsf/html"  prefix="h" %>
5.    <f:view>
6.       <head>
7.          <link href="styles.css" rel="stylesheet" type="text/css"/>
8.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
9.          <title><h:outputText value="#{msgs.bookWindowTitle}"/></title>
10.      </head>
11.      <body>
12.         <h:form>
13.            <h:panelGrid columns="2" styleClass="book"
14.                   columnClasses="menuColumn, chapterColumn">
15.               <f:facet name="header">
16.                  <f:subview id="header">
17.                     <c:import url="/bookHeader.jsp"/>
18.                  </f:subview>
19.               </f:facet>
20.
21.               <f:subview id="menu">
22.                  <c:import url="/bookMenu.jsp"/>
23.               </f:subview>
24.
25.               <f:verbatim>
26.                  <c:import url="/bookContent.jsp"/>
27.               </f:verbatim>
28.            </h:panelGrid>
29.         </h:form>
30.      </body>
31.   </f:view>
32. </html>
```

Listing 8–7 book-viewer-include/bookHeader.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
3.
4. <h:panelGrid columns="1" styleClass="bookHeader">
5.    <h:graphicImage value="#{book.image}"/>
6.    <h:outputText value="#{msgs[book.titleKey]}" styleClass="bookTitle"/>
7.    <f:verbatim><hr></f:verbatim>
8. </h:panelGrid>
```


Listing 8–8 book-viewer-include/bookMenu.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
3.
4. <h:dataTable value="#{book.chapterKeys}" var="chapterKey"
5.          styleClass="links" columnClasses="linksColumn">
6.    <h:column>
7.       <h:commandLink>
8.          <h:outputText value="#{msgs[chapterKey]}"/>
9.          <f:param name="chapter" value="#{chapterKey}"/>
10.      </h:commandLink>
11.   </h:column>
12. </h:dataTable>
```

Listing 8–9 book-viewer-include/bookContent.jsp

```
1. <%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
2.
3. <c:import url="${param.chapter}.html"/>
```

Looking at Tiles

We've seen how to encapsulate and include content and how that strategy increases flexibility—it's much easier to reuse content if you include it rather than mixing it all in one file. Now that you can create user interfaces with pluggable content, you may be satisfied with that level of flexibility and reuse; but wait, there's more.

In addition to *encapsulating content*, you can use Tiles to *encapsulate layout*. For the application shown in Figure 8–2 on page 316, encapsulating layout means making the layout code—the `h:panelGrid` and its contents listed in Listing 8–6 on page 329—available for reuse. As it stands in Listing 8–6, that layout code can only be used by the JSF page shown in Figure 8–2. If you implement JSF pages with identical layouts, you must *replicate that layout code for every page*. With Tiles, you define a layout that can be reused by multiple *tiles*, which are nothing more mysterious than imported JSP pages. *Tiles lets you implement layout code once and reuse it among many pages.*

But reusing layout is just the beginning of the Tiles bag of tricks. You can do more:

- Nest tiles.
- Extend tiles.

- Restrict tiles to users of a particular role.
- Attach controllers (Java objects) to tiles that are invoked just before their tile is displayed.

Those are the core features that Tiles offers in the pursuit of the ultimate flexibility in crafting web-based user interfaces.

Installing Tiles

Tiles is distributed only with Struts 1.1, but it doesn't depend on Struts at all, so you can use it standalone or with other web application frameworks, such as JSF. Because Tiles comes with Struts, you must download Struts 1.1 from http://jakarta.apache.org/site/binindex.cgi.

Here is how you install Tiles:

1. Download Struts 1.1 from http://jakarta.apache.org/site/binindex.cgi.
2. Copy the following JAR files from `$STRUTS_HOME/lib` to `/WEB-INF/lib`: `struts.jar`, `commons-beanutils.jar`, `commons-collections.jar`, and `commons-digester.jar`
3. Add the Tiles servlet to your deployment descriptor (`web.xml`).
4. Set the Tiles configuration file to `/WEB-INF/tiles.xml` in `web.xml`.

Your deployment descriptor should look similar to the one listed in Listing 8–10.

Listing 8–10 /WEB-INF/web.xml

```
1. <?xml version="1.0"?>
2. <!DOCTYPE web-app PUBLIC
3.    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4.    "http://java.sun.com/dtd/web-app_2_3.dtd">
5.
6. <web-app>
7.    <servlet>
8.       <servlet-name>Tiles Servlet</servlet-name>
9.       <servlet-class>org.apache.struts.tiles.TilesServlet</servlet-class>
10.      <init-param>
11.         <param-name>definitions-config</param-name>
12.         <param-value>/WEB-INF/tiles.xml</param-value>
13.      </init-param>
14.      <load-on-startup>2</load-on-startup>
15.   </servlet>
16.
17.   <servlet>
18.      <servlet-name>Faces Servlet</servlet-name>
```

Listing 8–10 /WEB-INF/web.xml (cont.)

```
19.        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
20.        <load-on-startup>1</load-on-startup>
21.     </servlet>
22.
23.     <servlet-mapping>
24.        <servlet-name>Faces Servlet</servlet-name>
25.        <url-pattern>*.faces</url-pattern>
26.     </servlet-mapping>
27.
28.     <welcome-file-list>
29.        <welcome-file>index.html</welcome-file>
30.     </welcome-file-list>
31. </web-app>
```

Notice that you must load the Tiles servlet when your application starts. You do that with the `load-on-startup` element, as we did in the preceding listing.

The `definitions-config` initialization parameter for the Tiles servlet specifies either a single configuration file—as we did in the preceding listing—or a comma-separated list of configuration files. Those configuration files contain your tile definitions. You can name those files anything you want as long as they end in `.xml`. In Listing 8–10 we specified a single file in `/WEB-INF` named `tiles.xml`. The following section shows you what to put in your Tiles configuration files.

Using Tiles with the Book Viewer

Using Tiles with JSF is a simple three-step process:

1. Use `tiles:insert` to insert a tile definition in a JSF page.
2. Define the tile in your Tiles configuration file.
3. Implement the tile's layout.

For the book viewer, we start in `book.jsp`, where we insert a tile named book:

```
...
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles"
    prefix="tiles" %>
...
<h:form>
    <tiles:insert definition="book" flush="false"/>
</h:form>
...
```

We define the book tile in /WEB-INF/tiles.xml:

```
<definition name="book" path="/headerMenuContentLayout.jsp">
      <put name="header"  value="/bookHeader.jsp"/>
      <put name="menu" value="/bookMenu.jsp"/>
      <put name="content" value="/bookContent.jsp"/>
</definition>
```

The previous snippet of XML defines a tile. The tile's layout is specified with the definition element's path attribute. The tile attributes, specified with put elements, are used by the layout. That layout looks like this:

```
<%-- this is /headerMenuContentLayout.jsp --%>

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles"%>

<h:panelGrid columns="2" styleClass="gridClass"
    headerClass="headerClass"
    columnClasses="menuClass, contentClass">

   <f:facet name="header">
      <f:subview id="header">
        <tiles:insert attribute="header" flush="false"/>
      </f:subview>
   </f:facet>

   <f:subview id="menu">
      <tiles:insert attribute="menu" flush="false"/>
   </f:subview>

   <f:subview id="content">
      <tiles:insert attribute="content" flush="false"/>
   </f:subview>
</h:panelGrid>
```

The tiles:insert tag dynamically includes content. That content is the value of the attribute tag of tiles:insert. For example, the preceding code inserts the header attribute. That attribute's value is /header.jsp, so tiles:insert dynamically includes that file.

Notice that we specified a flush="false" attribute for the tiles:insert tag. That is necessary for most modern servlet containers because those containers disallow buffer flushing inside custom tags. If your servlet container throws an exception stating that you cannot flush from a custom tag, then you know you've forgotten to specify that attribute, which is true by default.

What have we gained by using Tiles in this example? *We've encapsulated layout so that we can reuse it in other tiles, instead of replicating that layout code from one JSF page to another.* For example, you could reuse the book viewer's layout, implemented in `/headerMenuContentLayout.jsp`, for other pages in the application that have the same layout.

Parameterizing Tiles

There's one flaw to the layout listed in the previous section: it hardcodes CSS classes, namely `gridClass`, `headerClass`, `menuClass`, and `contentClass`. That means that every web page using the header-menu-content layout will have the same look and feel. It would be better if we could parameterize the CSS class names. That way, other tiles with a header-menu-content layout could define their own look and feel. Let's see how we can do that. First, we add three attributes to the book tile:

```
<definition name="book" path="/headerMenuContentLayout.jsp">
    <put name="headerClass"  value="headerClass"/>
    <put name="menuClass" value="menuClass"/>
    <put name="contentClass" value="contentClass"/>

    <put name="header" value="/bookHeader.jsp"/>
    <put name="menu" value="/bookMenu.jsp"/>
    <put name="content" value="/bookContent.jsp"/>
</definition>
```

Then we use those attributes in the layout:

```
<%-- this is an excerpt of /headerMenuContentLayout.jsp --%>
...
<tiles:importAttribute scope="request"/>

<h:panelGrid columns="2" styleClass="#{gridClass}"
    headerClass="#{headerClass}"
    columnClasses="#{menuClass}, #{contentClass}">
    ...
</h:panelGrid>
```

Tile attributes, such as `headerClass`, `menuClass`, etc., in the preceding code, exist in *tiles scope*, which is inaccessible to JavaServer Faces. To make our attributes accessible to the layout JSF page listed above, we use the `tiles:importAttribute` tag. That tag imports all tile attributes to the scope you specify with the `scope` attribute. In the preceding code, we imported them to request scope.

Now we can specify different CSS classes for other tiles:


```
<definition name="anotherTile" path="/headerMenuContentLayout.jsp">
      <put name="headerClass"  value="aDifferentHeaderClass"/>
      ...
</definition>
```

NOTE: The `tiles:importAttribute` tag also lets you import one attribute at a time; for example: `<tiles:importAttribute name="headerClass" scope="..."/>`.

Extending Tiles

In "Parameterizing Tiles" on page 334 we defined a tile that looked like this:

```
<definition name="book" path="/headerMenuContentLayout.jsp">
    <put name="headerClass"  value="headerClass"/>
    <put name="menuClass" value="menuClass"/>
    <put name="contentClass" value="contentClass"/>

    <put name="header"  value="/bookHeader.jsp"/>
    <put name="menu" value="/bookMenu.jsp"/>
    <put name="content" value="/bookContent.jsp"/>
</definition>
```

There are two distinct types of attributes in that tile: CSS classes and included content. Although the latter is specific to the book tile, the former can be used by tiles that represent something other than books. Because of that generality, we split the book tile into two:

```
<definition name="header-menu-content" path="/headerMenuContentLayout.jsp">
    <put name="headerClass"  value="headerClass"/>
    <put name="menuClass" value="menuClass"/>
    <put name="contentClass" value="contentClass"/>
</definition

<definition name="book" extends="header-menu-content">
    <put name="header"  value="/bookHeader.jsp"/>
    <put name="menu" value="/bookMenu.jsp"/>
    <put name="content" value="/bookContent.jsp"/>
</definition>
```

Now the `book` tile *extends* the `header-menu-content` tile. When you extend a tile, you inherit its attributes, much the same as object-oriented inheritance. Because of that inheritance, both `book` tile definitions in this section have the same attributes. But now, part of the original `book` tile—the CSS class attributes—are available for reuse by other tiles that extend the `header-menu-content` tile.

NOTE: Here's one more thing to consider about Tiles. Imagine the book viewer has been a huge success and Project Gutenberg has commissioned you to implement a library that can display all 6,000+ of their books. You define more than 6,000 tiles that reuse the same layout—one tile for each book—and present your finished product to the folks at Gutenberg. They think it's great, but they want you to add a footer to the bottom of the page. Since you've used Tiles, you only need to change the single layout used by all your tiles. Imagine the difficulty you would encounter making that change if you had replicated the layout code more than 6,000 times!

Figure 8–6 shows the directory structure for the "tileized" version of the book viewer. That directory structure is the same as the previous version of the book viewer, except that we've added a layout—`headerMenuContentLayout.jsp`—and the tiles definition file, `/WEB-INF/tiles.xml`.

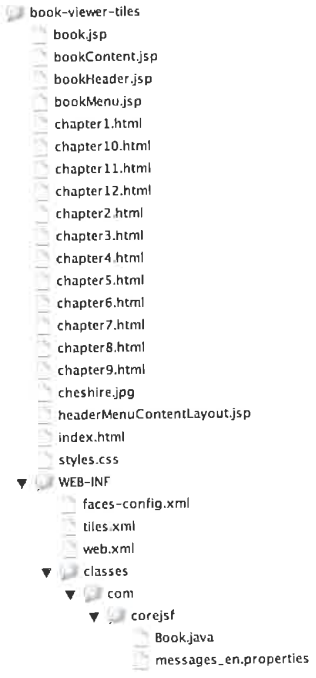


Figure 8–6 Book Viewer with Extended Tile Directory Structure

Listing 8–11 through Listing 8–13 show the Tiles definition file, the book layout, and the JSF page that displays Alice in Wonderland. We left out the listings of the other files in the application because they are unchanged from the application discussed in "Content Inclusion in JSP-Based Applications" on page 326.

Listing 8–11 book-viewer-tiles/WEB-INF/tiles.xml

```
1. <!DOCTYPE tiles-definitions PUBLIC
2.  "-//Apache Software Foundation//DTD Tiles Configuration//EN"
3.  "http://jakarta.apache.org/struts/dtds/tiles-config.dtd">
4.
5. <tiles-definitions>
6.    <definition name="menu-header-content" path="/headerMenuContentLayout.jsp">
7.       <put name="gridClass"           value="headerMenuContent"/>
8.       <put name="headerClass"         value="header"/>
9.       <put name="menuColumnClass"     value="menuColumn"/>
10.      <put name="contentColumnClass"  value="contentColumn"/>
11.   </definition>
12.
13.   <definition name="book" extends="menu-header-content">
14.      <put name="header"  value="/bookHeader.jsp"/>
15.      <put name="menu"    value="/bookMenu.jsp"/>
16.      <put name="content" value="/bookContent.jsp"/>
17.   </definition>
18. </tiles-definitions>
```

Listing 8–12 book-viewer-tiles/headerMenuContentLayout.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"  prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"  prefix="h" %>
3. <%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>
4.
5. <tiles:importAttribute scope="request"/>
6.
7. <h:panelGrid columns="2" styleClass="#{gridClass}"
8.    headerClass="#{headerClass}"
9.       columnClasses="#{menuColumnClass}, #{contentColumnClass}">
10.   <f:facet name="header">
11.      <f:subview id="header">
12.         <tiles:insert attribute="header" flush="false"/>
13.      </f:subview>
14.   </f:facet>
15.   <f:subview id="menu">
16.      <tiles:insert attribute="menu" flush="false"/>
```

Listing 8–12 book-viewer-tiles/headerMenuContentLayout.jsp (cont.)

```
17.    </f:subview>
18.
19.    <f:verbatim>
20.       <tiles:insert attribute="content" flush="false"/>
21.    </f:verbatim>
22. </h:panelGrid>
```

Listing 8–13 book-viewer-tiles/book.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/core"  prefix="f" %>
4.    <%@ taglib uri="http://java.sun.com/jsf/html"  prefix="h" %>
5.    <%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>
6.
7.    <f:view>
8.       <head>
9.          <link href="styles.css" rel="stylesheet" type="text/css"/>
10.         <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
11.         <title><h:outputText value="#{msgs.bookWindowTitle}"/></title>
12.      </head>
13.      <body>
14.         <f:subview id="book">
15.            <h:form>
16.               <tiles:insert definition="book" flush="false"/>
17.            </h:form>
18.         </f:subview>
19.      </body>
20.   </f:view>
21. </html>
```

The Library

In this section, we turn the book viewer into a library, shown in Figure 8–7. The library application shown in Figure 8–7 contains a menu at the top of the page that lets you select a book, either Alice in Wonderland or Peter Pan. The rest of the application works like the book viewer we've discussed throughout this chapter.

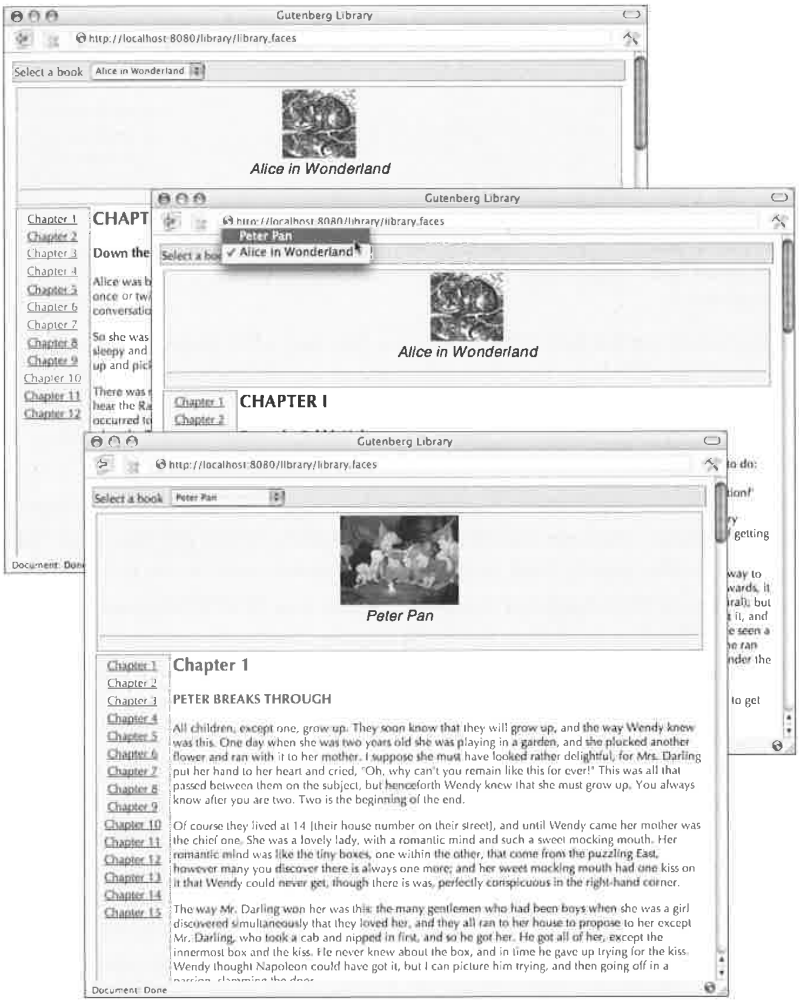


Figure 8–7 Library Implemented with JSF and Tiles

The library employs two Tiles techniques that are of interest to us: nesting tiles and using tile controllers.

Nested Tiles

The library shown in Figure 8–7 contains a book viewer. So does the library tile:

```
<definition name="book">
    ...
</definition>

<definition name="library" path="/libraryLayout.jsp"
      controllerClass="com.corejsf.LibraryTileController">
   <put name="header"  value="/bookSelector.jsp"/>
   <put name="book" value="book"/>
</definition>
```

Notice the value for the `book` attribute—it's a tile, not a JSP page. Using a tile name instead of a JSP page lets you nest tiles, as we did by nesting the `book` tile in the library.

Tile Controllers

In our book viewer application, we had one managed bean named `book`—see "The Book Viewer" on page 318 for more information about the `book` bean. The library, on the other hand, must be aware of more than one book.

In this section—with a sleight of hand—we show you how to support multiple books without having to change the book viewer. The book viewer will continue to manipulate a `book` bean, but that bean will no longer be a managed bean. Instead, it will be the book that was last selected in the library's pull-down menu at the top of the page.

We accomplish that sleight of hand with a Tiles controller. Tiles lets you attach a Java object, known as a tile controller, to a tile. That object's class must implement the `org.apache.struts.tiles.Controller` interface, which defines a single `perform` method. Tiles invokes that method just before it loads the controller's associated tile. Tile controllers have access to their tile's context, which lets the controller access the tile's attributes or create new attributes.

We attach a controller to the `library` tile. The controller looks for a `library` attribute in session scope. If the library's not there, the controller creates a library and stores it in session scope. The controller then consults the library's `selectedBook` property to see if a book has been selected. If so, the controller sets the value of the `book` session attribute to the selected book. If there is no selected book, the controller sets the `book` attribute to Alice in Wonderland. Subsequently, when the `library` tile is loaded, the book viewer accesses the selected book. The controller is listed in Listing 8–19 on page 345.

Figure 8–8 shows the directory structure for the library application. For brevity, we left out the book HTML files.

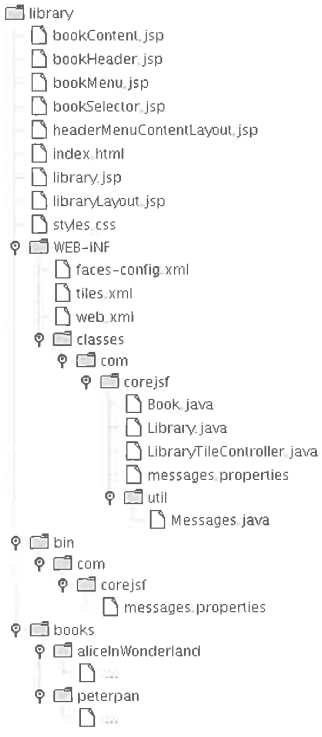


Figure 8–8 Library Directory Structure

The files shown in Figure 8–8 are listed in Listing 8–14 through Listing 8–27, with the exception of the HTML files. As you look through those listings, note the effort required to add a new book. All you have to do is modify the constructor in `Library.java`—see Listing 8–18 on page 343—to create your book and add it to the book map. You could even implement the `Library` class so that it reads XML book definitions. That way, you could add books without any programming. Digesting XML is an easy task with Tiles's distant cousin, the Apache Commons Digester. See http://jakarta.apache.org/commons/digester/ for more information about the Digester.

Listing 8–14 library/library.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core"  prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html"  prefix="h" %>
4.    <%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>
5.
6.    <f:view>
7.       <head>
8.          <link href="styles.css" rel="stylesheet" type="text/css"/>
9.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
10.         <title><h:outputText value="#{msgs.libraryWindowTitle}"/></title>
11.      </head>
12.      <body>
13.         <f:subview id="library">
14.            <h:form>
15.               <tiles:insert definition="library" flush="false"/>
16.            </h:form>
17.         </f:subview>
18.      </body>
19.   </f:view>
20. </html>
```

Listing 8–15 library/WEB-INF/tiles.xml

```
1. <!DOCTYPE tiles-definitions PUBLIC
2.  "-//Apache Software Foundation//DTD Tiles Configuration//EN"
3.  "http://jakarta.apache.org/struts/dtds/tiles-config.dtd">
4.
5. <tiles-definitions>
6.    <definition name="menu-header-content" path="/headerMenuContentLayout.jsp">
7.       <put name="gridClass"           value="headerMenuContent"/>
8.       <put name="headerClass"         value="header"/>
9.       <put name="menuColumnClass"     value="menuColumn"/>
10.      <put name="contentColumnClass"  value="contentColumn"/>
11.   </definition>
12.
13.   <definition name="book" extends="menu-header-content">
14.      <put name="header"  value="/bookHeader.jsp"/>
15.      <put name="menu"    value="/bookMenu.jsp"/>
16.      <put name="content" value="/bookContent.jsp"/>
17.   </definition>
18.
19.   <definition name="library" path="/libraryLayout.jsp"
20.                  controllerClass="com.corejsf.LibraryTileController">
21.      <put name="header"  value="/bookSelector.jsp"/>
22.      <put name="book" value="book"/>
23.   </definition>
24. </tiles-definitions>
```


Listing 8-16 library/libraryLayout.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"  prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"  prefix="h" %>
3. <%@ taglib uri="http://jakarta.apache.org/struts/tags-tiles" prefix="tiles" %>
4.
5. <h:panelGrid columns="1" styleClass="book" headerClass="libraryHeader">
6.    <f:facet name="header">
7.       <f:subview id="header">
8.          <tiles:insert attribute="header" flush="false"/>
9.       </f:subview>
10.   </f:facet>
11.
12.   <f:subview id="book">
13.      <tiles:insert attribute="book" flush="false"/>
14.   </f:subview>
15. </h:panelGrid>
```

Listing 8-17 library/bookSelector.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
3.
4. <h:outputText value="#{msgs.selectABookPrompt}"/>
5.
6. <f:verbatim>  </f:verbatim>
7.
8. <h:selectOneMenu onchange="submit()" value="#{library.book}"
9.       valueChangeListener="#{library.bookSelected}">
10.   <f:selectItems value="#{library.bookItems}"/>
11. </h:selectOneMenu>
```

Listing 8-18 library/WEB-INF/classes/com/corejsf/Library.java

```
1. package com.corejsf;
2.
3. import java.util.*;
4. import javax.faces.model.SelectItem;
5. import javax.faces.event.ValueChangeEvent;
6.
7. public class Library {
8.    private Map bookMap = new HashMap();
9.    private Book initialBook = null;
10.   private List bookItems = null;
11.   private String book = null;
12.   private String selectedBook = null;
13.
14.   public Library() {
```


Listing 8–18 library/WEB-INF/classes/com/corejsf/Library.java (cont.)

```
15.         Book peterpan = new Book();
16.         Book aliceInWonderland = new Book();
17.
18.         initialBook = peterpan;
19.
20.         aliceInWonderland.setDirectory("books/aliceInWonderland");
21.         aliceInWonderland.setTitleKey("aliceInWonderland");
22.         aliceInWonderland.setImage("books/aliceInWonderland/cheshire.jpg");
23.         aliceInWonderland.setNumChapters(12);
24.
25.         peterpan.setDirectory("books/peterpan");
26.         peterpan.setTitleKey("peterpan");
27.         peterpan.setImage("books/peterpan/peterpan.jpg");
28.         peterpan.setNumChapters(15);
29.
30.         bookMap.put("aliceInWonderland", aliceInWonderland);
31.         bookMap.put("peterpan", peterpan);
32.     }
33.     public void setBook(String book) { this.book = book; }
34.     public String getBook() { return book; }
35.
36.     public Map getBooks() {
37.         return bookMap;
38.     }
39.     public void bookSelected(ValueChangeEvent e) {
40.         selectedBook = (String) e.getNewValue();
41.     }
42.     public Book getSelectedBook() {
43.         return selectedBook != null ? (Book) bookMap.get(selectedBook) :
44.                                                 initialBook;
45.     }
46.     public List getBookItems() {
47.         if(bookItems == null) {
48.             bookItems = new LinkedList();
49.             Iterator it = bookMap.values().iterator();
50.             while(it.hasNext()) {
51.                 Book book = (Book)it.next();
52.                 bookItems.add(new SelectItem(book.getTitleKey(),
53.                                             getBookTitle(book.getTitleKey())));
54.             }
55.         }
56.         return bookItems;
57.     }
58.     private String getBookTitle(String key) {
59.         return com.corejsf.util.Messages.
60.                         getString("com.corejsf.messages", key, null);
61.     }
62. }
```


Listing 8–19 library/WEB-INF/classes/com/corejsf/LibraryTileController.java

```
1. package com.corejsf;
2.
3.
4. import java.io.IOException;
5. import javax.servlet.ServletContext;
6. import javax.servlet.ServletException;
7. import javax.servlet.http.HttpServletRequest;
8. import javax.servlet.http.HttpServletResponse;
9. import javax.servlet.http.HttpSession;
10. import org.apache.struts.tiles.ComponentContext;
11. import org.apache.struts.tiles.Controller;
12.
13. public class LibraryTileController implements Controller {
14.    public void perform(ComponentContext tilesContext,
15.                        HttpServletRequest request, HttpServletResponse response,
16.                        ServletContext context)
17.                                     throws IOException, ServletException {
18.       HttpSession session = request.getSession();
19.
20.       String chapter = (String) request.getParameter("chapter");
21.       session.setAttribute("chapter", chapter == null || "".equals(chapter) ?
22.                            "chapter1" : chapter);
23.
24.       Library library = (Library) session.getAttribute("library");
25.
26.       if(library == null) {
27.          library = new Library();
28.          session.setAttribute("library", library);
29.       }
30.
31.       Book selectedBook = library.getSelectedBook();
32.       if(selectedBook != null) {
33.          session.setAttribute("book", selectedBook);
34.       }
35.    }
36. }
```

Listing 8–20 library/WEB-INF/classes/com/corejsf/Book.java

```
1. package com.corejsf;
2.
3. import java.util.LinkedList;
4. import java.util.List;
5.
6. public class Book {
7.    private String titleKey;
```

Listing 8-20 library/WEB-INF/classes/com/corejsf/Book.java (cont.)

```
8.    private String image;
9.    private String directory;
10.   private int numChapters;
11.   private List chapterKeys = null;
12.
13.   // PROPERTY: titleKey
14.   public void setTitleKey(String titleKey) { this.titleKey = titleKey; }
15.   public String getTitleKey() { return titleKey; }
16.
17.   // PROPERTY: image
18.   public void setImage(String image) { this.image = image; }
19.   public String getImage() { return image; }
20.
21.   // PROPERTY: directory
22.   public void setDirectory(String directory) { this.directory = directory; }
23.   public String getDirectory() { return directory; }
24.
25.   // PROPERTY: numChapters
26.   public void setNumChapters(int numChapters) { this.numChapters = numChapters; }
27.   public int getNumChapters() { return numChapters; }
28.
29.   // PROPERTY: chapterKeys
30.   public List getChapterKeys() {
31.      if(chapterKeys == null) {
32.         chapterKeys = new LinkedList();
33.         for(int i=1; i <= numChapters; ++i)
34.            chapterKeys.add("chapter" + i);
35.      }
36.      return chapterKeys;
37.   }
38. }
```

Listing 8-21 library/bookHeader.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
3.
4. <h:panelGrid columns="1" styleClass="bookHeader">
5.    <h:graphicImage value="#{book.image}"/>
6.    <h:outputText value="#{msgs[book.titleKey]}" styleClass="bookTitle"/>
7.    <f:verbatim><hr></f:verbatim>
8. </h:panelGrid>
```


Listing 8–22 library/bookMenu.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
3.
4. <h:dataTable value="#{book.chapterKeys}" var="chapterKey"
5.         styleClass="links" columnClasses="linksColumn">
6.    <h:column>
7.       <h:commandLink>
8.          <h:outputText value="#{msgs[chapterKey]}"/>
9.          <f:param name="chapter" value="#{chapterKey}"/>
10.      </h:commandLink>
11.   </h:column>
12. </h:dataTable>
```

Listing 8–23 library/bookContent.jsp

```
1. <%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
2.
3. <c:import url="${book.directory}/${chapter}.html"/>
```

Listing 8–24 library/styles.css

```
1. .library {
2.    vertical-align: top;
3.    width: 100%;
4.    height: 100%;
5. }
6. .libraryHeader {
7.    width: 100%;
8.    text-align: left;
9.    vertical-align: top;
10.   background-color: #ddd;
11.   font-weight: lighter;
12.   border: thin solid #777;
13. }
14. .bookHeader {
15.   width: 100%;
16.   text-align: center;
17.   background-color: #eee;
18.   border: thin solid CornflowerBlue;
19. }
20. .bookTitle {
21.   text-align: center;
22.   font-style: italic;
23.   font-size: 1.3em;
24.   font-family: Helvetica;
```


Listing 8–24 library/styles.css (cont.)

```
25. }
26. .menuColumn {
27.     vertical-align: top;
28.     background-color: #eee;
29.     border: thin solid #777;
30. }
31. .chapterColumn {
32.     vertical-align: top;
33.     text-align: left;
34.     width: *;
35.     padding: 3px;
36. }
37. .contentColumn {
38.     vertical-align: top;
39.     text-align: left;
40.     width: *;
41. }
42. .links {
43.     width: 85px;
44.     vertical-align: top;
45.     text-align: center;
46. }
47. .linksColumn {
48.     vertical-align: top;
49.     text-align: center;
50. }
```

Listing 8–25 library/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2.
3. <!DOCTYPE faces-config PUBLIC
4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
6.
7. <faces-config>
8.
9. </faces-config>
```

Listing 8–26 library/WEB-INF/web.xml

```
1. <?xml version="1.0"?>
2. <!DOCTYPE web-app PUBLIC
3.     "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4.     "http://java.sun.com/dtd/web-app_2_3.dtd">
5. <web-app>
6.     <servlet>
```


Listing 8–26 library/WEB-INF/web.xml (cont.)

```
7.      <servlet-name>Tiles Servlet</servlet-name>
8.      <servlet-class>org.apache.struts.tiles.TilesServlet</servlet-class>
9.      <init-param>
10.        <param-name>definitions-config</param-name>
11.        <param-value>/WEB-INF/tiles.xml</param-value>
12.     </init-param>
13.     <load-on-startup>2</load-on-startup>
14.  </servlet>
15.
16.  <servlet>
17.     <servlet-name>Faces Servlet</servlet-name>
18.     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
19.     <load-on-startup>1</load-on-startup>
20.  </servlet>
21.
22.  <servlet-mapping>
23.     <servlet-name>Faces Servlet</servlet-name>
24.     <url-pattern>*.faces</url-pattern>
25.  </servlet-mapping>
26.
27.  <welcome-file-list>
28.     <welcome-file>index.html</welcome-file>
29.  </welcome-file-list>
30. </web-app>
```

Listing 8–27 library/classes/com/corejsf/messages.properties

```
1. libraryWindowTitle=Gutenberg Library
2. aliceInWonderland=Alice in Wonderland
3. peterpan=Peter Pan
4. selectABookPrompt=Select a book
5.
6. chapter1=Chapter 1
7. chapter2=Chapter 2
8. chapter3=Chapter 3
9. chapter4=Chapter 4
10. chapter5=Chapter 5
11. chapter6=Chapter 6
12. chapter7=Chapter 7
13. chapter8=Chapter 8
14. chapter9=Chapter 9
15. chapter10=Chapter 10
16. chapter11=Chapter 11
17. chapter12=Chapter 12
18. chapter13=Chapter 13
19. chapter14=Chapter 14
20. chapter15=Chapter 15
```


Custom Components

Topics in This Chapter

Chapter 9

JSF provides a basic set of components for building HTML-based web applications such as text fields, checkboxes, buttons, and so on. However, most user-interface designers will desire more advanced components, such as calendars, tabbed panes, or navigation trees, that are not part of the standard JSF component set. Fortunately, JSF makes it possible to build reusable JSF components with rich behavior.

This chapter shows you how to implement custom components. We use two custom components—a tabbed pane and a spinner, shown in Figure 9–1—to illustrate the various aspects of creating custom components.

Figure 9–1 The TabbedPane and the Spinner

The JSF API lets you implement custom components and associated tags with the same features as the JSF standard tags. For example, `h:input` uses a value binding to associate a text field's value with a bean property—you could use value bindings to wire calendar cells to bean properties. JSF standard input components fire value change events when their value changes—you could fire value change events when a different date is selected in the calendar.

The first part of this chapter uses the spinner component to illustrate basic issues that you encounter in all custom components. We then revisit the spinner to show more advanced issues:

- "Using an External Renderer" on page 378
- "Calling Converters from External Renderers" on page 384
- "Supporting Value Change Listeners" on page 385
- "Supporting Method Bindings" on page 386

The second half of the chapter examines a tabbed pane component that illustrates the following aspects of custom component development.

- "Processing `SelectItem` Children" on page 406
- "Processing Facets" on page 407
- "Encoding CSS Styles" on page 410
- "Using Hidden Fields" on page 411
- "Saving and Restoring State" on page 412
- "Firing Action Events" on page 414

Implementing Custom Components with Classes

In the following sections, we discuss the classes that you need to implement custom components.

To motivate the discussion, we will develop a spinner component. A spinner lets you enter a number in a text field, either by typing it directly in the field or by activating an increment or decrement button. Figure 9–2 shows an application that uses two spinners for a credit card's expiration date, one for the month and another for the year.

In Figure 9–2, from top to bottom, all proceeds as expected. The user enters valid values so navigation takes us to a designated JSF page that echoes those values.

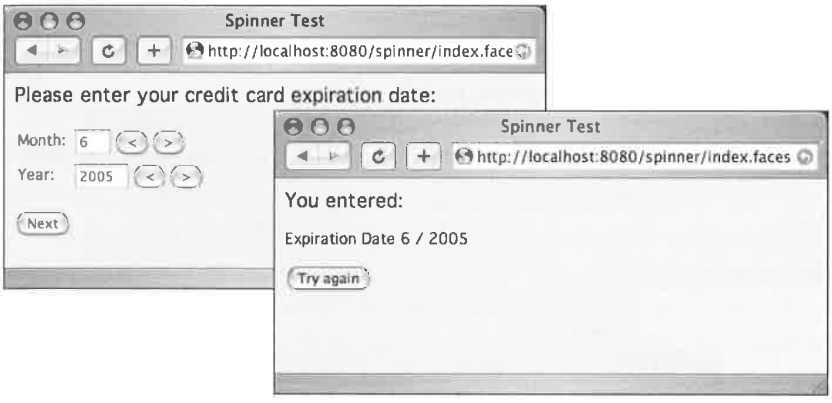


Figure 9–2 Using the Spinner Component

The spinner insists on integer values. Figure 9–3 shows an attempt to enter bad data. We let the standard integer converter handle conversion errors. You can see how we did it in "Using Converters" on page 364.

Here's how you use `corejsf:spinner`:

```
<%@ taglib uri="http://corejsf.com/spinner" prefix="corejsf" %>
...
<corejsf:spinner value="#{cardExpirationDate.month}"
   id="monthSpinner" minimum="1" maximum="12" size="3"/>
<h:message for="monthSpinner"/>
...
<corejsf:spinner value="#{cardExpirationDate.year}"
   id="yearSpinner" minimum="1900" maximum="2100" size="5"/>
<h:message for="yearSpinner"/>
```

The `corejsf:spinner` tag supports the following attributes.

- `binding`
- `id`
- `minimum`
- `maximum`
- `rendered`
- `size`
- `value`

Only one of the attributes—`value`—is required.

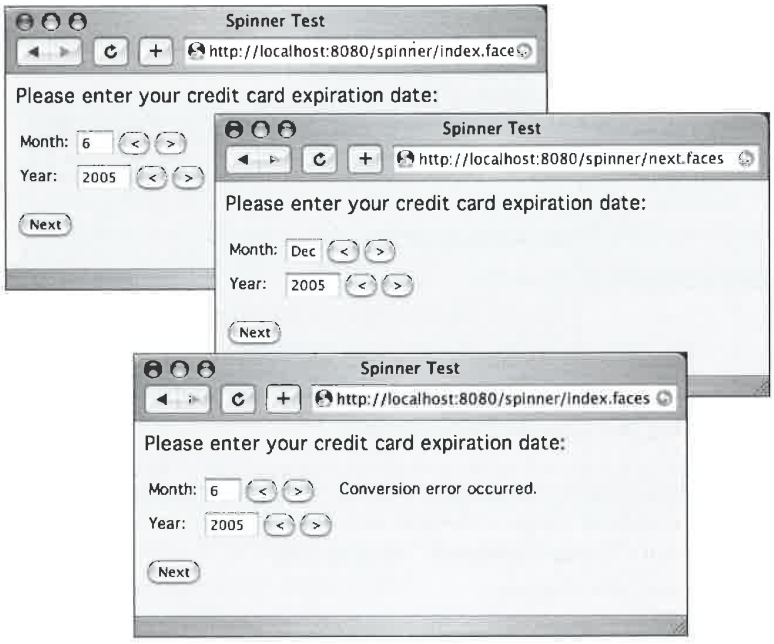


Figure 9–3 Handling Conversion Failures

The `minimum` and `maximum` attributes let you assign a range of valid values; for example, the month spinner has a minimum of 1 and a maximum of 12. You can also limit the size of the spinner's text field with the `size` attribute. The `value` attribute can take a literal string—for example, `value="2"`—or a value binding, for example, `value="#{someBean.someProperty}"`.

Finally, the spinner supports the `binding`, `id`, and `rendered` attributes, which are discussed in Chapter 4. Support for those attributes is free because our tag class extends the `javax.faces.webapp.UIComponentTag` class.

In the preceding code fragment we assigned explicit identifiers to our spinners with the `id` attribute. We did that so we could display conversion errors with `h:message`. The spinner component doesn't require users to specify an identifier. If an identifier is not specified, JSF generates one automatically.

Users of JSF custom tags need not understand how those tags are implemented. Users simply need to know the functionality of a tag and the set of available attributes. Just as for any component model, the expectation is that a few skilled programmers will create tags that can be used by many page developers.

Tags and Components

Minimally, a tag for a JSF custom component requires two classes:

- A class that processes tag attributes. By convention, the class name has a `Tag` suffix; for example, `SpinnerTag`.
- A component class that maintains state, renders a user interface, and processes input. By convention, the class name has a `UI` prefix; for example, `UISpinner`.

The tag class is part of the plumbing. It creates the component and transfers tag attribute values to component properties and attributes. The implementation of the tag class is largely mechanical. See "Implementing Custom Component Tags" on page 367 for more information on tag classes.

The UI class does the important work. It has two separate responsibilities:

- To *render* the user interface by encoding markup
- To *process user input* by decoding the current HTTP request

Component classes can delegate rendering and processing input to a separate renderer. By using different renderers, you can support multiple clients such as web browsers and cell phones. Initially, our spinner component will render itself, but in "Using an External Renderer" on page 378, we show you how to implement a separate renderer for the spinner.

A component's `UI` class must extend the `UIComponent` class. That interface defines 36 methods, so you will want to extend an existing class that implements the interface. You can choose from the classes shown in Figure 9–4.

Our `UISpinner` class will extend `UIInput`, which extends `UIOutput` and implements the `EditableValueHolder` interface. Our `UITabbedPane` will extend `UICommand`, which implements the `ActionSource` interface.

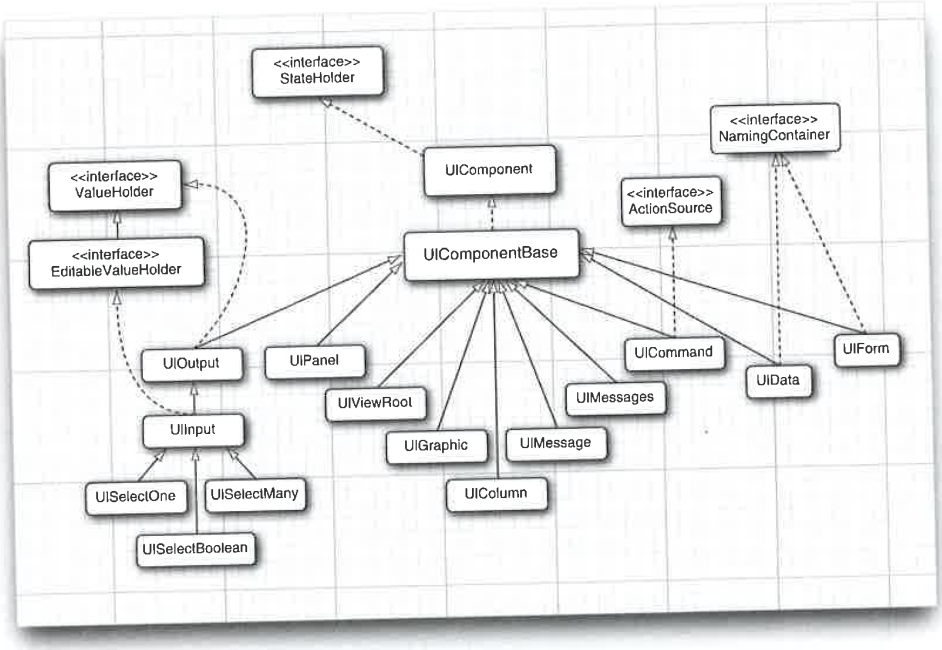


Figure 9–4 JSF Component Hierarchy (not all classes are shown)

The Custom Component Developer's Toolbox

When you implement custom components you will become very familiar with a handful of JSF classes:

- `javax.faces.component.UIComponent`
- `javax.faces.webapp.UIComponentTag`
- `javax.faces.context.FacesContext`
- `javax.faces.application.Application`
- `javax.faces.context.ResponseWriter`

`UIComponent` is an abstract class that defines what it means to be a component. Each component manages several important categories of data. These include:

- A list of *child components*. For example, the children of the `h:panelGrid` component are the components that are placed in the grid location. However, a component need not have any children.
- A map of *facet components*. Facets are similar to child components, but each facet has a key, not a position in a list. It is up to the component how

to lay out its facets. For example, the `h:dataTable` component has header and footer facets.

- A map of *attributes*. This is a general-purpose map that you can use to store arbitrary key/value pairs.
- A map of *value bindings*. This is another general-purpose map that you can use to store arbitrary value bindings. For example, if the spinner tag has an attribute `value="#{cardExpirationDate.month}"`, then the tag handler constructs a `ValueBinding` object for the given value binding expression and stores it under the key `"value"`.
- A collection of *listeners*. This collection is maintained by the JSF framework.

When you define your own JSF components, you usually subclass one of the following three standard component classes:

- `UICommand`, if your component produces actions similar to a command button or link.
- `UIOutput`, if your component displays a value but does not allow the user to edit it.
- `UIInput`, if your component reads a value from the user (such as the spinner).

If you look at Figure 9–4, you will find that these three classes implement interfaces that specify these distinct responsibilities:

- `ActionSource` defines methods for managing action listeners and actions.
- `ValueHolder` defines methods for managing a component value, a local value, and a converter.
- `EditableValueHolder` extends `ValueHolder` and adds methods for managing validators and value change listeners.

TIP: You often need to cast a generic `UIComponent` parameter to a subclass in order to access values, converters, and so on. Rather than casting to a specific class such as `UISpinner`, cast to an interface type, such as `ValueHolder`. That makes it easier to reuse your code.

`UIComponentTag` is a superclass for the tags you implement for your custom components. It implements mundane things—like support for `binding`, `id`, and `rendered` attributes—so you can concentrate on supporting the attributes your tags require.

The `FacesContext` class contains JSF-related request information. Among other things, you can access request parameters through `FacesContext`, get a reference

to the Application object, get the current view root component, or get a reference to the response writer, which you use to encode markup.

The Application class keeps track of objects shared by a single application—for example, the set of supported locales and available converters and validators. The Application class also serves as a factory, with factory methods for components, converters, validators, value bindings, and method bindings. In this chapter, we're mostly interested in using the Application class to create converters, value bindings, and method bindings.

Nearly all custom components generate markup, so you will want to use the ResponseWriter class to ease that task. Response writers have methods for starting and ending HTML elements and methods for writing element attributes.

We now return to the spinner implementation and view the spinner from a number of different perspectives. We start with every component's most basic tasks—generating markup and processing requests—and then turn to the more mundane issue of implementing the corresponding tag handler class.

Encoding: Generating Markup

JSF components generate markup for their user interfaces. By default, the standard JSF components generate HTML. Components can do their own encoding, or they can delegate encoding to a separate renderer. The latter is the more elegant approach because it lets you plug in different renderers, for example to encode markup in something other than HTML. However, for simplicity, we will start out with a spinner that renders itself.

Components encode markup with three methods:

- encodeBegin()
- encodeChildren()
- encodeEnd()

The methods are called by JSF at the end of the life cycle, in the order in which they are listed above. JSF invokes encodeChildren only if a component returns true from its getRendersChildren method. By default, getRendersChildren returns false for most components.

For simple components, like our spinner, that don't have children, you don't need to implement encodeChildren. The spinner also has no compelling reason for overriding both encodeBegin and encodeEnd, so we do all our encoding in encodeBegin.

The spinner generates HTML for a text field and two buttons; that HTML looks like this:

```
<input type="text" name="..." value="current value"/>
<input type="submit" name="..." value="<"/>
<input type="submit" name="..." value=">"/>
```


Here's how that HTML is encoded in UISpinner.

```
public class UISpinner extends UIInput {
   private static final String MORE = ".more";
   private static final String LESS = ".less";
   ...
   public void encodeBegin(FacesContext context) throws IOException {
      ResponseWriter writer = context.getResponseWriter();
      String clientId = getClientId(context);

      encodeInputField(writer, clientId);
      encodeDecrementButton(writer, clientId);
      encodeIncrementButton(writer, clientId);
   }
   private void encodeInputField(ResponseWriter writer, String clientId)
         throws IOException {
      writer.startElement("input", this);
      writer.writeAttribute("name", clientId, "clientId");

      Object v = getValue();
      if (v != null)
         writer.writeAttribute("value", v.toString(), "value");

      Integer size = (Integer) getAttributes().get("size");
      if (size != null) writer.writeAttribute("size", size, "size");

      writer.endElement("input");
   }
   private void encodeDecrementButton(ResponseWriter writer, String clientId)
         throws IOException {
      writer.startElement("input", this);
      writer.writeAttribute("type", "submit", null);
      writer.writeAttribute("name", clientId + LESS, null);
      writer.writeAttribute("value", "<", "value");
      writer.endElement("input");
   }
   private void encodeIncrementButton(ResponseWriter writer, String clientId)
         throws IOException {
      writer.startElement("input", this);
      writer.writeAttribute("type", "submit", null);
      writer.writeAttribute("name", clientId + MORE, null);
      writer.writeAttribute("value", ">", "value");
      writer.endElement("input");
   }
   ...
}
```

The ResponseWriter class has convenience methods for writing markup. The startElement and endElement methods produce the element delimiters. They keep track of child elements, so you don't have to worry about the distinction between <input .../> and <input ...>...</input>. The writeAttribute method writes an attribute name/value pair with the appropriate escape characters. The last parameter of the startElement and writeAttribute methods is intended for tool support, but it is currently unused. You are supposed to pass the rendered component object or attribute name, or null if the output doesn't directly correspond to a component or attribute.

UISpinner.encodeBegin faces two challenges. First, it must get the current state of the spinner. The numerical value is easily obtained with the getValue method that the spinner inherits from UIInput. The size is retrieved from the component's attribute map, using the getAttributes method. (As you will see in the section "Implementing Custom Component Tags" on page 367, the SpinnerTag class stores the tag's size attribute in the component's attribute map.)

Second, the encoding method needs to come up with names for the HTML elements the spinner encodes. It calls the getClientId method to obtain the client ID of the component, which is composed of the ID of the enclosing form and the ID of this component, such as _id1:monthSpinner. That identifier is created by the JSF implementation. The increment and decrement button names start with the client ID and end in .more and .less, respectively. Here is a complete example of the HTML generated by the spinner:

```
<input type="text" name="_id1:monthSpinner" value="1" size="3"/>
<input type="submit" name="_id1:monthSpinner.less" value="<"/>
<input type="submit" name="_id1:monthSpinner.more" value=">"/>
```

In the next section we discuss how those names are used by the spinner's decode method.

javax.faces.component.UIComponent

- void encodeBegin(FacesContext context) throws IOException

  JSF calls this method—in the "Render Response" phase of the JSF life cycle—only if the component's renderer type is null, signifying that the component renders itself.

- String getClientId(FacesContext context)

  Returns the client ID for this component. The JSF framework creates the client ID from the ID of the enclosing form (or, more generally, the enclosing *naming container*) and the ID of this component.

- `Map getAttributes()`

  Returns a mutable map of component attributes and properties. You use this method to view, add, update, or remove attributes from a component. You can also use this map to view or update properties. The map's `get` and `put` methods check whether the key matches a component property. If so, the property getter or setter is called.

NOTE: The spinner is a simple component with no children, so its encoding is rather basic. For a more complicated example, see how the tabbed pane renderer encodes markup. That renderer is shown in Listing 9–18 on page 414.

NOTE: JSF invokes a component's `encodeChildren` method if the component returns `true` from `getRendersChildren`. Interestingly, it doesn't matter whether the component actually has children—as long as the component's `getRendersChildren` method returns `true`, JSF calls `encodeChildren` even if the component has no children.

`javax.faces.context.FacesContext`

- `ResponseWriter getResponseWriter()`

  Returns a reference to the response writer. You can plug your own response writer into JSF if you want. By default, JSF uses a response writer that can write HTML tags.

`javax.faces.context.ResponseWriter`

- `void startElement(String elementName, UIComponent component)`

  Writes the start tag for the specified element. The `component` parameter lets tools associate a component and its markup. The 1.0 version of the JSF reference implementation ignores this attribute.
- `void endElement(String elementName)`

  Writes the end tag for the specified element.
- `void writeAttribute(String attributeName, String attributeValue, String componentProperty)`

  Writes an attribute and its value. This method must be called between calls to `startElement()` and `endElement()`. The `componentProperty` is the name of the component property that corresponds to the attribute. Its use is meant for tools. It is not supported by the 1.0 reference implementation.

Decoding: Processing Request Values

To understand the decoding process, keep in mind how a web application works. The server sends an HTML form to the browser. The browser sends back a POST request that consists of name/value pairs. That POST request is the only data that the server can use to interpret the user's actions inside the browser.

If the user clicks on the increment or decrement button, the ensuing POST request includes the names and values of *all* text fields, but only the name and value of the *clicked* button. For example, if the user clicks the month spinner's increment button in the application shown in Figure 9–1 on page 351, the following request parameters are transferred to the server from the browser:

Name	Value
_id1:monthSpinner	1
_id1:yearSpinner	12
_id1:monthSpinner.more	>

When our spinner decodes an HTTP request, it looks for the request parameter names that match its client ID and processes the associated values. The spinner's decode method is listed below.

```
public void decode(FacesContext context) {
   Map requestMap = context.getExternalContext().getRequestParameterMap();
   String clientId = getClientId(context);

   int increment;
   if (requestMap.containsKey(clientId + MORE)) increment = 1;
   else if (requestMap.containsKey(clientId + LESS)) increment = -1;
   else increment = 0;

   try {
      int submittedValue
         = Integer.parseInt((String) requestMap.get(clientId));
      int newValue = getIncrementedValue(submittedValue, increment);
      setSubmittedValue("" + newValue);
      setValid(true);
   }
   catch(NumberFormatException ex) {
      // let the converter take care of bad input, but we still have
      // to set the submitted value or the converter won't have
      // any input to deal with
      setSubmittedValue((String) requestMap.get(clientId));
   }
}
```


The decode method looks at the request parameters to determine which of the spinner's buttons, if any, triggered the request. If a request parameter named *clientId*.less exists, where *clientId* is the client ID of the spinner we're decoding, then we know the decrement button was activated. If the decode method finds a request parameter named *clientId*.more, then we know the increment button was activated. If neither parameter exists, we know the request was not initiated by the spinner, so we set the increment to zero. We still need to update the value—the user might have typed a value into the text field and clicked the "Next" button.

Our naming convention works for multiple spinners in a page because each spinner is encoded with the spinner component's client ID, which is guaranteed to be unique. If you have multiple spinners in a single page, each spinner component decodes its own request.

Once the decode method determines that one of the spinner's buttons was clicked, it increments the spinner's value by 1 or –1, depending on which button the user activated. That incremented value is calculated by a private getIncrementedValue method.

```
private int getIncrementedValue(int submittedValue, int increment) {
   Integer minimum = (Integer) getAttributes().get("minimum");
   Integer maximum = (Integer) getAttributes().get("maximum");
   int newValue = submittedValue + increment;

   if ((minimum == null || newValue >= minimum.intValue()) &&
      (maximum == null || newValue <= maximum.intValue()))
      return newValue;
   else
      return submittedValue;
}
```

The getIncrementedValue method checks the value the user entered in the spinner against the spinner's minimum and maximum attributes. Those attributes are set by the spinner's tag handler class.

After it gets the incremented value, the decode method calls the spinner component's setSubmittedValue method. That method stores the submitted value in the component. Subsequently, in the JSF life cycle, that submitted value will be converted and validated by the JSF framework.

CAUTION: You must call setValid(true) after setting the submitted value. Otherwise, the input is not considered valid, and the current page is simply redisplayed.

`javax.faces.component.UIComponent`

- `void decode(FacesContext context)`

  JSF calls this method—at the beginning of the JSF life cycle—only if the component's renderer type is `null`, signifying that the component renders itself.

  The `decode` method decodes request parameters. Typically, components transfer request parameter values to component properties or attributes. Components that fire action events queue them in this method.

`javax.faces.context.FacesContext`

- `ExternalContext getExternalContext()`

  Returns a reference to a context proxy. Typically, the real context is a servlet or portlet context. If you use the external context instead of using the real context directly, your applications can work with servlets and portlets.

`javax.faces.context.ExternalContext`

- `Map getRequestParameterMap()`

  Returns a map of request parameters. Custom components typically call this method in `decode()` to see if they were the component that triggered the request.

`javax.faces.component.EditableValueHolder`

- `void setSubmittedValue(Object submittedValue)`

  Sets a component's submitted value—input components have editable values, so `UIInput` implements the `EditableValueHolder` interface. The submitted value is the value the user entered, presumably in a web page. For HTML-based applications, that value is always a string, but the method accepts an `Object` reference in deference to other display technologies.

- `void setValid(boolean valid)`

  Custom components use this method to indicate their value's validity. If a component can't convert its value, it sets the `valid` property to `false`.

Using Converters

The spinner component uses the standard JSF integer converter to convert strings to `Integer` objects, and vice versa. The `UISpinner` constructor simply calls `setConverter`, like this:

```
public class UISpinner extends UIInput {
   ...
   public UISpinner() {
      setConverter(new IntegerConverter()); // to convert the submitted value
      setRendererType(null);                // this component renders itself
   }
```

The spinner's decode method traps invalid inputs in the NumberFormatException catch clause. However, instead of reporting the error, it simply sets the component's submitted value to the user input. Later on in the JSF life cycle, the standard integer converter will try to convert that value and will generate an appropriate error message for bad input.

Listing 9–1 contains the complete code for the UISpinner class.

Listing 9–1 spinner/WEB-INF/classes/com/corejsf/UISpinner.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
5. import javax.faces.component.UIInput;
6. import javax.faces.context.FacesContext;
7. import javax.faces.context.ResponseWriter;
8. import javax.faces.convert.IntegerConverter;
9.
10. public class UISpinner extends UIInput {
11.    private static final String MORE = ".more";
12.    private static final String LESS = ".less";
13.
14.    public UISpinner() {
15.       setConverter(new IntegerConverter()); // to convert the submitted value
16.       setRendererType(null);                // this component renders itself
17.    }
18.
19.    public void encodeBegin(FacesContext context) throws IOException {
20.       ResponseWriter writer = context.getResponseWriter();
21.       String clientId = getClientId(context);
22.
23.       encodeInputField(writer, clientId);
24.       encodeDecrementButton(writer, clientId);
25.       encodeIncrementButton(writer, clientId);
26.    }
27.
28. public void decode(FacesContext context) {
29.       Map requestMap = context.getExternalContext().getRequestParameterMap();
30.       String clientId = getClientId(context);
31.
```


Listing 9–1 spinner/WEB-INF/classes/com/corejsf/UISpinner.java (cont.)

```
32.     int increment;
33.     if (requestMap.containsKey(clientId + MORE)) increment = 1;
34.     else if(requestMap.containsKey(clientId + LESS)) increment = -1;
35.     else increment = 0;
36.
37.     try {
38.        int submittedValue
39.           = Integer.parseInt((String) requestMap.get(clientId));
40.
41.        int newValue = getIncrementedValue(submittedValue, increment);
42.        setSubmittedValue("" + newValue);
43.        setValid(true);
44.     }
45.     catch(NumberFormatException ex) {
46.        // let the converter take care of bad input, but we still have
47.        // to set the submitted value, or the converter won't have
48.        // any input to deal with
49.        setSubmittedValue((String) requestMap.get(clientId));
50.     }
51.  }
52.
53.  private void encodeInputField(ResponseWriter writer, String clientId)
54.        throws IOException {
55.     writer.startElement("input", this);
56.     writer.writeAttribute("name", clientId, "clientId");
57.
58.     Object v = getValue();
59.     if (v != null)
60.        writer.writeAttribute("value", v.toString(), "value");
61.
62.     Integer size = (Integer)getAttributes().get("size");
63.     if(size != null)
64.        writer.writeAttribute("size", size, "size");
65.
66.     writer.endElement("input");
67.  }
68.
69.  private void encodeDecrementButton(ResponseWriter writer, String clientId)
70.        throws IOException {
71.     writer.startElement("input", this);
72.     writer.writeAttribute("type", "submit", null);
73.     writer.writeAttribute("name", clientId + LESS, null);
74.     writer.writeAttribute("value", "<", "value");
75.     writer.endElement("input");
76.  }
```


Listing 9–1 spinner/WEB-INF/classes/com/corejsf/UISpinner.java (cont.)

```
77. private void encodeIncrementButton(ResponseWriter writer, String clientId)
78.                                                    throws IOException {
79.       writer.startElement("input", this);
80.       writer.writeAttribute("type", "submit", null);
81.       writer.writeAttribute("name", clientId + MORE, null);
82.       writer.writeAttribute("value", ">", "value");
83.       writer.endElement("input");
84.    }
85.
86.    private int getIncrementedValue(int submittedValue, int increment) {
87.       Integer minimum = (Integer) getAttributes().get("minimum");
88.       Integer maximum = (Integer) getAttributes().get("maximum");
89.       int newValue = submittedValue + increment;
90.
91.       if ((minimum == null || newValue >= minimum.intValue()) &&
92.          (maximum == null || newValue <= maximum.intValue()))
93.          return newValue;
94.       else
95.          return submittedValue;
96.    }
97. }
```

`javax.faces.component.ValueHolder`

- `void setConverter(Converter converter)`

  Input and output components both have values and therefore both implement the `ValueHolder` interface. Values must be converted, so the `ValueHolder` interface defines a method for setting the converter. Custom components use this method to associate themselves with standard or custom converters.

Implementing Custom Component Tags

Now that you have seen how to implement the spinner component, there is one remaining chore: to supply a tag handler. Component tag handlers are similar to the tag handlers for converters and validators that you saw in Chapter 6. A tag handler needs to gather the attributes that were supplied in the JSF tag and move them into the component object.

Follow these steps to create a tag handler for your custom component:

- Implement a tag class.
- Create (or update) a tag library descriptor (TLD).

JSF provides two tag superclasses, UIComponentTag and UIComponentBodyTag, that you can extend to implement your tag class. You extend the former if your component does not process its *body* (that is, the child tags and text between the start and end tag), and the latter if it does. Only four of the standard JSF tags extend UIComponentBodyTag: f:view, f:verbatim, h:commandLink, and h:outputLink. Our spinner component does not process its body, so it extends UIComponentTag.

NOTE: A tag that implements UIComponentTag can *have* a body, provided that the body tags know how to process themselves. For example, you can add an f:attribute child to a spinner.

Let's look at the implementation of the SpinnerTag class:

```
public class SpinnerTag extends UIComponentTag {
   private String minimum = null;
   private String maximum = null;
   private String size = null;
   private String value = null;
   ...
}
```

The spinner tag class has an instance field for each attribute. The tag class should keep all attributes as String objects, so that the tag user can supply either value binding expressions or values.

Tag classes have five responsibilities:

- To identify a component type
- To identify a renderer type
- To provide setter methods for tag attributes
- To store tag attribute values in the tag's component
- To release resources

The SpinnerTag class identifies its component type as com.corejsf.Spinner and its renderer type as null. A null renderer type means that a component renders itself or nominates its own renderer.

```
public String getComponentType() { return "com.corejsf.Spinner"; }
public String getRendererType()  { return null; }
```

SpinnerTag provides setter methods for the attributes it supports: minimum, maximum, value, and size.

```
public void setMinimum(String newValue) { minimum = newValue; }
public void setMaximum(String newValue) { maximum = newValue; }
public void setSize(String newValue) { size = newValue; }
public void setValue(String newValue) { value = newValue; }
```

Tags must override a setProperties method to copy tag attribute values to the component. The method name is somewhat of a misnomer because it usually sets component attributes or value bindings, not properties.

```
public void setProperties(UIComponent component) {
   // always call the superclass method
   super.setProperties(component);

   setInteger(component, "size", size);
   setInteger(component, "minimum", minimum);
   setInteger(component, "maximum", maximum);
   setString(component, "value", value);
}
```

The spinner tag's setInteger and setString methods are helper methods that set a component attribute or value binding. Here is the setInteger method:

```
public void setInteger(UIComponent component,
      String attributeName, String attributeValue) {
   if (attributeValue == null) return;
   if (isValueReference(attributeValue))
      setValueBinding(component, attributeName, attributeValue);
   else
      component.getAttributes().put(attributeName,
            new Integer(attributeValue));
}
```

If the attribute value is a value reference (such as "#{cardExpirationDate.year}), then we call the setValueBinding helper method. That method goes through the usual laborious contortions to create a ValueBinding object and to pass it to the component's map of value bindings.

```
public void setValueBinding(UIComponent component,
      String attributeName, String attributeValue) {
   FacesContext context = FacesContext.getCurrentInstance();
   Application app = context.getApplication();
   ValueBinding vb = app.createValueBinding(attributeValue);
   component.setValueBinding(attributeName, vb);
}
```

If the attribute value is not a value reference, we convert the attribute value to the target type and put it into the component's attribute map.

NOTE: The map returned by the `UIComponent.getAttributes` method is smart: it can access both attributes and properties. For example, if you call the map's `put` method with an attribute whose name is "`value`", the `setValue` method is called. If the attribute name is "`minimum`", the name/value pair is put into the component's attribute map since the `UISpinner` class doesn't have a `setMinimum` method. Unfortunately, the map isn't smart enough to deal with value bindings.

Finally, tags must implement the `release` method to release resources and reset all instance fields, so that the tag object can be reused for parsing other tags.

```
public void release() {
   // always call the superclass method
   super.release();

   minimum = null;
   maximum = null;
   size = null;
   value = null;
}
```

NOTE: Tag classes must call superclass methods when they override `setProperties` and `release`.

Listing 9–2 contains the complete code for the tag handler.

After you've created your tag class, you need to declare your new tag in a tag library descriptor. Listing 9–3 shows how the `corejsf:spinner` tag is defined. You might notice that we've declared three attributes in the TLD that are not in the `SpinnerTag` class: `binding`, `id`, and `rendered`. We don't need accessor methods for them in the `SpinnerTag` class, because those methods are implemented by `UIComponentTag`.

Listing 9–2 spinner/WEB-INF/classes/com/corejsf/SpinnerTag.java

```
1. package com.corejsf;
2.
3. import javax.faces.application.Application;
4. import javax.faces.component.UIComponent;
5. import javax.faces.context.FacesContext;
6. import javax.faces.el.ValueBinding;
7. import javax.faces.webapp.UIComponentTag;
8.
9. public class SpinnerTag extends UIComponentTag {
10.    private String minimum = null;
11.    private String maximum = null;
```


Listing 9–2 spinner/WEB-INF/classes/com/corejsf/SpinnerTag.java (cont.)

```
12.    private String size = null;
13.    private String value = null;
14.
15.    public String getRendererType() { return null; }
16.    public String getComponentType() { return "com.corejsf.Spinner"; }
17.
18.    public void setMinimum(String newValue) { minimum = newValue; }
19.    public void setMaximum(String newValue) { maximum = newValue; }
20.    public void setSize(String newValue) { size = newValue; }
21.    public void setValue(String newValue) { value = newValue; }
22.
23.    public void setProperties(UIComponent component) {
24.       // always call the superclass method
25.       super.setProperties(component);
26.
27.       setInteger(component, "size", size);
28.       setInteger(component, "minimum", minimum);
29.       setInteger(component, "maximum", maximum);
30.       setString(component, "value", value);
31.    }
32.
33.    public void setInteger(UIComponent component,
34.          String attributeName, String attributeValue) {
35.       if (attributeValue == null) return;
36.       if (isValueReference(attributeValue))
37.          setValueBinding(component, attributeName, attributeValue);
38.       else
39.          component.getAttributes().put(attributeName,
40.                new Integer(attributeValue));
41.    }
42.
43.    public void setString(UIComponent component,
44.          String attributeName, String attributeValue) {
45.       if (attributeValue == null) return;
46.       if (isValueReference(attributeValue))
47.          setValueBinding(component, attributeName, attributeValue);
48.       else
49.          component.getAttributes().put(attributeName, attributeValue);
50.    }
51.
52.    public void setValueBinding(UIComponent component,
53.          String attributeName, String attributeValue) {
54.       FacesContext context = FacesContext.getCurrentInstance();
55.       Application app = context.getApplication();
56.       ValueBinding vb = app.createValueBinding(attributeValue);
```


Listing 9–2 spinner/WEB-INF/classes/com/corejsf/SpinnerTag.java (cont.)

```
57. component.setValueBinding(attributeName, vb);
58.    }
59.
60.    public void release() {
61.       // always call the superclass method
62.       super.release();
63.
64.       minimum = null;
65.       maximum = null;
66.       size = null;
67.       value = null;
68.    }
69. }
```

Listing 9–3 spinner/WEB-INF/spinner.tld

```
1. <?xml version="1.0" encoding="ISO-8859-1" ?>
2. <!DOCTYPE taglib
3.   PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
4.   "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
5. <taglib>
6.    <tlib-version>0.03</tlib-version>
7.    <jsp-version>1.2</jsp-version>
8.    <short-name>spinner</short-name>
9.    <uri>http://corejsf.com/spinner</uri>
10.   <description>This tag library contains a spinner tag</description>
11.
12.   <tag>
13.      <name>spinner</name>
14.      <tag-class>com.corejsf.SpinnerTag</tag-class>
15.
16.      <attribute>
17.         <name>binding</name>
18.         <description>A value binding that points to a bean property</description>
19.      </attribute>
20.
21.      <attribute>
22.         <name>id</name>
23.         <description>The client id of this component</description>
24.      </attribute>
25.
```


Listing 9-3 spinner/WEB-INF/spinner.tld (cont.)

```
26.     <attribute>
27.        <name>rendered</name>
28.        <description>Is this component rendered?</description>
29.     </attribute>
30.
31.     <attribute>
32.        <name>minimum</name>
33.        <description>The spinner minimum value</description>
34.     </attribute>
35.
36.     <attribute>
37.        <name>maximum</name>
38.        <description>The spinner maximum value</description>
39.     </attribute>
40.
41.     <attribute>
42.        <name>size</name>
43.        <description>The size of the input field</description>
44.     </attribute>
45.
46.     <attribute>
47.        <name>value</name>
48.        <required>true</required>
49.        <description>The value of the spinner</description>
50.     </attribute>
51.  </tag>
52. </taglib>
```

`javax.faces.webapp.UIComponentTag`

- `void setProperties(UIComponent component)`

  Transfers tag attribute values to component properties, attributes, or both. Custom components must call the superclass `setProperties` method to make sure that properties are set for the attributes `UIComponentTag` supports: `binding`, `id`, and `rendered`.

`javax.faces.context.FacesContext`

- `static FacesContext getCurrentInstance()`

  Returns a reference to the current `FacesContext` instance.

`javax.faces.application.Application`

- `ValueBinding createValueBinding(String valueReferenceExpression)`
  Creates a value binding and stores it in the application. The string must be a value reference expression of this form: #{...}

`javax.faces.component.UIComponent`

- `void setValueBinding(String name, ValueBinding valueBinding)`
  Stores a value binding by name in the component.

`javax.faces.webapp.UIComponentTag`

- `static boolean isValueReference(String expression)`
  Returns true if `expression` starts with "#{" and ends with "}".

The Spinner Application

After a number of different perspectives of the spinner component, it's time to take a look at the spinner example in its entirety. This section lists the code for the spinner test application shown in Figure 9–1 on page 351. The directory structure is shown in Figure 9–5 and the code is shown in Listing 9–4 through Listing 9–9.

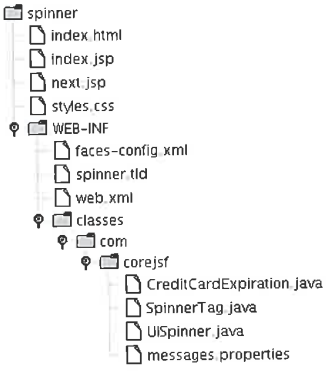


Figure 9–5 Spinner Directory Structure

Listing 9–4 spinner/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <%@ taglib uri="http://corejsf.com/spinner" prefix="corejsf" %>
5.    <f:view>
6.       <head>
7.          <link href="styles.css" rel="stylesheet" type="text/css"/>
8.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
9.          <title><h:outputText value="#{msgs.windowTitle}"/></title>
10.      </head>
11.      <body>
12.         <h:form id="spinnerForm">
13.            <h:outputText value="#{msgs.creditCardExpirationPrompt}"
14.               styleClass="pageTitle"/>
15.            <p/>
16.            <h:panelGrid columns="3">
17.               <h:outputText value="#{msgs.monthPrompt}"/>
18.               <corejsf:spinner value="#{cardExpirationDate.month}"
19.                  id="monthSpinner" minimum="1" maximum="12" size="3"/>
20.               <h:message for="monthSpinner"/>
21.               <h:outputText value="#{msgs.yearPrompt}"/>
22.               <corejsf:spinner value="#{cardExpirationDate.year}"
23.                  id="yearSpinner" minimum="1900" maximum="2100" size="5"/>
24.               <h:message for="yearSpinner"/>
25.            </h:panelGrid>
26.            <p/>
27.            <h:commandButton value="#{msgs.nextButtonPrompt}" action="next"/>
28.         </h:form>
29.      </body>
30.   </f:view>
31. </html>
```

Listing 9–5 spinner/next.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.
5.    <f:view>
6.       <head>
```


Listing 9–5 spinner/next.jsp (cont.)

```
7.          <link href="styles.css" rel="stylesheet" type="text/css"/>
8.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
9.          <title><h:outputText value="#{msgs.windowTitle}"/></title>
10.       </head>
11.       <body>
12.          <h:form>
13.             <h:outputText value="#{msgs.youEnteredPrompt}" styleClass="pageTitle"/>
14.             <p>
15.             <h:outputText value="#{msgs.expirationDatePrompt}"/>
16.                <h:outputText value="#{cardExpirationDate.month}"/> /
17.                <h:outputText value="#{cardExpirationDate.year}"/>
18.             <p>
19.             <h:commandButton value="Try again" action="again"/>
20.          </h:form>
21.       </body>
22.    </f:view>
23. </html>
```

Listing 9–6 spinner/WEB-INF/classes/com/corejsf/CreditCardExpiration.java

```
1. package com.corejsf;
2.
3. public class CreditCardExpiration {
4.    private int month = 1;
5.    private int year = 2000;
6.
7.    // PROPERTY: month
8.    public int getMonth() { return month; }
9.    public void setMonth(int newValue) { month = newValue; }
10.
11.   // PROPERTY: year
12.   public int getYear() { return year; }
13.   public void setYear(int newValue) { year = newValue; }
14. }
```

Listing 9–7 spinner/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2.
3. <!DOCTYPE faces-config PUBLIC
4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
6.
```


Listing 9–7 spinner/WEB-INF/faces-config.xml (cont.)

```
7. <faces-config>
8.
9.    <navigation-rule>
10.      <from-view-id>/index.jsp</from-view-id>
11.      <navigation-case>
12.         <from-outcome>next</from-outcome>
13.         <to-view-id>/next.jsp</to-view-id>
14.      </navigation-case>
15.   </navigation-rule>
16.
17.   <navigation-rule>
18.      <from-view-id>/next.jsp</from-view-id>
19.      <navigation-case>
20.         <from-outcome>again</from-outcome>
21.         <to-view-id>/index.jsp</to-view-id>
22.      </navigation-case>
23.   </navigation-rule>
24.
25.   <component>
26.      <component-type>com.corejsf.Spinner</component-type>
27.      <component-class>com.corejsf.UISpinner</component-class>
28.   </component>
29.
30.   <managed-bean>
31.      <managed-bean-name>cardExpirationDate</managed-bean-name>
32.      <managed-bean-class>com.corejsf.CreditCardExpiration</managed-bean-class>
33.      <managed-bean-scope>session</managed-bean-scope>
34.   </managed-bean>
35.
36. </faces-config>
```

Listing 9–8 spinner/WEB-INF/classes/com/corejsf/messages.properties

```
1. windowTitle=Spinner Test
2. creditCardExpirationPrompt=Please enter your credit card expiration date:
3. monthPrompt=Month:
4. yearPrompt=Year:
5. nextButtonPrompt=Next
6. youEnteredPrompt=You entered:
7. expirationDatePrompt=Expiration Date
8. changes=Changes:
```


Listing 9–9 spinner/styles.css

```
1. body {
2.     background: #eee;
3. }
4. .pageTitle {
5.     font-size: 1.25em;
6. }
```

Revisiting the Spinner

Let's revisit the spinner listed in the previous section. That spinner has two serious drawbacks. First, the spinner component renders itself, so you couldn't, for example, attach a separate renderer to the spinner when you migrate your application to cell phones. Second, the spinner requires a roundtrip to the server every time a user clicks on the increment or decrement button. Nobody would implement an industrial-strength spinner with those deficiencies. Let's see how to address them.

While we are at it, we will also add another feature to the spinner—the ability to attach value change listeners.

Using an External Renderer

In the preceding example, the `UISpinner` class was in charge of its own rendering. However, most UI classes delegate rendering to a separate class. Using separate renderers is a good idea: it becomes easy to replace renderers, to adapt to a different UI toolkit, or simply to achieve different HTML effects. In "Encoding JavaScript to Avoid Server Roundtrips" on page 396 we see how to use an alternative renderer that uses JavaScript to keep track of the spinner's value on the client.

Using an external renderer requires these steps:

- Define an ID string for your renderer.
- Declare the renderer in a JSF configuration file.
- Modify your tag class to return the renderer's ID from `getRendererType()`.
- Implement the renderer class.

The identifier—in our case, `com.corejsf.Spinner`—must be defined in a JSF configuration file, like this:


```
<faces-config>
   ...
   <component>
      <component-type>com.corejsf.Spinner</component-type>
      <component-class>com.corejsf.UISpinner</component-class>
   </component>

   <render-kit>
      <renderer>
         <component-family>javax.faces.Input</component-family>
         <renderer-type>com.corejsf.Spinner</renderer-type>
         <renderer-class>com.corejsf.SpinnerRenderer</renderer-class>
      </renderer>
   </render-kit>
</faces-config>
```

The `component-family` element serves to overcome a historical problem. The names of the standard HTML tags are meant to indicate the component type and the renderer type. For example, an `h:selectOneMenu` is a `UISelectOne` component whose renderer has type `javax.faces.Menu`. That same renderer can also be used for the `h:selectManyMenu` tag. But the scheme didn't work so well. The renderer for `h:inputText` writes an HTML `input` text field. That renderer won't work for `h:outputText`—you don't want to use a text field for output. So, instead of identifying renderers by individual components, renderers are determined by the renderer type and the *component family*. Table 9–1 shows the component families of all standard component classes. In our case, we use the component family `javax.faces.Input` because `UISpinner` is a subclass of `UIInput`.

Table 9–1 Component Families of Standard Component Classes

Component Class	Component Family
UICommand	javax.faces.Command
UIData	javax.faces.Data
UIForm	javax.faces.Form
UIGraphic	javax.faces.Graphic
UIInput	javax.faces.Input
UIMessage	javax.faces.Message

Table 9–1 Component Families of Standard Component Classes (cont.)

Component Class	Component Family
UIMessages	javax.faces.Messages
UIOutput	javax.faces.Output
UIPanel	javax.faces.Panel
UISelectBoolean	javax.faces.SelectBoolean
UISelectMany	javax.faces.SelectMany
UISelectOne	javax.faces.SelectOne

The getRendererType of your tag class needs to return the renderer ID.

```
public class SpinnerTag extends UIComponentTag {
   ...
   public String getComponentType() { return "com.corejsf.Spinner"; }
   public String getRendererType()  { return "com.corejsf.Spinner"; }
   ...
}
```

NOTE: Component IDs and Renderer IDs have separate name spaces. It is okay to use the same string as a component ID and a renderer ID.

It is also a good idea to set the renderer type in the component constructor:

```
public class UISpinner extends UIInput {
   public UISpinner() {
      setConverter(new IntegerConverter()); // to convert the submitted value
      setRendererType("com.corejsf.Spinner"); // this component has a renderer
   }
}
```

Then the renderer type is properly set if a component is programmatically constructed of components, without the use of tags.

The final step is implementing the renderer itself. Renderers extend the javax.faces.render.Renderer class. That class has seven methods, four of which are familiar:

- void encodeBegin(FacesContext context, UIComponent component)
- void encodeChildren(FacesContext context, UIComponent component)
- void encodeEnd(FacesContext context, UIComponent component)
- void decode(FacesContext context, UIComponent component)

The renderer methods listed above are almost identical to their component counterparts except that the renderer methods take an additional argument: a reference to the component being rendered. To implement those methods for the spinner renderer, we move the component methods to the renderer and apply code changes to compensate for the fact that the renderer is passed a reference to the component. That's easy to do.

Here are the remaining `Renderer` methods:

- `Object getConvertedValue(FacesContext context, UIComponent component, Object submittedValue)`
- `boolean getRendersChildren()`
- `String convertClientId(FacesContext context, String clientId)`

The `getConvertedValue` method converts a component's submitted value from a string to an object. The default implementation in the `Renderer` class simply returns the value.

The `getRendersChildren` method specifies whether a renderer is responsible for rendering its component's children. If that method returns `true`, JSF will call the renderer's `encodeChildren` method; if it returns `false` (the default behavior), the JSF implementation won't call that method.

The `convertClientId` method converts an ID string (such as `_id1:monthSpinner`) so that it can be used on the client—some clients may place restrictions on IDs, such as disallowing special characters. However, the default implementation simply returns the ID string, unchanged.

If you have a component that renders itself, it's usually a simple task to move code from the component to the renderer. Listing 9–10 and Listing 9–11 show the code for the spinner component and renderer, respectively.

Listing 9–10 spinner2/WEB-INF/classes/com/corejsf/UISpinner.java

```
1. package com.corejsf;
2.
3. import javax.faces.component.UIInput;
4. import javax.faces.convert.IntegerConverter;
5.
6. public class UISpinner extends UIInput {
7.    public UISpinner() {
8.       setConverter(new IntegerConverter()); // to convert the submitted value
9.       setRendererType("com.corejsf.Spinner");  // this component has a renderer
10.   }
11. }
```


Listing 9–11 spinner2/WEB-INF/classes/com/corejsf/SpinnerRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
5. import javax.faces.component.UIComponent;
6. import javax.faces.component.EditableValueHolder;
7. import javax.faces.component.UIInput;
8. import javax.faces.context.FacesContext;
9. import javax.faces.context.ResponseWriter;
10. import javax.faces.convert.ConverterException;
11. import javax.faces.render.Renderer;
12.
13. public class SpinnerRenderer extends Renderer {
14.    private static final String MORE = ".more";
15.    private static final String LESS = ".less";
16.
17.    public Object getConvertedValue(FacesContext context, UIComponent component,
18.          Object submittedValue) throws ConverterException {
19.       return com.corejsf.util.Renderers.getConvertedValue(context, component,
20.          submittedValue);
21.    }
22.
23.    public void encodeBegin(FacesContext context, UIComponent spinner)
24.          throws IOException {
25.       ResponseWriter writer = context.getResponseWriter();
26.       String clientId = spinner.getClientId(context);
27.
28.       encodeInputField(spinner, writer, clientId);
29.       encodeDecrementButton(spinner, writer, clientId);
30.       encodeIncrementButton(spinner, writer, clientId);
31.    }
32.
33.    public void decode(FacesContext context, UIComponent component) {
34.       EditableValueHolder spinner = (EditableValueHolder) component;
35.       Map requestMap = context.getExternalContext().getRequestParameterMap();
36.       String clientId = component.getClientId(context);
37.
38.       int increment;
39.       if (requestMap.containsKey(clientId + MORE)) increment = 1;
40.       else if (requestMap.containsKey(clientId + LESS)) increment = -1;
41.       else increment = 0;
42.
43.       try {
44.          int submittedValue
45.             = Integer.parseInt((String) requestMap.get(clientId));
```

Listing 9–11 spinner2/WEB-INF/classes/com/corejsf/SpinnerRenderer.java (cont.)

```
46.
47.         int newValue = getIncrementedValue(component, submittedValue,
48.            increment);
49.         spinner.setSubmittedValue("" + newValue);
50.         spinner.setValid(true);
51.      }
52.      catch(NumberFormatException ex) {
53.         // let the converter take care of bad input, but we still have
54.         // to set the submitted value, or the converter won't have
55.         // any input to deal with
56.         spinner.setSubmittedValue((String) requestMap.get(clientId));
57.      }
58.   }
59.
60.   private void encodeInputField(UIComponent spinner, ResponseWriter writer,
61.         String clientId) throws IOException {
62.      writer.startElement("input", spinner);
63.      writer.writeAttribute("name", clientId, "clientId");
64.
65.      Object v = ((UIInput)spinner).getValue();
66.      if(v != null)
67.         writer.writeAttribute("value", v.toString(), "value");
68.
69.      Integer size = (Integer)spinner.getAttributes().get("size");
70.      if(size != null)
71.         writer.writeAttribute("size", size, "size");
72.
73.      writer.endElement("input");
74.   }
75.
76.   private void encodeDecrementButton(UIComponent spinner,
77.         ResponseWriter writer, String clientId) throws IOException {
78.      writer.startElement("input", spinner);
79.      writer.writeAttribute("type", "submit", null);
80.      writer.writeAttribute("name", clientId + LESS, null);
81.      writer.writeAttribute("value", "<", "value");
82.      writer.endElement("input");
83.   }
84.
85.   private void encodeIncrementButton(UIComponent spinner,
86.         ResponseWriter writer, String clientId) throws IOException {
87.      writer.startElement("input", spinner);
88.      writer.writeAttribute("type", "submit", null);
89.      writer.writeAttribute("name", clientId + MORE, null);
90.      writer.writeAttribute("value", ">", "value");
```

Listing 9–11 spinner2/WEB-INF/classes/com/corejsf/SpinnerRenderer.java (cont.)

```
91.         writer.endElement("input");
92.     }
93.
94.     private int getIncrementedValue(UIComponent spinner, int submittedValue,
95.             int increment) {
96.         Integer minimum = (Integer) spinner.getAttributes().get("minimum");
97.         Integer maximum = (Integer) spinner.getAttributes().get("maximum");
98.         int newValue = submittedValue + increment;
99.
100.        if ((minimum == null || newValue >= minimum.intValue()) &&
101.            (maximum == null || newValue <= maximum.intValue()))
102.            return newValue;
103.        else
104.            return submittedValue;
105.    }
106. }
```

Calling Converters from External Renderers

If you compare Listing 9–10 and Listing 9–11 with Listing 9–1, you'll see that we moved most of the code from the original component class to a new renderer class.

However, there is a hitch. As you can see from Listing 9–10, the spinner handles conversions simply by invoking `setConverter()` in its constructor. Because the spinner is an input component, its superclass—`UIInput`—uses the specified converter during the "Process Validations" phase of the life cycle.

But when the spinner delegates to a renderer, it's the renderer's responsibility to convert the spinner's value by overriding `Renderer.getConvertedValue()`. So we must replicate the conversion code from `UIInput` in a custom renderer. We placed that code—which is required in all renderers that use a converter—in the static `getConvertedValue` method of the class `com.corejsf.util.Renderers` (see Listing 9–12 on page 388).

NOTE: The `Renderers.getConvertedValue` method shown in Listing 9–12 is a necessary evil because `UIInput` does not make its conversion code publicly available. That code resides in `UIInput.validate`, which looks like this in the JSF 1.0 Reference Implementation:

```
// This code is from the javax.faces.component.UIInput class:

public void validate(FacesContext context) {
```



```
    if (renderer != null) {
        newValue = renderer.getConvertedValue(context, this,
                                                  submittedValue);
    } else if (submittedValue instanceof String) {
        // If there's no Renderer and we've got a String,
        // run it through the Converter (if any)
        Converter converter = getConverterWithType(context);
        ...
        // much more code follows for converting the UIInput's value
        // and for dealing with conversion failures...
    }
}
```

Because UIInput's conversion code is buried in the validate method, it's not available for a renderer to reuse, as would be the case, for example, if UIInput implemented that code in a public getConvertedValue method. Because UIInput's conversion code can't be reused, you must reimplement it for custom components that use standard converters to convert their values. Fortunately, we've already done it for you.

Supporting Value Change Listeners

If your custom component is an input component, you can fire value change events to interested listeners. For example, in a calendar application, you may want to update another component whenever a month spinner value changes.

Fortunately, it is easy to support value change listeners. The UIInput class automatically generates value change events whenever the input value has changed. Recall that there are two ways of attaching a value change listener. You can add one or more listeners with f:valueChangeListener, like this:

```
<corejsf:spinner ...>
    <f:valueChangeListener type="com.corejsf.SpinnerListener"/>
    ...
</corejsf:spinner>
```

Or you can use a valueChangeListener attribute:

```
<corejsf:spinner value="#{cardExpirationDate.month}"
   id="monthSpinner" minimum="1" maximum="12" size="3"
   valueChangeListener="#{cardExpirationDate.changeListener}"/>
```

The first way doesn't require any effort on the part of the component implementor. The second way merely requires that your tag handler supports the valueChangeListener attribute. The attribute value is a method binding that requires special handling—the topic of the next section.

Supporting Method Bindings

Four commonly used attributes require method bindings—see Table 9–2. You create a MethodBinding object by calling the createMethodBinding method of the Application class. That method has two parameters: the method binding expression and an array of Class objects that describe the method's parameter types. For example, this code creates a method binding for a value change listener:

```
FacesContext context = FacesContext.getCurrentInstance();
Application app = context.getApplication();
Class[] paramTypes = new Class[] { ValueChangeListener.class };
MethodBinding mb = app.createMethodBinding(attributeValue, paramTypes);
```

You then store the MethodBinding object with the component in the usual way:

```
component.getAttributes().put("valueChangeListener", mb);
```

Alternatively, you can call the property setter directly:

```
((EditableValueHolder) component).setValueChangeListener(mb);
```

Table 9–2 Method Binding Attributes

Attribute Name	Method Parameters
valueChangeListener	ValueChangeEvent
validator	FacesContext, UIComponent, Object
actionListener	ActionEvent
action	*none*

Nobody likes to write this tedious code, so we bundled it with the setValueChangeListener method of the convenience class com.corejsf.util.Tags (see Listing 9–13 on page 390). The SpinnerTag class simply calls

```
com.corejsf.util.Tags.setValueChangeListener(component,
   valueChangeListener);
```

Action listeners and validators follow exactly the same pattern—see the setActionListener and setValidator methods in the com.corejsf.util.Tags class.

However, actions are slightly more complex. An action can either be a method binding or a fixed string, for example


```
<h:commandButton value="Login" action="#{loginController.verifyUser}"/>
```

or

```
<h:commandButton value="Login" action="login"/>
```

But the setAction method of the ActionSource interface requires a MethodBinding in all cases. Therefore, we must construct a MethodBinding object whose getExpressionString method returns the given string—see the setAction method of the Tags class.

In the next sample program, we demonstrate the value change listener by keeping a count of all value changes that we display on the form (see Figure 9–6).

```
public class CreditCardExpiration {
   private int changes = 0;
   // to demonstrate the value change listener
   public void changeListener(ValueChangeEvent e) {
        changes++;
   }
}
```

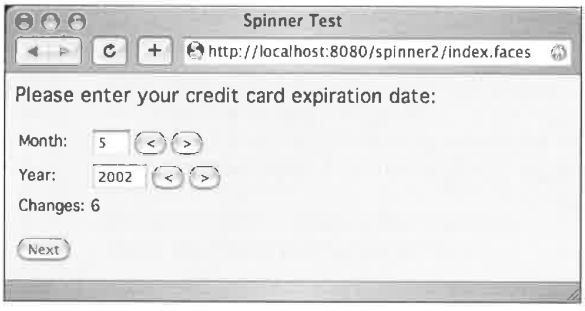


Figure 9–6 Counting the Value Changes

Figure 9–7 shows the directory structure of the sample application. As you can see, we rely on the Core JSF Renderers and Tags convenience classes that contain much of the repetitive code. (The Renderers class also contains a getSelectedItems method that we need later in this chapter—ignore it for now.) Listing 9–14 contains the revised SpinnerTag class, and Listing 9–15 shows the faces-config.xml file.

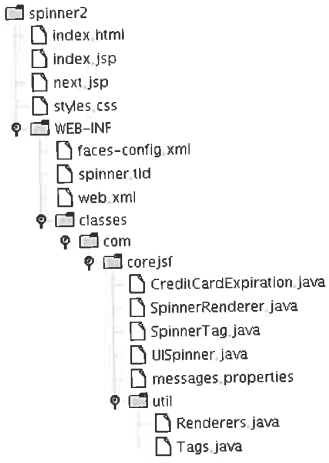


Figure 9–7 Directory Structure of the Revisited Spinner Example

Listing 9–12 spinner2/WEB-INF/classes/com/corejsf/util/Renderers.java

```
package com.corejsf.util;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import javax.faces.application.Application;
import javax.faces.component.UIComponent;
import javax.faces.component.UIForm;
import javax.faces.component.UISelectItem;
import javax.faces.component.UISelectItems;
import javax.faces.component.ValueHolder;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.el.ValueBinding;
import javax.faces.model.SelectItem;

public class Renderers {
   public static Object getConvertedValue(FacesContext context,
      UIComponent component,
```

Listing 9–12 spinner2/WEB-INF/classes/com/corejsf/util/Renderers.java (cont.)

```
25.         Object submittedValue) throws ConverterException {
26.      if (submittedValue instanceof String) {
27.         Converter converter = getConverter(context, component);
28.         if (converter != null) {
29.            return converter.getAsObject(context, component,
30.                  (String) submittedValue);
31.         }
32.      }
33.      return submittedValue;
34.   }
35.
36.   public static Converter getConverter(FacesContext context,
37.      UIComponent component) {
38.      if (!(component instanceof ValueHolder)) return null;
39.      ValueHolder holder = (ValueHolder) component;
40.
41.      Converter converter = holder.getConverter();
42.      if (converter != null) return converter;
43.
44.      ValueBinding valueBinding = component.getValueBinding("value");
45.      if (valueBinding == null) return null;
46.
47.      Class targetType = valueBinding.getType(context);
48.      if (targetType == null) return null;
49.      // Version 1.0 of the reference implementation will not apply a converter
50.      // if the target type is String or Object, but that is a bug.
51.
52.      Application app = context.getApplication();
53.      return app.createConverter(targetType);
54.   }
55.
56.   public static String getFormId(FacesContext context, UIComponent component) {
57.      UIComponent parent = component;
58.      while (!(parent instanceof UIForm)) parent = parent.getParent();
59.      return parent.getClientId(context);
60.   }
61.
62.   public static List getSelectItems(UIComponent component) {
63.      ArrayList list = new ArrayList();
64.      Iterator children = component.getChildren().iterator();
65.      while (children.hasNext()) {
66.         UIComponent child = (UIComponent) children.next();
67.
68.         if (child instanceof UISelectItem) {
69.            Object value = ((UISelectItem) child).getValue();
```

Listing 9–12 spinner2/WEB-INF/classes/com/corejsf/util/Renderers.java (cont.)

```
70.             if (value == null) {
71.                UISelectItem item = (UISelectItem) child;
72.                list.add(new SelectItem(item.getItemValue(),
73.                      item.getItemLabel(),
74.                      item.getItemDescription(),
75.                      item.isItemDisabled()));
76.             } else if (value instanceof SelectItem) {
77.                list.add(value);
78.             }
79.          } else if (child instanceof UISelectItems) {
80.             Object value = ((UISelectItems) child).getValue();
81.             if (value instanceof SelectItem)
82.                list.add(value);
83.             else if (value instanceof SelectItem[])
84.                list.addAll(Arrays.asList((SelectItem[]) value));
85.             else if (value instanceof Collection)
86.                list.addAll((Collection) value);
87.             else if (value instanceof Map) {
88.                Iterator entries = ((Map) value).entrySet().iterator();
89.                while (entries.hasNext()) {
90.                   Map.Entry entry = (Map.Entry) entries.next();
91.                   list.add(new SelectItem(entry.getKey(),
92.                         "" + entry.getValue()));
93.                }
94.             }
95.          }
96.       }
97.       return list;
98.    }
99. }
```

Listing 9–13 spinner2/WEB-INF/classes/com/corejsf/util/Tags.java

```
1. package com.corejsf.util;
2.
3. import java.io.Serializable;
4. import javax.faces.application.Application;
5. import javax.faces.component.UIComponent;
6. import javax.faces.context.FacesContext;
7. import javax.faces.el.MethodBinding;
8. import javax.faces.el.ValueBinding;
9. import javax.faces.event.ActionEvent;
10. import javax.faces.event.ValueChangeEvent;
11. import javax.faces.webapp.UIComponentTag;
```


Listing 9–13 spinner2/WEB-INF/classes/com/corejsf/util/Tags.java (cont.)

```
12.
13. public class Tags {
14.    public static void setString(UIComponent component, String attributeName,
15.          String attributeValue) {
16.       if (attributeValue == null)
17.          return;
18.       if (UIComponentTag.isValueReference(attributeValue))
19.          setValueBinding(component, attributeName, attributeValue);
20.       else
21.          component.getAttributes().put(attributeName, attributeValue);
22.    }
23.
24.    public static void setInteger(UIComponent component,
25.          String attributeName, String attributeValue) {
26.       if (attributeValue == null) return;
27.       if (UIComponentTag.isValueReference(attributeValue))
28.          setValueBinding(component, attributeName, attributeValue);
29.       else
30.          component.getAttributes().put(attributeName,
31.                new Integer(attributeValue));
32.    }
33.
34.    public static void setBoolean(UIComponent component,
35.          String attributeName, String attributeValue) {
36.       if (attributeValue == null) return;
37.       if (UIComponentTag.isValueReference(attributeValue))
38.          setValueBinding(component, attributeName, attributeValue);
39.       else
40.          component.getAttributes().put(attributeName,
41.                new Boolean(attributeValue));
42.    }
43.
44.    public static void setValueBinding(UIComponent component, String attributeName,
45.          String attributeValue) {
46.       FacesContext context = FacesContext.getCurrentInstance();
47.       Application app = context.getApplication();
48.       ValueBinding vb = app.createValueBinding(attributeValue);
49.       component.setValueBinding(attributeName, vb);
50.    }
51.
52.    public static void setActionListener(UIComponent component,
53.       String attributeValue) {
54.       setMethodBinding(component, "actionListener", attributeValue,
55.             new Class[] { ActionEvent.class });
56.    }
```


Listing 9–13 spinner2/WEB-INF/classes/com/corejsf/util/Tags.java (cont.)

```
57.
58.  public static void setValueChangeListener(UIComponent component,
59.        String attributeValue) {
60.     setMethodBinding(component, "valueChangeListener", attributeValue,
61.           new Class[] { ValueChangeEvent.class });
62.  }
63.
64.  public static void setValidator(UIComponent component,
65.        String attributeValue) {
66.     setMethodBinding(component, "validator", attributeValue,
67.         new Class[] { FacesContext.class, UIComponent.class, Object.class });
68.  }
69.
70.  public static void setAction(UIComponent component, String attributeValue) {
71.     if (attributeValue == null) return;
72.     if (UIComponentTag.isValueReference(attributeValue))
73.        setMethodBinding(component, "action", attributeValue,
74.              new Class[] {});
75.     else {
76.        FacesContext context = FacesContext.getCurrentInstance();
77.        Application app = context.getApplication();
78.        MethodBinding mb = new ActionMethodBinding(attributeValue);
79.        component.getAttributes().put("action", mb);
80.     }
81.  }
82.
83.  public static void setMethodBinding(UIComponent component, String attributeName,
84.        String attributeValue, Class[] paramTypes) {
85.     if (attributeValue == null)
86.        return;
87.     if (UIComponentTag.isValueReference(attributeValue)) {
88.        FacesContext context = FacesContext.getCurrentInstance();
89.        Application app = context.getApplication();
90.        MethodBinding mb = app.createMethodBinding(attributeValue, paramTypes);
91.        component.getAttributes().put(attributeName, mb);
92.     }
93.  }
94.
95.  private static class ActionMethodBinding
96.        extends MethodBinding implements Serializable {
97.     private String result;
98.
99.     public ActionMethodBinding(String result) { this.result = result; }
100.    public Object invoke(FacesContext context, Object params[]) {
101.       return result;
```


Listing 9–13 spinner2/WEB-INF/classes/com/corejsf/util/Tags.java (cont.)

```
102.        }
103.        public String getExpressionString() { return result; }
104.        public Class getType(FacesContext context) { return String.class; }
105.     }
106. }
```

Listing 9–14 spinner2/WEB-INF/classes/com/corejsf/SpinnerTag.java

```
1. package com.corejsf;
2.
3. import javax.faces.component.UIComponent;
4. import javax.faces.webapp.UIComponentTag;
5.
6. public class SpinnerTag extends UIComponentTag {
7.    private String minimum = null;
8.    private String maximum = null;
9.    private String size = null;
10.   private String value = null;
11.   private String valueChangeListener = null;
12.
13.   public String getRendererType() { return "com.corejsf.Spinner"; }
14.   public String getComponentType() { return "com.corejsf.Spinner"; }
15.
16.   public void setMinimum(String newValue) { minimum = newValue; }
17.   public void setMaximum(String newValue) { maximum = newValue; }
18.   public void setSize(String newValue) { size = newValue; }
19.   public void setValue(String newValue) { value = newValue; }
20.   public void setValueChangeListener(String newValue)  {
21.      valueChangeListener = newValue;
22.   }
23.
24.   public void setProperties(UIComponent component) {
25.      // always call the superclass method
26.      super.setProperties(component);
27.
28.      com.corejsf.util.Tags.setInteger(component, "size", size);
29.      com.corejsf.util.Tags.setInteger(component, "minimum", minimum);
30.      com.corejsf.util.Tags.setInteger(component, "maximum", maximum);
31.      com.corejsf.util.Tags.setString(component, "value", value);
32.      com.corejsf.util.Tags.setValueChangeListener(component,
33.            valueChangeListener);
34.   }
35.
36.   public void release() {
```


Listing 9–14 spinner2/WEB-INF/classes/com/corejsf/SpinnerTag.java (cont.)

```
37.         // always call the superclass method
38.         super.release();
39.
40.         minimum = null;
41.         maximum = null;
42.         size = null;
43.         value = null;
44.         valueChangeListener = null;
45.     }
46. }
```

Listing 9–15 spinner2/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2.
3. <!DOCTYPE faces-config PUBLIC
4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
6.
7. <faces-config>
8.
9.    <navigation-rule>
10.       <from-view-id>/index.jsp</from-view-id>
11.       <navigation-case>
12.          <from-outcome>next</from-outcome>
13.          <to-view-id>/next.jsp</to-view-id>
14.       </navigation-case>
15.    </navigation-rule>
16.
17.    <navigation-rule>
18.       <from-view-id>/next.jsp</from-view-id>
19.       <navigation-case>
20.          <from-outcome>again</from-outcome>
21.          <to-view-id>/index.jsp</to-view-id>
22.       </navigation-case>
23.    </navigation-rule>
24.
25.    <managed-bean>
26.       <managed-bean-name>cardExpirationDate</managed-bean-name>
27.       <managed-bean-class>com.corejsf.CreditCardExpiration</managed-bean-class>
28.       <managed-bean-scope>session</managed-bean-scope>
29.    </managed-bean>
30.
```


Listing 9–15 spinner2/WEB-INF/faces-config.xml (cont.)

```
31.    <component>
32.       <component-type>com.corejsf.Spinner</component-type>
33.       <component-class>com.corejsf.UISpinner</component-class>
34.    </component>
35.
36.    <render-kit>
37.       <renderer>
38.          <component-family>javax.faces.Input</component-family>
39.          <renderer-type>com.corejsf.Spinner</renderer-type>
40.          <renderer-class>com.corejsf.SpinnerRenderer</renderer-class>
41.       <renderer>
42.    <render-kit>
43. </faces-config>
```

`javax.faces.context.FacesContext`

- `Application getApplication()`
  Returns a reference to the application object.

`javax.faces.application.Application`

- `ValueBinding createMethodBinding(String valueReferenceExpression, Class[] arguments)`
  Creates a method binding and stores it in the application. The `valueReferenceExpression` must be a value reference expression. The `Class[]` represents the types of the arguments passed to the method.

`javax.faces.component.EditableValueHolder`

- `void setValueChangeListener(MethodBinding listenerMethod)`
  Sets a method binding for a component that implements the `EditableValueHolder` interface. That method must return void and is passed a `ValueChangeEvent`.

`javax.faces.event.ValueChangeEvent`

- `Object getOldValue()`
  Returns the component's old value.
- `Object getNewValue()`
  Returns the component's new value.

`javax.faces.component.ValueHolder`

- `Converter getConverter()`
  Returns the converter associated with a component. The `ValueHolder` interface is implemented by input and output components.

`javax.faces.component.UIComponent`

- `ValueBinding getValueBinding(String valueBindingName)`
  Returns a value binding previously set by calling `UIComponent.setValueBinding()`. That method is discussed on page 374.

`javax.faces.el.ValueBinding`

- `Class getType(FacesContext context) throws EvaluationException, PropertyNotFoundException`
  Returns the class of the object to which a value binding applies. That class can subsequently be used to access a converter with `Application.createConverter()`.

`javax.faces.application.Application`

- `Converter createConverter(Class targetClass) throws FacesException, NullPointerException`
  Creates a converter, given its target class. JSF implementations maintain a map of valid converter types, which are typically specified in a faces configuration file. If `targetClass` is a key in that map, this method creates an instance of the associated converter (specified as the value for the `targetClass` key) and returns it. If `targetClass` is not in the map, this method searches the map for a key that corresponds to `targetClass`'s interfaces and superclasses, in that order, until it finds a matching class. Once a matching class is found, this method creates an associated converter and returns it. If no converter is found for the `targetClass`, it's interfaces, or it's superclasses, this method returns `null`.

Encoding JavaScript to Avoid Server Roundtrips

The spinner component performs a roundtrip to the server every time you click one of its buttons. That roundtrip updates the spinner's value on the server. Those roundtrips can take a severe bite out of the spinner's performance, so in almost all circumstances, it's better to store the spinner's value

on the client and update the component's value only when the form in which the spinner resides is submitted. We can do that with JavaScript that looks like this:

```
<input type="text" name="_id1:monthSpinner" value="0"/>

<script language="JavaScript">
   document['_id1']['_id1:monthSpinner'].spin = function (increment) {
      var v = parseInt(this.value) + increment;
      if (isNaN(v)) return;
      if ('min' in this && v < this.min) return;
      if ('max' in this && v > this.max) return;
         this.value = v;
   };
   document['_id1']['_id1:monthSpinner'].min = 0;
</script>

<input type="button" value="<"
      onclick="document['_id1']['_id1:monthSpinner'].spin(-1);"/>
<input type="button" value=">"
      onclick="document['_id1']['_id1:monthSpinner'].spin(1);"/>
```

When you write JavaScript code that accesses fields in a form, you need to have access to the form ID, such as '_id1' in the expression

```
document['_id1']['_id1:monthSpinner']
```

The second array index is simply the client ID of the component.

Obtaining the form ID is a common task, and we added a convenience method to the `com.corejsf.util.Renderers` class for this purpose:

```
public static String getFormId(FacesContext context, UIComponent component) {
   UIComponent parent = component;
   while (!(parent instanceof UIForm)) parent = parent.getParent();
   return parent.getClientId(context);
}
```

We won't go into the details of JavaScript programming here, but note that we are a bit paranoid about injecting global JavaScript functions into an unknown page. We don't want to risk name conflicts. Fortunately, JavaScript is a well-designed language with a flexible object model. Rather than writing a global `spin` function, we define `spin` to be a method of the text field object. JavaScript lets you enhance the capabilities of objects on-the-fly, simply by adding methods and fields. We use the same approach with the minimum and maximum values of the spinner, adding `min` and `max` fields if they are required.

The spinner renderer that encodes the preceding JavaScript is shown in Listing 9–16.

Note that the UISpinner component is completely unaffected by this change. Only the renderer has been updated, thus demonstrating the power of pluggable renderers.

Listing 9–16 spinner-js/WEB-INF/classes/com/corejsf/JSSpinnerRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.text.MessageFormat;
5. import java.util.Map;
6. import javax.faces.component.EditableValueHolder;
7. import javax.faces.component.UIComponent;
8. import javax.faces.component.UIInput;
9. import javax.faces.context.FacesContext;
10. import javax.faces.context.ResponseWriter;
11. import javax.faces.convert.ConverterException;
12. import javax.faces.render.Renderer;
13.
14. public class JSSpinnerRenderer extends Renderer {
15.    private static final String MORE = ".more";
16.    private static final String LESS = ".less";
17.
18.    public Object getConvertedValue(FacesContext context, UIComponent component,
19.          Object submittedValue) throws ConverterException {
20.       return com.corejsf.util.Renderers.getConvertedValue(context, component,
21.          submittedValue);
22.    }
23.
24.    public void encodeBegin(FacesContext context, UIComponent component)
25.          throws IOException {
26.       ResponseWriter writer = context.getResponseWriter();
27.       String clientId = component.getClientId(context);
28.       String formId = com.corejsf.util.Renderers.getFormId(context, component);
29.
30.       UIInput spinner = (UIInput)component;
31.       Integer min = (Integer) component.getAttributes().get("minimum");
32.       Integer max = (Integer) component.getAttributes().get("maximum");
33.       Integer size = (Integer) component.getAttributes().get("size");
34.
35.       writer.write(MessageFormat.format(
36.          "<input type=\"text\" name=\"{0}\" value=\"{1}\"",
37.          new Object[] { clientId, spinner.getValue().toString() } ));
38.
39.       if (size != null)
```


Listing 9-16 spinner-js/WEB-INF/classes/com/corejsf/JSSpinnerRenderer.java (cont.)

```
40.         writer.write(MessageFormat.format(
41.            " size=\"{0}\"", new Object[] { size } ));
42.      writer.write(MessageFormat.format("/>"
43.         + "<script language=\"JavaScript\">"
44.         + "document.forms[''{0}''][''{1}''].spin = function (increment) '{'"
45.         + "var v = parseInt(this.value) + increment;"
46.         + "if (isNaN(v)) return;"
47.         + "if (\"min\" in this && v < this.min) return;"
48.         + "if (\"max\" in this && v > this.max) return;"
49.         + "this.value = v;"
50.         + "};",
51.         new Object[] { formId, clientId } ));
52.
53.      if (min != null) {
54.         writer.write(MessageFormat.format(
55.            "document.forms[''{0}''][''{1}''].min = {2};",
56.            new Object[] { formId, clientId, min }));
57.      }
58.      if (max != null) {
59.         writer.write(MessageFormat.format(
60.            "document.forms[''{0}''][''{1}''].max = {2};",
61.            new Object[] { formId, clientId, max }));
62.      }
63.      writer.write(MessageFormat.format(
64.         "</script>"
65.         + "<input type=\"button\" value=\"<\""
66.         + "onclick=\"document.forms[''{0}''][''{1}''].spin(-1); }\"/>"
67.         + "<input type=\"button\" value=\">\""
68.         + "onclick=\"document.forms[''{0}''][''{1}''].spin(1); }\"/>",
69.         new Object[] { formId, clientId }));
70.   }
71.
72.   public void decode(FacesContext context, UIComponent component) {
73.      EditableValueHolder spinner = (EditableValueHolder) component;
74.      Map requestMap = context.getExternalContext().getRequestParameterMap();
75.      String clientId = component.getClientId(context);
76.
77.      int increment;
78.      if (requestMap.containsKey(clientId + MORE)) increment = 1;
79.      else if (requestMap.containsKey(clientId + LESS)) increment = -1;
80.      else increment = 0;
81.
82.      try {
83.         int submittedValue
```

Listing 9–16 spinner-js/WEB-INF/classes/com/corejsf/JSSpinnerRenderer.java (cont.)

```
            = Integer.parseInt((String) requestMap.get(clientId));

         int newValue = getIncrementedValue(component, submittedValue,
            increment);
         spinner.setSubmittedValue("" + newValue);
         spinner.setValid(true);
      }
      catch(NumberFormatException ex) {
         // let the converter take care of bad input, but we still have
         // to set the submitted value, or the converter won't have
         // any input to deal with
         spinner.setSubmittedValue((String) requestMap.get(clientId));
      }
   }

   private void encodeDecrementButton(UIComponent spinner,
         ResponseWriter writer, String clientId) throws IOException {
      writer.startElement("input", spinner);
      writer.writeAttribute("type", "submit", null);
      writer.writeAttribute("name", clientId + LESS, null);
      writer.writeAttribute("value", "<", "value");
      writer.endElement("input");
   }

   private void encodeIncrementButton(UIComponent spinner,
         ResponseWriter writer, String clientId) throws IOException {
      writer.startElement("input", spinner);
      writer.writeAttribute("type", "submit", null);
      writer.writeAttribute("name", clientId + MORE, null);
      writer.writeAttribute("value", ">", "value");
      writer.endElement("input");
   }

   private int getIncrementedValue(UIComponent spinner, int submittedValue,
         int increment) {
      Integer minimum = (Integer) spinner.getAttributes().get("minimum");
      Integer maximum = (Integer) spinner.getAttributes().get("maximum");
      int newValue = submittedValue + increment;

      if ((minimum == null || newValue >= minimum.intValue()) &&
         (maximum == null || newValue <= maximum.intValue()))
         return newValue;
      else
         return submittedValue;
   }
}
```


Using Child Components and Facets

The spinner discussed in the first half of this chapter is a simple component that nonetheless illustrates a number of useful techniques for implementing custom components. To illustrate more advanced custom component techniques, we switch to a more complicated component: a tabbed pane, as shown in Figure 9–8.

Figure 9–8 The Tabbed Pane Component

In Chapter 7, we showed you how to create an ad hoc tabbed pane with standard JSF tags such as `h:graphicImage` and `h:commandLink`. In this chapter we show you how to implement a tabbed pane component.

Of course, the advantage of a custom component over an ad hoc implementation is that the former is reusable. For example, we can easily reuse the tabbed pane component to create a tabbed pane just like the ad hoc version, as shown in Figure 9–9.

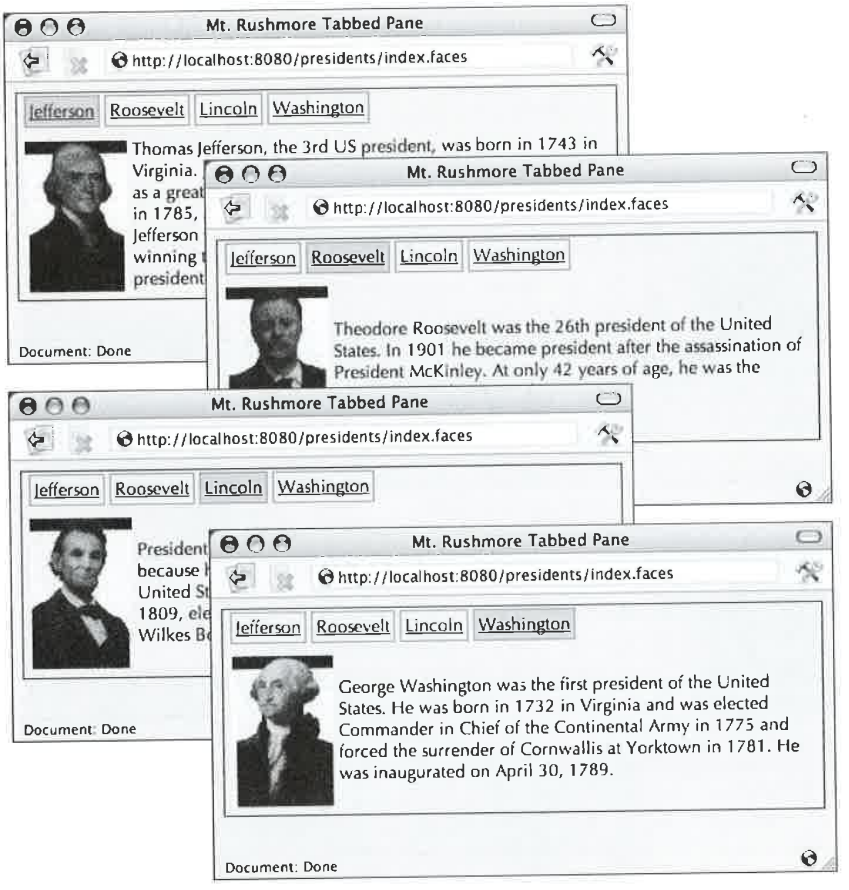


Figure 9–9 Reusing the Tabbed Pane Component

The tabbed pane component has some interesting features:

- You can *use CSS classes* for the tabbed pane as a whole and also for selected and unselected tabs.
- You *specify tabs* with `f:selectItem` tags (or `f:selectItems`), like the standard JSF menu and listbox tags specify menu or listbox items.

- You can *specify tabbed pane content* (for example, the picture and description in Figure 9–9) *with a URL* (which the tabbed pane renderer includes) *or a facet* (which the renderer renders). For example, you could specify the content for the Washington tab in Figure 9–9 as `/washington.jsp` or `washington`. If you use the former, the tabbed pane renderer includes the response from the specified JSP page. If you use the latter, the renderer looks for a facet of the tabbed pane named `washington`. (This use of facets is similar to the use of `header` and `footer` facets in the `h:dataTable` tag.)
- The tabbed pane renderer *uses the servlet request dispatcher* to include the content associated with a tab if that content is a URL.
- You can *add an action listener* to the tabbed pane. That listener is notified whenever a tab is selected.
- You can *localize tab text* by specifying keys from a resource bundle instead of the actual text displayed in the tab.
- The tabbed pane *uses hidden fields* to transmit the selected tab and its content from the client to the server.

Because the tabbed pane has so many features, there are several ways in which you can use it. Here's a simple use:

```
<corejsf:tabbedPane>
   <f:selectItem itemLabel="Jefferson"  itemValue="/jefferson.jsp"/>
   <f:selectItem itemLabel="Roosevelt"  itemValue="/roosevelt.jsp"/>
   <f:selectItem itemLabel="Lincoln"    itemValue="/lincoln.jsp"/>
   <f:selectItem itemLabel="Washington" itemValue="/washington.jsp"/>
</corejsf:tabbedPane>
```

The preceding code results in a rather plain-looking tabbed pane, as shown in Figure 9–10.

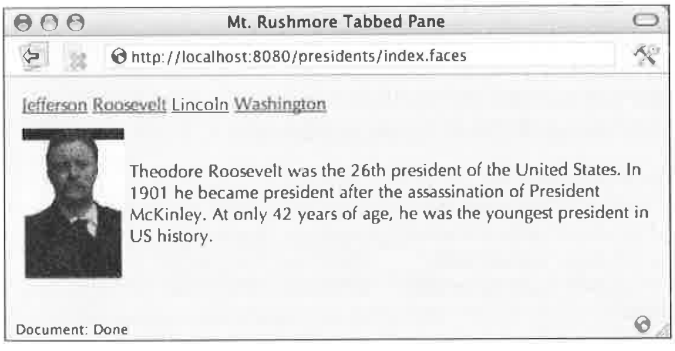


Figure 9–10 A Plain Tabbed Pane

To get the effect shown in Figure 9–9, you can use CSS styles, like this:

```
<corejsf:tabbedPane styleClass="tabbedPane"
      tabClass="tab" selectedTabClass="selectedTab">
   <f:selectItem itemLabel="Jefferson"  itemValue="/jefferson.jsp"/>
   <f:selectItem itemLabel="Roosevelt"  itemValue="/roosevelt.jsp"/>
   <f:selectItem itemLabel="Lincoln"    itemValue="/lincoln.jsp"/>
   <f:selectItem itemLabel="Washington" itemValue="/washington.jsp"/>
</corejsf:tabbedPane>
```

You can also use a single `f:selectItems` tag in lieu of multiple `f:selectitem` tags, like this:

```
<corejsf:tabbedPane styleClass="tabbedPane"
      tabClass="tab" selectedTabClass="selectedTab">
   <f:selectItems value="#{tabbedPaneBean.tabs}"/>
</corejsf:tabbedPane>
```

The preceding items are created by a bean:

```
public class TabbedPaneBean {
   private static final SelectItem[] tabs = {
      new SelectItem("/jefferson.jsp",  "Jefferson"),
      new SelectItem("/roosevelt.jsp",  "Roosevelt"),
      new SelectItem("/lincoln.jsp",    "Lincoln"),
      new SelectItem("/washington.jsp", "Washington"),
   };

   public SelectItem[] getTabs() {
      return tabs;
   }
}
```

In the previous example we directly specified the text displayed in each tab as select item labels: Jefferson, Roosevelt, etc. Before the tabbed pane renderer encodes a tab, it looks to see if those labels are keys in a resource bundle—if so, the renderer encodes the key's value. If the labels are not keys in a resource bundle, the renderer just encodes the labels as they are. You specify the resource bundle with the `resourceBundle` attribute, like this:

```
<corejsf:tabbedPane resourceBundle="com.corejsf.messages">
   <f:selectItem itemLabel="jeffersonTabKey"  itemValue="/jefferson.jsp"/>
   <f:selectItem itemLabel="rooseveltTabKey"  itemValue="/roosevelt.jsp"/>
   <f:selectItem itemLabel="lincolnTabKey"    itemValue="/lincoln.jsp"/>
   <f:selectItem itemLabel="washingtonTabKey" itemValue="/washington.jsp"/>
</corejsf:tabbedPane>
```

Notice the item labels—they are all keys in the messages resource bundle:


```
...
jeffersonTabText=Jefferson
rooseveltTabText=Roosevelt
lincolnTabText=Lincoln
washingtonTabText=Washington
...
```

There's one more way to specify tabs: with a facet, like this:

```
<corejsf:tabbedPane >

   ...
   <f:selectItem itemLabel="Jefferson"  itemValue="jefferson"/>
   ...
   <f:facet name="jefferson">
      <h:panelGrid columns="2">
         <h:graphicImage value="/images/jefferson.jpg"/>
         <h:outputText value="#{msgs.jeffersonDiscussion}"/>
      </h:panelGrid>
   </f:facet>
</corejsf:tabbedPane>
```

Up to now we've used URLs for item values. The contents of that URL are included by the tabbed pane renderer. But in the preceding code we specify a facet instead of a URL—the Jefferson select item's value is `jefferson`, which corresponds to a facet of the same name. Because we specified a facet, the tabbed pane renderer renders the facet instead of including content.

Finally, the tabbed pane component fires an action event when a user selects a tab. You can use the `f:actionListener` tag to add one or more action listeners, or you can specify a method that handles action events with the tabbed pane's `actionListener` attribute, like this:

```
<corejsf:tabbedPane ... actionListener="#{tabbedPaneBean.presidentSelected}">
   <f:selectItems value="#{tabbedPaneBean.tabs}"/>
</corejsf:tabbedPane>
```

Now that we have an overview of the tabbed pane component, let's take a closer look at how it implements advanced features. Here's what we'll cover in this section.

- "Saving and Restoring State" on page 412
- "Firing Action Events" on page 414

Processing `SelectItem` *Children*

The tabbed pane lets you specify tabs with `f:selectItem` or `f:selectItems`. Those tags create `UISelectItem` components and add them to the tabbed pane as children. Because the tabbed pane renderer has children and because it renders those children, it overrides `rendersChildren()` and `encodeChildren()`.

```
public boolean rendersChildren() {
    return true;
}
public void encodeChildren(FacesContext context, UIComponent component)
      throws java.io.IOException {
   // if the tabbedpane component has no children, this method is still called
   if (component.getChildCount() == 0) {
      return;
   }
   ...
   List items = com.corejsf.util.Renderers.getSelectItems(context, component);
   Iterator it = items.iterator();
   while (it.hasNext())
      encodeTab(context, writer, (SelectItem) it.next(), component);
      ...
   }
   ...
}
```

Generally, a component that processes its children contains code such as the following:

```
Iterator children = component.getChildren().iterator();
while (children.hasNext()) {
   UIComponent child = (UIComponent) children.next();
   processChild(context, writer, child, component);
}
```

However, our situation is more complex. Recall from Chapter 4 that you can specify a single select item, a collection of select items, an array of select items, or a map of Java objects as the value for the `f:selectItems` tag. Whenever your class processes children that are of type `SelectItem` or `SelectItems`, you need to deal with this mix of possibilities. The `com.corejsf.util.Renderers.getSelectItems` method accounts for all those data types and synthesizes them into a list of `SelectItem` objects. Here is the code for the helper method:


```
public static List getSelectItems(UIComponent component) {
   ArrayList list = new ArrayList();
   Iterator children = component.getChildren().iterator();
   while (children.hasNext()) {
      UIComponent child = (UIComponent) children.next();

      if (child instanceof UISelectItem) {
         Object value = ((UISelectItem) child).getValue();
         if (value == null) {
            UISelectItem item = (UISelectItem) child;
            list.add(new SelectItem(item.getItemValue(),
                  item.getItemLabel(),
                  item.getItemDescription(),
                  item.isItemDisabled()));
         } else if (value instanceof SelectItem) {
            list.add(value);
         }
      } else if (child instanceof UISelectItems) {
         Object value = ((UISelectItems) child).getValue();
         if (value instanceof SelectItem)
            list.add(value);
         else if (value instanceof SelectItem[])
            list.addAll(Arrays.asList((SelectItem[]) value));
         else if (value instanceof Collection)
            list.addAll((Collection) value);
         else if (value instanceof Map) {
            Iterator entries = ((Map) value).entrySet().iterator();
            while (entries.hasNext()) {
               Map.Entry entry = (Map.Entry) entries.next();
               list.add(new SelectItem(entry.getKey(),
                     "" + entry.getValue()));
            }
         }
      }
   }
   return list;
}
```

The encodeChildren method of the TabbedPaneRenderer calls this method and encodes each child into a tab. You will see the details in "Using Hidden Fields" on page 411.

Processing Facets

The tabbed pane lets you specify URLs or facet names for the content associated with a particular tag. The renderer accounts for that duality in its encodeEnd method:


```
    public void encodeEnd(FacesContext context, UIComponent component)
                                                  throws java.io.IOException {
       ResponseWriter writer = context.getResponseWriter();
       UITabbedPane tabbedPane = (UITabbedPane) component;
       String content = tabbedPane.getContent();
       ...
       if (content != null) {
          UIComponent facet = component.getFacet(content);
          if (facet != null) {
             if (facet.isRendered()) {
                facet.encodeBegin(context);
                if (facet.getRendersChildren())
                   facet.encodeChildren(context);
                facet.encodeEnd(context);
             }
          }
          else
             includePage(context, component);
       }
    }
    ...
}
```

The `UITabbedPane` class has a field `content` that stores the facet name or URL of the currently displayed tab.

The `encodeEnd` method checks to see whether the content of the currently selected tab is the name of a facet of this component. If so, it encodes the facet by invoking its `encodeBegin`, `encodeChildren`, and `encodeEnd` methods. Whenever a renderer renders its own children, it needs to take over this responsibility.

If the content of the current tab is not a facet, the renderer assumes the content is a URL and includes it, as shown in the following section.

`javax.faces.component.UIComponent`

- `UIComponent getFacet(String facetName)`

  Returns a reference to the facet if it exists. If the facet does not exist, the method returns `null`.

- `boolean getRendersChildren()`

  Returns a boolean that's `true` if the component renders its children, `false` otherwise. A component's `encodeChildren` method won't be called if this method does not return `true`. By default, `getRendersChildren` returns `false`.

- `boolean isRendered()`
  Returns the `rendered` property. The component is only rendered if the `rendered` property is `true`.

Including Content

As you saw in the preceding section, the tabbed pane renderer's `encodeEnd` method calls the `includePage` method when the content is described by a URL. Here's the `includePage` method:

```
private void includePage(FacesContext fc, UIComponent component) {
   ExternalContext ec = fc.getExternalContext();
   ServletContext sc = (ServletContext) ec.getContext();
   UITabbedPane tabbedPane = (UITabbedPane) component;
   String content = tabbedPane.getContent();

   ServletRequest request = (ServletRequest) ec.getRequest();
   ServletResponse response = (ServletResponse) ec.getResponse();
   try {
      sc.getRequestDispatcher(content).include(request, response);
   }
   catch(Exception ex) {
      System.out.println("Couldn't load page: " + content);
   }
}
```

The `includePage` method uses the servlet request dispatcher to include the response from the specified URL. The request dispatcher reads the requested URL and writes its content to the response writer.

`javax.servlet.ServletContext`

- `RequestDispatcher getRequestDispatcher(String path)`
  Returns a reference to a request dispatcher, given a path to a resource.

`javax.servlet.RequestDispatcher`

- `void include(ServletRequest request, ServletResponse response) throws IllegalStateException, IOException, ServletException`
  Includes the content of some resource. The path to that resource is passed to the `RequestDispatcher` constructor.

Encoding CSS Styles

You can support CSS styles in two steps:

- Add an attribute to the tag library descriptor.
- Encode the component's attribute in your renderer's `encode` methods.

First, we add attributes `styleClass`, `tabClass`, and `selectedTabClass` to the TLD:

```
<taglib>
    ...
    <tag>
       ...
       <attribute>
         <name>styleClass</name>
         <description>The CSS style for this component</description>
       </attribute>
       ...
  </tag>
</taglib>
```

We then write attributes for the CSS classes:

```
public class TabbedPaneRenderer extends Renderer {
   ...
   public void encodeBegin(FacesContext context, UIComponent component)
         throws java.io.IOException {
      ResponseWriter writer = context.getResponseWriter();
      writer.startElement("table", component);

      String styleClass = (String) component.getAttributes().get("styleClass");
      if (styleClass != null)
         writer.writeAttribute("class", styleClass, "styleClass");

      writer.write("\n"); // to make generated HTML easier to read
   }
   public void encodeChildren(FacesContext context, UIComponent component)
            throws java.io.IOException {
      ...
      encodeTab(context, responseWriter, selectItem, component);
      ...
   }
   ...
   private void encodeTab(FacesContext context, ResponseWriter writer,
         SelectItem item, UIComponent component) throws java.io.IOException {
      ...
      String tabText = getLocalizedTabText(component, item.getLabel());
      ...
```



```
        String tabClass = null;
        if (content.equals(selectedContent))
           tabClass = (String) component.getAttributes().get("selectedTabClass");
        else
           tabClass = (String) component.getAttributes().get("tabClass");

        if (tabClass != null)
           writer.writeAttribute("class", tabClass, "tabClass");
        ...
     }
     ...
  }
```

We encode the styleClass attribute for the tabbed pane's outer table and encode the tabClass and selectedTabClass attribute for each individual tag.

javax.faces.model.SelectItem

- Object getValue()

  Returns the select item's value.

Using Hidden Fields

Each tab in the tabbed pane is encoded as a hyperlink, like this:

```
<a href="#" onclick="document.forms[formId][clientId].value=content;
   document.forms[formId].submit();"/>
```

When a user clicks on a particular hyperlink, the form is submitted (The href value corresponds to the current page). Of course, the server needs to know which tab was selected. This information is stored in a *hidden field* that is placed after all the tabs:

```
<input type="hidden" name="clientId"/>
```

When the form is submitted, the name and value of the hidden field are sent back to the server, allowing the decode method to activate the selected tab.

The renderer's encodeTab method produces the hyperlink tags. The encodeEnd method calls encodeHiddenFields(), which encodes the hidden field. You can see the details in Listing 9–18 on page 414.

When the tabbed pane renderer decodes the incoming request, it uses the request parameter, associated with the hidden field, to set the tabbed pane component's content.

```
public void decode(FacesContext context, UIComponent component) {
   Map requestParams = context.getExternalContext().getRequestParameterMap();
   String clientId = component.getClientId(context);
   String content = (String) (requestParams.get(clientId));
```

```
        if (content != null && !content.equals("")) {
           UITabbedPane tabbedPane = (UITabbedPane) component;
           tabbedPane.setContent(content);
        }
        ...
     }
     ...
  }
```

Saving and Restoring State

The UITabbedPane class has an instance field that stores the facet name or URL of the currently displayed tab. Whenever your components have instance fields and there is a possibility that they are used in a web application that saves state on the client, then you need to implement the saveState and restoreState methods of the StateHolder interface.

These methods have the following form:

```
public Object saveState(FacesContext context) {
   Object values[] = new Object[n];
   values[0] = super.saveState(context);
   values[1] = instance field #1;
   values[2] = instance field #2;
   ...
   return values;
}

public void restoreState(FacesContext context, Object state) {
   Object values[] = (Object[]) state;
   super.restoreState(context, values[0]);
   instance field #1 = (Type) values[1];
   instance field #2 = (Type) values[2];
   ...
}
```

Listing 9–17 shows how the UITabbedPane class saves and restores its state.

To test why state saving is necessary, run this experiment:

- Comment out the saveState and restoreState methods.
- Activate client-side state saving by adding these lines to web.xml:

```
<context-param>
   <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
   <param-value>client</param-value>
</context-param>
```

- Add a button to the index.jsp page of the bears application:

```
<h:commandButton value="Redisplay"/>
```


- Run the application and click on a tab.
- Click the "Redisplay" button. The current page is redisplayed, but no tab is selected!

This problem occurs because the state of the page is saved on the client, encoded as the value of a hidden field. When the page is redisplayed, a new `UITabbedPane` object is constructed and its `restoreState` method is called. If the `UITabbedPane` class does not override the `restoreState` method, the `content` field is not restored.

NOTE: In Chapter 6, you saw that you could save the state of converters and validators simply by making the converter or validator class serializable. This approach does not work for components—you must use the `StateHolder` methods.

TIP: If you store all of your component state as *attributes*, you don't have to implement the `saveState` and `restoreState` methods because component attributes are automatically saved by the JSF implementation. For example, the tabbed pane can simply use a "`content`" attribute instead of the `content` field.

Then you don't need the `UITabbedPane` class at all. Simply use the `UICommand` superclass and declare the component class like this:

```
<component>
   <component-type>com.corejsf.TabbedPane</component-type>
   <component-class>javax.faces.component.UICommand</component-class>
</component>
```

Frankly, that's what we do in our own code. You will find several examples in Chapters 11 and 12. The standard JSF components use the more elaborate mechanism to minimize the size of the state information.

Listing 9–17 bears/WEB-INF/classes/com/corejsf/UITabbedPane.java

```
1. package com.corejsf;
2.
3. import javax.faces.component.UICommand;
4. import javax.faces.context.FacesContext;
5.
6. public class UITabbedPane extends UICommand {
7.    private String content;
8.
9.    public String getContent() { return content; }
```

Listing 9–17 bears/WEB-INF/classes/com/corejsf/UITabbedPane.java (cont.)

```
10.   public void setContent(String newValue) { content = newValue; }
11.
12.   public Object saveState(FacesContext context) {
13.      Object values[] = new Object[2];
14.      values[0] = super.saveState(context);
15.      values[1] = content;
16.      return values;
17.   }
18.
19.   public void restoreState(FacesContext context, Object state) {
20.      Object values[] = (Object[]) state;
21.      super.restoreState(context, values[0]);
22.      content = (String) values[1];
23.   }
24. }
```

Firing Action Events

When your component handles action events or actions, you need to take the following steps:

- Your component should extend `UICommmand`.
- You need to queue an `ActionEvent` in the `decode` method of your renderer.

The tabbed pane component fires an action event when a user selects one of its tabs. That action is queued by `TabbedPaneRenderer` in the `decode` method.

```
public void decode(FacesContext context, UIComponent component) {
   ...
   UITabbedPane tabbedPane = (UITabbedPane) component;
   ...
   component.queueEvent(new ActionEvent(tabbedPane));
}
```

This completes the discussion of the `TabbedPaneRenderer` class. You will find the complete code in Listing 9–18. The `TabbedPaneTag` class is as boring as ever, and we do not show it here.

Listing 9–18 bears/WEB-INF/classes/com/corejsf/TabbedPaneRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Iterator;
5. import java.util.List;
6. import java.util.Map;
7. import java.util.logging.Level;
```


Listing 9–18 bears/WEB-INF/classes/com/corejsf/TabbedPaneRenderer.java (cont.)

```
8. import java.util.logging.Logger;
9. import javax.faces.component.UIComponent;
10. import javax.faces.context.ExternalContext;
11. import javax.faces.context.FacesContext;
12. import javax.faces.context.ResponseWriter;
13. import javax.faces.event.ActionEvent;
14. import javax.faces.model.SelectItem;
15. import javax.faces.render.Renderer;
16. import javax.servlet.ServletContext;
17. import javax.servlet.ServletException;
18. import javax.servlet.ServletRequest;
19. import javax.servlet.ServletResponse;
20.
21. // Renderer for the UITabbedPane component
22.
23. public class TabbedPaneRenderer extends Renderer {
24.    private static Logger logger = Logger.getLogger("com.corejsf.util");
25.
26.    // By default, getRendersChildren() returns false, so encodeChildren()
27.    // won't be invoked unless we override getRendersChildren() to return true
28.
29.    public boolean getRendersChildren() {
30.       return true;
31.    }
32.
33.    // The decode method gets the value of the request parameter whose name
34.    // is the client Id of the tabbedpane component. The request parameter
35.    // is encoded as a hidden field by encodeHiddenField, which is called by
36.    // encodeEnd. The value for the parameter is set by JavaScript generated
37.    // by the encodeTab method. It is the name of a facet or a JSP page.
38.
39.    // The decode method uses the request parameter value to set the
40.    // tabbedpane component's content attribute.
41.    // Finally, decode() queues an action event that's fired to registered
42.    // listeners in the Invoke Application phase of the JSF lifecycle. Action
43.    // listeners can be specified with the <corejsf:tabbedpane>'s actionListener
44.    // attribute or with <f:actionListener> tags in the body of the
45.    // <corejsf:tabbedpane> tag.
46.
47.    public void decode(FacesContext context, UIComponent component) {
48.       Map requestParams = context.getExternalContext().getRequestParameterMap();
49.       String clientId = component.getClientId(context);
50.
51.       String content = (String) (requestParams.get(clientId));
52.       if (content != null && !content.equals("")) {
53.          UITabbedPane tabbedPane = (UITabbedPane) component;
```


Listing 9-18 bears/WEB-INF/classes/com/corejsf/TabbedPaneRenderer.java (cont.)

```
54.         tabbedPane.setContent(content);
55.      }
56.
57.      component.queueEvent(new ActionEvent(component));
58.   }
59.
60.   // The encodeBegin method writes the starting <table> HTML element
61.   // with the CSS class specified by the <corejsf:tabbedpane>'s styleClass
62.   // attribute (if supplied)
63.
64.   public void encodeBegin(FacesContext context, UIComponent component)
65.         throws java.io.IOException {
66.      ResponseWriter writer = context.getResponseWriter();
67.      writer.startElement("table", component);
68.
69.      String styleClass = (String) component.getAttributes().get("styleClass");
70.      if (styleClass != null)
71.         writer.writeAttribute("class", styleClass, null);
72.
73.      writer.write("\n"); // to make generated HTML easier to read
74.   }
75.
76.   // encodeChildren() is invoked by the JSF implementation after encodeBegin().
77.   // The children of the <corejsf:tabbedpane> component are UISelectItem
78.   // components, set with one or more <f:selectItem> tags or a single
79.   // <f:selectItems> tag in the body of <corejsf:tabbedpane>
80.
81.   public void encodeChildren(FacesContext context, UIComponent component)
82.         throws java.io.IOException {
83.      // if the tabbedpane component has no children, this method is still
84.      // called
85.      if (component.getChildCount() == 0) {
86.         return;
87.      }
88.
89.      ResponseWriter writer = context.getResponseWriter();
90.      writer.startElement("thead", component);
91.      writer.startElement("tr", component);
92.      writer.startElement("th", component);
93.
94.      writer.startElement("table", component);
95.      writer.startElement("tbody", component);
96.      writer.startElement("tr", component);
97.
98.      List items = com.corejsf.util.Renderers.getSelectItems(component);
```


Listing 9–18 bears/WEB-INF/classes/com/corejsf/TabbedPaneRenderer.java (cont.)

```
99.      Iterator it = items.iterator();
100.     while (it.hasNext())
101.        encodeTab(context, writer, (SelectItem) it.next(), component);
102.
103.     writer.endElement("tr");
104.     writer.endElement("tbody");
105.     writer.endElement("table");
106.
107.     writer.endElement("th");
108.     writer.endElement("tr");
109.     writer.endElement("thead");
110.     writer.write("\n"); // to make generated HTML easier to read
111.  }
112.
113.  // encodeEnd() is invoked by the JSF implementation after encodeChildren().
114.  // encodeEnd() writes the table body and encodes the tabbedpane's content
115.  // in a single table row.
116.
117.  // The content for the tabbed pane can be specified as either a URL for
118.  // a JSP page or a facet name, so encodeEnd() checks to see if it's a facet;
119.  // if so, it encodes it; if not, it includes the JSP page
120.
121.  public void encodeEnd(FacesContext context, UIComponent component)
122.        throws java.io.IOException {
123.     ResponseWriter writer = context.getResponseWriter();
124.     UITabbedPane tabbedPane = (UITabbedPane) component;
125.     String content = tabbedPane.getContent();
126.
127.     writer.startElement("tbody", component);
128.     writer.startElement("tr", component);
129.     writer.startElement("td", component);
130.
131.     if (content != null) {
132.        UIComponent facet = component.getFacet(content);
133.        if (facet != null) {
134.           if (facet.isRendered()) {
135.              facet.encodeBegin(context);
136.              if (facet.getRendersChildren())
137.                 facet.encodeChildren(context);
138.              facet.encodeEnd(context);
139.           }
140.        } else
141.           includePage(context, component);
142.     }
143.
```


Listing 9–18 bears/WEB-INF/classes/com/corejsf/TabbedPaneRenderer.java (cont.)

```
144.        writer.endElement("td");
145.        writer.endElement("tr");
146.        writer.endElement("tbody");
147.
148.        // Close off the column, row, and table elements
149.        writer.endElement("table");
150.
151.        encodeHiddenField(context, writer, component);
152.    }
153.
154.    // The encodeHiddenField method is called at the end of encodeEnd().
155.    // See the decode method for an explanation of the field and its value.
156.
157.    private void encodeHiddenField(FacesContext context, ResponseWriter writer,
158.            UIComponent component) throws java.io.IOException {
159.        // write hidden field whose name is the tabbedpane's client Id
160.        writer.startElement("input", component);
161.        writer.writeAttribute("type", "hidden", null);
162.        writer.writeAttribute("name", component.getClientId(context), null);
163.        writer.endElement("input");
164.    }
165.
166.    // encodeTab, which is called by encodeChildren, encodes an HTML anchor
167.    // element with an onclick attribute which sets the value of the hidden
168.    // field encoded by encodeHiddenField and submits the tabbedpane's enclosing
169.    // form. See the decode method for more information about the hidden field.
170.    // encodeTab also writes out a class attribute for each tab corresponding
171.    // to either the tabClass attribute (for unselected tabs) or the
172.    // selectedTabClass attribute (for the selected tab).
173.
174.    private void encodeTab(FacesContext context, ResponseWriter writer,
175.            SelectItem item, UIComponent component) throws java.io.IOException {
176.        String tabText = getLocalizedTabText(component, item.getLabel());
177.        String content = (String) item.getValue();
178.
179.        writer.startElement("td", component);
180.        writer.startElement("a", component);
181.        writer.writeAttribute("href", "#", "href");
182.
183.        String clientId = component.getClientId(context);
184.        String formId = com.corejsf.util.Renderers.getFormId(context, component);
185.
186.        writer.writeAttribute("onclick",
187.        // write value for hidden field whose name is the tabbedpane's client Id
188.
```


Listing 9–18 bears/WEB-INF/classes/com/corejsf/TabbedPaneRenderer.java (cont.)

```
            "document.forms['" + formId + "']['" + clientId + "'].value='"
                  + content + "'; " +

                  // submit form in which the tabbedpane resides
                  "document.forms['" + formId + "'].submit(); ", null);

      UITabbedPane tabbedPane = (UITabbedPane) component;
      String selectedContent = tabbedPane.getContent();

      String tabClass = null;
      if (content.equals(selectedContent))
         tabClass = (String) component.getAttributes().get("selectedTabClass");
      else
         tabClass = (String) component.getAttributes().get("tabClass");

      if (tabClass != null)
         writer.writeAttribute("class", tabClass, null);

      writer.write(tabText);

      writer.endElement("a");
      writer.endElement("td");
      writer.write("\n"); // to make generated HTML easier to read
   }

   // Text for the tabs in the tabbedpane component can be specified as
   // a key in a resource bundle, or as the actual text that's displayed
   // in the tab. Given that text, the getLocalizedTabText method tries to
   // retrieve a value from the resource bundle specified with the
   // <corejsf:tabbedpane>'s resourceBundle attribute. If no value is found,
   // getLocalizedTabText just returns the string it was passed.

   private String getLocalizedTabText(UIComponent tabbedPane, String key) {
      String bundle = (String) tabbedPane.getAttributes().get("resourceBundle");
      String localizedText = null;

      if (bundle != null) {
         localizedText = com.corejsf.util.Messages.getString(bundle, key, null);
      }
      if (localizedText == null)
         localizedText = key;
      // The key parameter was not really a key in the resource bundle,
      // so just return the string as is
      return localizedText;
   }
```

Listing 9–18 bears/WEB-INF/classes/com/corejsf/TabbedPaneRenderer.java (cont.)

```
234.
235.    // includePage uses the servlet request dispatcher to include the page
236.    // corresponding to the selected tab.
237.
238.    private void includePage(FacesContext fc, UIComponent component) {
239.       ExternalContext ec = fc.getExternalContext();
240.       ServletContext sc = (ServletContext) ec.getContext();
241.       UITabbedPane tabbedPane = (UITabbedPane) component;
242.       String content = tabbedPane.getContent();
243.
244.       ServletRequest request = (ServletRequest) ec.getRequest();
245.       ServletResponse response = (ServletResponse) ec.getResponse();
246.       try {
247.          sc.getRequestDispatcher(content).include(request, response);
248.       } catch (ServletException ex) {
249.          logger.log(Level.WARNING, "Couldn't load page: " + content, ex);
250.       } catch (IOException ex) {
251.          logger.log(Level.WARNING, "Couldn't load page: " + content, ex);
252.       }
253.    }
254. }
```

Using the Tabbed Pane

The bears application shown in Figure 9–8 on page 401 uses a bean to specify the URLs for the tabs. The bean code is in Listing 9–19. Companion code also contains a presidents application that specifies the tabs with facets.

The directory structure for the application is shown in Figure 9–11. Listing 9–20 shows the index.jsp page, and Listing 9–21 shows one of the pages that make up the tab content. The other pages look similar and are omitted. Listing 9–22 through Listing 9–25 show the tag library descriptor, tag class, faces configuration file and the stylesheet for the tabbed pane application.

You have now seen how to implement custom components. We covered all essential issues that you will encounter as you develop your own components. The code in this chapter should make a good starting point for your component implementations.

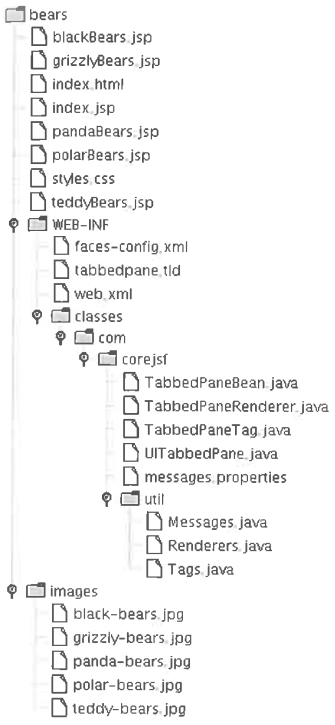


Figure 9–11 The Bears Directory Structure

Listing 9–19 bears/WEB-INF/classes/com/corejsf/TabbedPaneBean.java

```
1. package com.corejsf;
2.
3. import javax.faces.model.SelectItem;
4.
5. public class TabbedPaneBean {
6.    private static final SelectItem[] tabs = {
7.       new SelectItem("/blackBears.jsp",   "blackTabText"),
8.       new SelectItem("/grizzlyBears.jsp", "grizzlyTabText"),
```


Listing 9-19 bears/WEB-INF/classes/com/corejsf/TabbedPaneBean.java (cont.)

```
9.        new SelectItem("/polarBears.jsp",   "polarTabText"),
10.       new SelectItem("/pandaBears.jsp",   "pandaTabText"),
11.       new SelectItem("/teddyBears.jsp",   "teddyTabText"),
12.    };
13.
14.    public SelectItem[] getTabs() {
15.       return tabs;
16.    }
17. }
```

Listing 9-20 bears/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <%@ taglib uri="http://corejsf.com/tabbedpane" prefix="corejsf" %>
5.
6.    <f:view>
7.       <head>
8.          <link href="styles.css" rel="stylesheet" type="text/css"/>
9.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
10.         <title>
11.            <h:outputText value="#{msgs.windowTitle}"/>
12.         </title>
13.      </head>
14.      <body>
15.         <h:form>
16.            <corejsf:tabbedPane styleClass="tabbedPane" tabClass="tab"
17.               selectedTabClass="selectedTab"
18.               resourceBundle="com.corejsf.messages">
19.               <f:selectItems value="#{tabbedPaneBean.tabs}"/>
20.            </corejsf:tabbedPane>
21.         </h:form>
22.      </body>
23.   </f:view>
24. </html>
```


Listing 9–21 bears/blackBears.jsp

```
1. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
2. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.
4. <f:subview id="blackBear">
5.    <h:panelGrid columns='2' columnClasses='bearDiscussionColumn'>
6.       <h:graphicImage value='/images/black-bears.jpg'/>
7.       <h:outputText value='#{msgs.blackBearDiscussion}'
8.                 styleClass='tabbedPaneContent'/>
9.    </h:panelGrid>
10. </f:subview>
```

Listing 9–22 bears/WEB-INF/tabbedpane.tld

```
1. <?xml version="1.0" encoding="ISO-8859-1" ?>
2.
3. <!DOCTYPE taglib
4.  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
5.  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
6.
7. <taglib>
8.    <tlib-version>0.03</tlib-version>
9.    <jsp-version>1.2</jsp-version>
10.   <short-name>corejsf</short-name>
11.   <uri>http://corejsf/components</uri>
12.   <description>A library containing a tabbed pane</description>
13.
14.   <tag>
15.      <name>tabbedPane</name>
16.      <tag-class>com.corejsf.TabbedPaneTag</tag-class>
17.       <body-content>JSP</body-content>
18.       <description>A tag for a tabbed pane component</description>
19.
20.       <attribute>
21.         <name>id</name>
22.         <required>false</required>
23.         <rtexprvalue>false</rtexprvalue>
24.         <description>Component id of this component</description>
25.       </attribute>
26.
```


Listing 9–22 bears/WEB-INF/tabbedpane.tld (cont.)

```
27.     <attribute>
28.       <name>binding</name>
29.       <required>false</required>
30.       <rtexprvalue>false</rtexprvalue>
31.       <description>Component reference expression for this component</descrip-
tion>
32.     </attribute>
33.
34.     <attribute>
35.       <name>rendered</name>
36.       <required>false</required>
37.       <rtexprvalue>false</rtexprvalue>
38.       <description>
39.         A flag indicating whether or not this component should be rendered.
40.        If not specified, the default value is true.
41.       </description>
42.     </attribute>
43.
44.     <attribute>
45.       <name>style</name>
46.       <required>false</required>
47.       <rtexprvalue>false</rtexprvalue>
48.       <description>The CSS style for this component</description>
49.     </attribute>
50.
51.     <attribute>
52.       <name>styleClass</name>
53.       <required>false</required>
54.       <rtexprvalue>false</rtexprvalue>
55.       <description>The CSS class for this component</description>
56.     </attribute>
57.
58.     <attribute>
59.       <name>tabClass</name>
60.       <required>false</required>
61.       <rtexprvalue>false</rtexprvalue>
62.       <description>The CSS class for unselected tabs</description>
63.     </attribute>
64.
```


Listing 9–22 bears/WEB-INF/tabbedpane.tld (cont.)

```
65.       <attribute>
66.         <name>selectedTabClass</name>
67.         <required>false</required>
68.         <rtexprvalue>false</rtexprvalue>
69.         <description>The CSS class for the selected tab</description>
70.       </attribute>
71.
72.       <attribute>
73.         <name>resourceBundle</name>
74.         <required>false</required>
75.         <rtexprvalue>false</rtexprvalue>
76.         <description>
77.           The resource bundle used to localize select item labels
78.         </description>
79.       </attribute>
80.
81.       <attribute>
82.         <name>actionListener</name>
83.         <required>false</required>
84.         <rtexprvalue>false</rtexprvalue>
85.         <description>
86.           A method reference that's called when a tab is selected
87.         </description>
88.       </attribute>
89.   </tag>
90. </taglib>
```

Listing 9–23 tabbedpane/WEB-INF/classes/com/corejsf/TabbedPaneTag.java

```
1. package com.corejsf;
2.
3. import javax.faces.application.Application;
4. import javax.faces.context.FacesContext;
5. import javax.faces.component.UIComponent;
6. import javax.faces.el.MethodBinding;
7. import javax.faces.event.ActionEvent;
8. import javax.faces.webapp.UIComponentBodyTag;
9.
```


Listing 9–23 tabbedpane/WEB-INF/classes/com/corejsf/TabbedPaneTag.java (cont.)

```
import com.corejsf.util.Tags;

// This tag supports the following attributes
//
// binding (supported by UIComponentBodyTag)
// id (supported by UIComponentBodyTag)
// style (supported by UIComponentBodyTag)
// rendered (supported by UIComponentBodyTag)
// styleClass
// tabClass
// selectedTabClass
// resourceBundle
// actionListener

public class TabbedPaneTag extends UIComponentBodyTag {
   private String style, styleClass, tabClass, selectedTabClass, resourceBundle,
                  actionListener;

   public String getRendererType () {
      return "TabbedPaneRenderer";
   }
   public String getComponentType() {
      return "Tabbed Pane";
   }

   // tabClass attribute
   public String getTabClass() { return tabClass; }
   public void setTabClass(String tabClass) { this.tabClass= tabClass; }

   // selectedTabClass attribute
   public String getSelectedTabClass() { return selectedTabClass; }
   public void setSelectedTabClass(String selectedTabClass) {
       this.selectedTabClass= selectedTabClass;
   }

   // styleClass attribute
```

Listing 9–23 tabbedpane/WEB-INF/classes/com/corejsf/TabbedPaneTag.java (cont.)

```
46.    public String getStyle() { return style; }
47.    public void setStyle(String style) { this.style= style; }
48.
49.    // styleClass attribute
50.    public String getStyleClass() { return styleClass; }
51.    public void setStyleClass(String styleClass) { this.styleClass = styleClass; }
52.
53.    // resourceBundle attribute
54.    public String getResourceBundle() { return resourceBundle; }
55.    public void setResourceBundle(String resourceBundle) {
56.        this.resourceBundle = resourceBundle;
57.    }
58.
59.    // actionListener attribute
60.    public String getActionListener() { return resourceBundle; }
61.    public void setActionListener(String actionListener) {
62.        this.actionListener = actionListener;
63.    }
64.
65.    protected void setProperties(UIComponent component) {
66.       // make sure you always call the superclass
67.       super.setProperties(component);
68.
69.       com.corejsf.util.Tags.setComponentAttribute(component, "style", style);
70.       com.corejsf.util.Tags.setComponentAttribute(component, "styleClass",
71.                                                     styleClass);
72.       com.corejsf.util.Tags.setComponentAttribute(component, "tabClass",
73.                                                     tabClass);
74.       com.corejsf.util.Tags.setComponentAttribute(component, "selectedTabClass",
75.                                                     selectedTabClass);
76.       com.corejsf.util.Tags.setComponentAttribute(component, "resourceBundle",
77.                                                     resourceBundle);
78.       com.corejsf.util.Tags.setComponentAttribute(component, "actionListener",
79.                                                     actionListener);
80.    }
81. }
```

Listing 9–24 tabbedpane/WEB-INF/faces-config.txt

```
<?xml version="1.0"?>

<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
   <managed-bean>
      <managed-bean-name>tabbedPaneBean</managed-bean-name>
      <managed-bean-class>com.corejsf.TabbedPaneBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>

   <navigation-rule>
      <from-view-id>/index.jsp</from-view-id>
         <navigation-case>
            <to-view-id>/welcome.jsp</to-view-id>
         </navigation-case>
   </navigation-rule>

   <component>
      <description>A tabbed pane</description>
      <component-type>Tabbed Pane</component-type>
      <component-class>com.corejsf.UITabbedPane</component-class>
   </component>

   <!-- order is important within elements -->
   <render-kit>
      <renderer>
         <component-family>javax.faces.Command</component-family>
         <renderer-type>TabbedPaneRenderer</renderer-type>
         <renderer-class>com.corejsf.TabbedPaneRenderer</renderer-class>
      </renderer>
   </render-kit>
</faces-config>
```


Listing 9–25 tabbedpane/styles.css

```
1. body {
2.    background: #ccc;
3. }
4. .emphasis {
5.    font-size: 3.5em;
6.    font-style: italic;
7. }
8. .tabbedPane {
9.    vertical-align: top;
10.   border: thin solid Blue;
11.   width: 96%;
12.   height 96%;
13. }
14. .tab {
15.   vertical-align: top;
16.   padding: 3px;
17.   border: thin solid Red;
18.   color: Yellow;
19.   background: LightSlateGray;
20. }
21. .selectedTab {
22.   vertical-align: top;
23.   padding: 3px;
24.   border: thin solid Black;
25.   color: LightSlateGray;
26.   background: Yellow;
27. }
28. .tabbedPaneContent {
29.   vertical-align: top;
30.   width: *;
31.   height: *;
32. }
```


External Services

Topics in This Chapter

Chapter 10

In this chapter, you learn how to access external services from your JSF application. We show you how to connect to databases, directories, and web services. Our primary interest lies in the clean separation between the application logic and the configuration of resources.

Accessing a Database

In this section, we assume that you are familiar with basic database commands in SQL (the Structured Query Language), as well as the JDBC (Java Database Connectivity) API. A good introduction to these topics can be found in *Horstmann & Cornell, Core Java, Vol. 2, ch. 4, Sun Microsystems Press, 2002.* For your convenience, here is a brief refresher of the basics.

Issuing SQL Statements

To issue SQL statements to a database, you need a *connection* object. There are various methods of obtaining a connection. The most elegant one is to make a *directory lookup*, using the Java Naming and Directory Interface (JNDI).

```
Context ctx = new InitialContext();
DataSource source = (DataSource) ctx.lookup("java:comp/env/jdbc/mydb");
Connection conn = source.getConnection();
```


Later in this chapter we show you how to configure the data source in the Tomcat container. For now, let's assume that the data source is properly configured to connect to your favorite database.

Once you have the Connection object, you create a Statement object that you use to send SQL statements to the database. You use the executeUpdate method for SQL statements that update the database, and the executeQuery method for queries that return a result set.

```
Statement stat = conn.createStatement();
stat.executeUpdate("INSERT INTO Users VALUES ('troosevelt', 'jabberwock')");
ResultSet result = stat.executeQuery("SELECT * FROM Users");
```

The ResultSet class has an unusual iteration protocol. You first call the next method to advance the cursor to the first row. (The next method returns false if no further rows are available.) Then you call the getString method to get a field value as a string. For example,

```
while (result.next()) {
   username = result.getString("username");
   password = result.getString("password");
   . . .
}
```

When you are done using the database, be certain that you close the connection. To ensure that the connection is closed under all circumstances, even when an exception occurs, wrap the query code inside a try/finally block, like this:

```
Connection conn = source.getConnection();
try {
   . . .
}
finally {
   conn.close();
}
```

Of course, there is much more to the JDBC API, but these simple concepts are sufficient to get you started.

Connection Management

One of the more vexing issues for the web developer is the management of database connections. There are two conflicting concerns. First, opening a connection to a database can be time consuming. Several seconds may elapse for the processes of connecting, authenticating, and acquiring resources to be completed. Thus, you cannot simply open a new connection for every page request.

On the flip side, you cannot keep open a huge number of connections to the database. Connections consume resources, both in the client program and in the database server. Commonly, a database puts a limit on the maximum number of concurrent connections that it allows. Thus, your application cannot simply open a connection whenever a user logs in and leave it open until the user logs off. After all, your user might walk away and never log off.

One common mechanism for solving these concerns is to *pool* the database connections. A connection pool holds database connections that are already opened. Application programs obtain connections from the pool. When the connections are no longer needed, they are returned to the pool, but they are not closed. Thus, the pool minimizes the time lag of establishing database connections.

Implementing a database connection pool is not easy, and it certainly should not be the responsibility of the application programmer. As of version 2.0, JDBC supports pooling in a pleasantly transparent way. When you receive a pooled `Connection` object, it is actually instrumented so that its `close` method merely returns it to the pool. It is up to the application server to set up the pool and to give you a data source whose `getConnection` method yields pooled connections.

Each application server has its own way of configuring the database connection pool. The details are not part of any Java standard—the JDBC specification is completely silent on this issue. In the next section, we describe how to configure Tomcat for connection pooling. The basic principle is the same with other application servers, but of course the details may differ considerably.

To maintain the pool, it is still essential that you close every connection object when you are done using it. Otherwise the pool will run dry, and new physical connections to the database will need to be opened. Properly closing connections is the topic of the next section.

Plugging Connection Leaks

Consider this simple sequence of statements:

```
DataSource source = ...
Connection conn = source.getConnection();
Statement stat = conn.createStatement();
String command = "INSERT INTO Users VALUES ('troosevelt', 'jabberwock')";
stat.executeUpdate(command);
conn.close();
```

The code looks clean—we open a connection, issue a command, and immediately close the connection. But there is a fatal flaw. If one of the method calls throws an exception, the call to the `close` method never happens!

In that case, an irate user may resubmit the request many times in frustration, leaking another connection object with every click.

To overcome this issue, *always* place the call to close inside a finally block:

```
DataSource source = ...
Connection conn = source.getConnection();
try {
   Statement stat = conn.createStatement();
   String command = "INSERT INTO Users VALUES ('troosevelt', 'jabberwock')";
   stat.executeUpdate(command);
}
finally {
   conn.close();
}
```

This simple rule completely solves the problem of leaking connections.

The rule is most effective if you *do not combine* this try/finally construct with any other exception handling code. In particular, do not attempt to catch a SQLException in the same try block:

```
// we recommend that you do NOT do this
Connection conn = null;
try {
   conn = source.getConnection();
   Statement stat = conn.createStatement();
   String command = "INSERT INTO Users VALUES ('troosevelt', 'jabberwock')";
   stat.executeUpdate(command);
}
catch (SQLException) {
   // log error
}
finally {
   conn.close(); // ERROR
}
```

That code has two subtle mistakes. First, if the call to getConnection throws an exception, then conn is still null, and you can't call close. Moreover, the call to close can also throw a SQLException. You could clutter up the finally clause with more code, but the result is a mess. Instead, use two separate try blocks:

```
// we recommend that you use separate try blocks
try {
   Connection conn = source.getConnection();
   try {
      Statement stat = conn.createStatement();
      String command = "INSERT INTO Users VALUES ('troosevelt', 'jabberwock')";
      stat.executeUpdate(command);
```



```
      }
      finally {
         conn.close();
      }
   }
   catch (SQLException) {
      // log error
   }
```

The inner try block ensures that the connection is closed. The outer try block ensures that the exception is logged.

NOTE: Of course, you can also tag your method with throws SQLException and leave the outer try block to the caller. That is often the best solution.

Using Prepared Statements

A common optimization technique for JDBC programs is the use of the PreparedStatement class. You use a *prepared statement* to speed up database operations if your code issues the same type of query multiple times. Consider the lookup of user passwords. You will repeatedly need to issue a query of the form

```
SELECT password FROM Users WHERE username=...
```

A prepared statement asks the database to precompile a query, that is, parse the SQL statement and compute a query strategy. That information is kept with the prepared statement and reused whenever the query is reissued.

You create a prepared statement with the prepareStatement method of the Connection class. Use a ? character for each parameter.

```
PreparedStatement stat = conn.prepareStatement(
   "SELECT password FROM Users WHERE username=?");
```

When you are ready to issue a prepared statement, first set the parameter values.

```
stat.setString(1, name);
```

(Note that the index value 1 denotes the first parameter.) Then issue the statement in the usual way:

```
ResultSet result = stat.executeQuery();
```

At first glance, it appears as if prepared statements would not be of much benefit in a web application. After all, you close the connection whenever you complete a user request. A prepared statement is tied to a database connection, and

all the work of establishing it is lost when the physical connection to the database is terminated.

However, if the physical database connections are kept in a pool, then there is a good chance that the prepared statement is still usable when you retrieve a connection. Many connection pool implementations will cache prepared statements. When you call `prepareStatement`, the pool will first look inside the statement cache, using the query string as a key. If the prepared statement is found, then it is reused. Otherwise, a new prepared statement is created and added to the cache.

All this activity is transparent to the application programmer. You simply request `PreparedStatement` objects and hope that, at least some of the time, the pool can retrieve an existing object for the given query.

You will see in the next section how to configure the connection pool in the Tomcat container to cache prepared statements.

CAUTION: You cannot keep a `PreparedStatement` object and reuse it beyond a single request scope. Once you close a pooled connection, all associated `PreparedStatement` objects also revert to the pool. Thus, you should not hang on to `PreparedStatement` objects beyond the current request. Instead, keep calling the `prepareStatement` method with the same query string, and chances are good that you'll get a cached statement object.

Configuring a Database Resource in Tomcat

In this section, we walk you through the steps of configuring a database resource pool in the Tomcat 5 container.

Locate the `conf/server.xml` file and look for the element that describes the host that will contain your web application, such as

```
<!-- Define the default virtual host -->
<Host name="localhost" debug="0" appBase="webapps"
   unpackWARs="false" autoDeploy="true">
...
</Host>
```

Inside this element, place a `DefaultContext` element that specifies both the database details (driver, URL, username, and password) and the desired characteristics of the pool.

Here is a typical example, specifying a connection pool to a PostgreSQL database. The values that you need to customize are highlighted.


```
<DefaultContext>
   <Resource name="jdbc/mydb" auth="Container"
      type="javax.sql.DataSource"/>
   <ResourceParams name="jdbc/mydb">
      <parameter>
         <name>factory</name>
          <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
      </parameter>
      <parameter>
         <name>driverClassName</name>
         <value>org.postgresql.Driver</value>
      </parameter>
      <parameter>
         <name>url</name>
         <value>jdbc:postgresql://127.0.0.1:5432/postgres</value>
      </parameter>
      <parameter>
        <name>username</name>
         <value>dbuser</value>
      </parameter>
      <parameter>
         <name>password</name>
         <value>dbpassword</value>
      </parameter>
      <parameter>
         <name>maxActive</name>
         <value>20</value>
      </parameter>
      <parameter>
         <name>maxIdle</name>
         <value>10</value>
      </parameter>
      <parameter>
         <name>poolPreparedStatements</name>
         <value>true</value>
      </parameter>
   </ResourceParams>
</DefaultContext>
```

NOTE: You can also add the `Resource` and `ResourceParams` elements into the context of a specific web application. Then the data source is available only to that application.

Note the name of the resource: `jdbc/mydb`. That name is used to obtain the data source from the JNDI directory service:

```
DataSource source = (DataSource) ctx.lookup("java:comp/env/jdbc/mydb");
```

The `java:comp/env` prefix is the standard JNDI directory lookup path to the component environment in a J2EE container. By convention, you place JDBC resources in the `jdbc` subpath. It is up to you how to name the individual resources.

To configure the pool, you specify a sequence of parameters—see Table 10–1 for the most common ones. A complete description of all valid parameters can be found at http://jakarta.apache.org/commons/dbcp/configuration.html.

Table 10–1 Common Tomcat Database Pool Parameters

Parameter Name	Description
`driverClassName`	The name of the JDBC driver, such as `org.postgresql.Driver`
`url`	The database URL, such as `jdbc:postgresql:mydb`
`username`	The database user name
`password`	The password of the database user
`maxActive`	The maximum number of simultaneous active connections, or zero for no limit.
`maxIdle`	The maximum number of active connections that can remain idle in the pool without extra ones being released, or zero for no limit.
`poolPreparedStatements`	`true` if prepared statements are pooled (default: `false`)
`removeAbandoned`	`true` if the pool should remove connections that appear to be abandoned (default: `false`)
`removeAbandonedTimeout`	The number of seconds after which an unused connection is considered abandoned (default: 300)
`logAbandoned`	`true` to log a stack trace of the code that abandoned the connection (default: `false`)

To activate the pooling of prepared statements, be sure to set `poolPreparedStatements` to `true`.

The last three parameters in Table 10–1 refer to a useful feature of the Tomcat pool. The pool can be instructed to monitor and remove connections that appear to be abandoned. If a connection has not been used for some time, then it is likely that an application forgot to close it. After all, a web application should always close its database connections after rendering the response to a user request. The pool can recycle unused connections and optionally log these

events. The logging is useful for debugging since it allows the application programmer to plug connection leaks.

The J2EE specification requires that resources are declared in the `web.xml` file of your web application. Add the following entry to your `web.xml` file:

```
<resource-ref>
   <res-ref-name>jdbc/mydb</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
</resource-ref>
```

Finally, you need to place the database driver file (such as `pg73jdbc3.jar` for the PostgreSQL database) into Tomcat's `common/lib` directory. If the database driver file has a `.zip` extension, you need to rename it to `.jar`, such as `classes12.jar` for the Oracle database.

TIP: You can find detailed configuration instructions for a number of popular databases at http://jakarta.apache.org/tomcat/tomcat-5.0-doc/jndi-data-source-examples-howto.html.

A Complete Database Example

In this example, we show how to verify a username/password combination. As with the example program in Chapter 1, we start with a simple login screen (Figure 10–1). If the username/password combination is correct, we show a welcome screen (Figure 10–2). Otherwise, we prompt the user to try again (Figure 10–3). Finally, if a JNDI or database error occurred, we show an error screen (Figure 10–4).

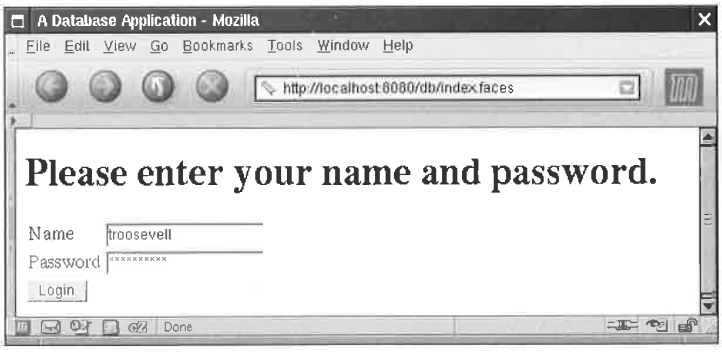


Figure 10–1 Login Screen

Figure 10–2 Welcome Screen

Figure 10–3 Authentication Error Screen

Figure 10–4 Internal Error Screen

Thus, we have four JSF pages, shown in Listings 10–1 through 10–4. Listing 10–5 shows the faces-config.xml file with the navigation rules. The navigation rules use the loginAction and logoutAction properties of the UserBean class. Listing 10–6 gives the code for the UserBean.

In our simple example, we add the database code directly into the UserBean class. It would also be possible to have two layers of objects: beans for communication with the JSF pages, and data access objects that represent entities in the database.

We place the code for database access into the separate method

```
public void doLogin() throws SQLException, NamingException
```

That method queries the database for the username/password combination and sets the loggedIn field to true if the username and password match.

The button on the index.jsp page references the login method of the user bean. That method calls the doLogin method and returns a result string for the navigation handler. The login method also deals with exceptions that the doLogin method reports. We assume that the doLogin method is focused on the database, not the user interface. If an exception occurs, doLogin should simply report it and take no further action. The login method, on the other hand, logs exceptions and returns a result string "internalError" to the navigation handler.

```
public String login() {
   try {
      doLogin();
   }
   catch (SQLException ex) {
      logger.log(Level.SEVERE, "loginAction", ex);
      return "internalError";
   }
   catch (NamingException ex) {
      logger.log(Level.SEVERE, "loginAction", ex);
      return "internalError";
   }
   if (loggedIn)
      return "loginSuccess";
   else
      return "loginFailure";
}
```

Before running this example, you need to carry out several housekeeping chores.

- Start your database.
- Create a table named Users and add one or more username/password entries:

```
CREATE TABLE Users (username CHAR(20), password CHAR(20))
INSERT INTO Users VALUES ('troosevelt', 'jabberwock')
```

- Place the database driver file into Tomcat's `common/lib` directory.
- Modify `conf/server.xml` and add the database resource.
- Restart Tomcat.

You can then deploy and test your application.

Figure 10–5 shows the directory structure for this application, and Figure 10–6 shows the navigation map.

NOTE: Lots of things can go wrong with database configurations. If the application has an internal error, look at the Tomcat logs (by default, in Tomcat's `logs/catalina.out`).

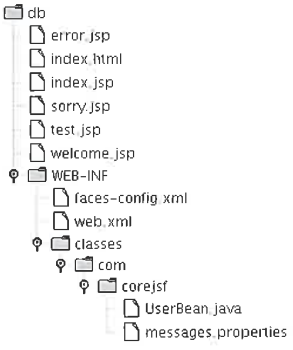


Figure 10–5 Directory Structure of the Database Application

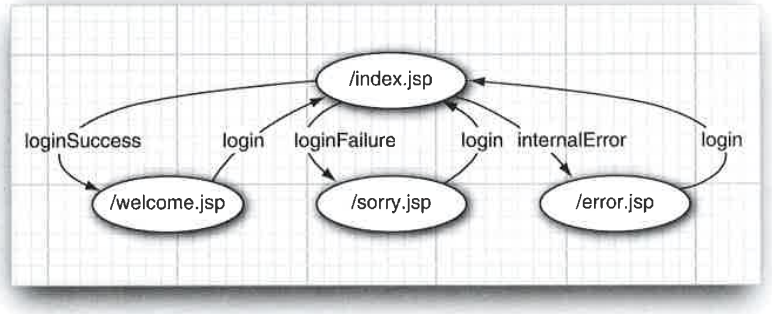


Figure 10–6 Navigation Map of the Database Application

Listing 10–1 db/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.enterNameAndPassword}"/></h1>
12.            <h:panelGrid columns="2">
13.               <h:outputText value="#{msgs.name}"/>
14.               <h:inputText value="#{user.name}"/>
15.
16.               <h:outputText value="#{msgs.password}"/>
17.               <h:inputSecret value="#{user.password}"/>
18.            </h:panelGrid>
19.            <h:commandButton value="#{msgs.login}" action="#{user.login}"/>
20.         </h:form>
21.      </body>
22.   </f:view>
23. </html>
```

Listing 10–2 db/welcome.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <p>
12.               <h:outputText value="#{msgs.welcome}"/>
13.               <h:outputText value="#{user.name}"/>!
14.            </p>
15.            <p>
16.               <h:commandButton value="#{msgs.logout}" action="#{user.logout}"/>
17.            </p>
18.         </h:form>
19.      </body>
20.   </f:view>
21. </html>
```


Listing 10–3 db/sorry.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.authError}"/></h1>
12.            <p>
13.               <h:outputText value="#{msgs.authError_detail}"/>!
14.            </p>
15.            <p>
16.               <h:commandButton value="#{msgs.continue}" action="login"/>
17.            </p>
18.         </h:form>
19.      </body>
20.   </f:view>
21. </html>
```

Listing 10–4 db/error.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.internalError}"/></h1>
12.            <p><h:outputText value="#{msgs.internalError_detail}"/></p>
13.            <p>
14.               <h:commandButton value="#{msgs.continue}" action="login"/>
15.            </p>
16.         </h:form>
17.      </body>
18.   </f:view>
19. </html>
```


Listing 10–5 db/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2. <!DOCTYPE faces-config PUBLIC
3.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
4.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
5. <faces-config>
6.    <navigation-rule>
7.       <from-view-id>/index.jsp</from-view-id>
8.       <navigation-case>
9.          <from-outcome>loginSuccess</from-outcome>
10.         <to-view-id>/welcome.jsp</to-view-id>
11.      </navigation-case>
12.      <navigation-case>
13.         <from-outcome>loginFailure</from-outcome>
14.         <to-view-id>/sorry.jsp</to-view-id>
15.      </navigation-case>
16.      <navigation-case>
17.         <from-outcome>internalError</from-outcome>
18.         <to-view-id>/error.jsp</to-view-id>
19.      </navigation-case>
20.   </navigation-rule>
21.   <navigation-rule>
22.      <from-view-id>/welcome.jsp</from-view-id>
23.      <navigation-case>
24.         <from-outcome>login</from-outcome>
25.         <to-view-id>/index.jsp</to-view-id>
26.      </navigation-case>
27.   </navigation-rule>
28.   <navigation-rule>
29.      <from-view-id>/sorry.jsp</from-view-id>
30.      <navigation-case>
31.         <from-outcome>login</from-outcome>
32.         <to-view-id>/index.jsp</to-view-id>
33.      </navigation-case>
34.   </navigation-rule>
35.   <navigation-rule>
36.      <from-view-id>/error.jsp</from-view-id>
37.      <navigation-case>
38.         <from-outcome>login</from-outcome>
39.         <to-view-id>/index.jsp</to-view-id>
40.      </navigation-case>
41.   </navigation-rule>
42.
43.   <managed-bean>
44.      <managed-bean-name>user</managed-bean-name>
45.      <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
46.      <managed-bean-scope>session</managed-bean-scope>
47.   </managed-bean>
48. </faces-config>
```


Listing 10–6 db/WEB-INF/classes/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.sql.Connection;
4. import java.sql.PreparedStatement;
5. import java.sql.ResultSet;
6. import java.sql.SQLException;
7. import java.util.logging.Level;
8. import java.util.logging.Logger;
9. import javax.naming.Context;
10. import javax.naming.InitialContext;
11. import javax.naming.NamingException;
12. import javax.sql.DataSource;
13.
14. public class UserBean {
15.    private String name;
16.    private String password;
17.    private boolean loggedIn;
18.    private Logger logger = Logger.getLogger("com.corejsf");
19.
20.    public String getName() { return name; }
21.    public void setName(String newValue) { name = newValue; }
22.
23.    public String getPassword() { return password; }
24.    public void setPassword(String newValue) { password = newValue; }
25.
26.    public String login() {
27.       try {
28.          doLogin();
29.       }
30.       catch (SQLException ex) {
31.          logger.log(Level.SEVERE, "loginAction", ex);
32.          return "internalError";
33.       }
34.       catch (NamingException ex) {
35.          logger.log(Level.SEVERE, "loginAction", ex);
36.          return "internalError";
37.       }
38.       if (loggedIn)
39.          return "loginSuccess";
40.       else
41.          return "loginFailure";
42.    }
43.
44.    public String logout() {
45.       loggedIn = false;
```


Listing 10–6 db/WEB-INF/classes/com/corejsf/UserBean.java (cont.)

```
46.         return "login";
47.      }
48.
49.      public void doLogin() throws SQLException, NamingException {
50.         Context ctx = new InitialContext();
51.         if (ctx == null) throw new NamingException("No initial context");
52.
53.         DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/mydb");
54.         if (ds == null) throw new NamingException("No data source");
55.
56.         Connection conn = ds.getConnection();
57.         if (conn == null) throw new SQLException("No connection");
58.
59.         try {
60.            PreparedStatement passwordQuery = conn.prepareStatement(
61.               "SELECT password from Users WHERE username = ?");
62.
63.            passwordQuery.setString(1, name);
64.
65.            ResultSet result = passwordQuery.executeQuery();
66.
67.            if (!result.next()) return;
68.            String storedPassword = result.getString("password");
69.            loggedIn = password.equals(storedPassword.trim());
70.         }
71.         finally {
72.            conn.close();
73.         }
74.      }
75. }
```

Using LDAP for Authentication

In the preceding section, you have seen how to read a username and password from a database. In this section, we look at LDAP, the Lightweight Directory Access Protocol. LDAP servers are more flexible and efficient for managing user information than are database servers. Particularly in large organizations, in which data replication is an issue, LDAP is preferred over relational databases for storing directory information.

Because LDAP is less commonly used than relational database technology, we briefly introduce it here. For an in-depth discussion of LDAP, we recommend the "LDAP bible": *Timothy Howes et al., Understanding and Deploying LDAP Directory Services, 2nd ed., Macmillan 2003.*

LDAP Directories

LDAP uses a *hierarchical* database. It keeps all data in a tree structure, not in a set of tables as a relational database would. Each entry in the tree has

- zero or more *attributes*. An attribute has a key and a value. An example attribute is `cn=John Q. Smith`. (The key `cn` stores the "common name." See Table 10–2 for the meaning of commonly used LDAP attributes.)
- one or more *object classes*. An object class defines the set of required and optional attributes for this element. For example, the object class `person` defines a required attribute `cn` and an optional attribute `telephoneNumber`. Of course, the object classes are different from Java classes, but they also support a notion of inheritance. For example, `inetOrgPerson` is a subclass of `person` with additional attributes.
- a *distinguished name* (for example, `uid=troosevelt,ou=people,dc=corejsf,dc=com`). The distinguished name is a sequence of attributes that trace a path joining the entry with the root of the tree. There may be alternate paths, but one of them must be specified as distinguished.

Table 10–2 Commonly Used LDAP Attributes

Attribute Name	Meaning
dc	Domain Component
cn	Common Name
sn	Surname
dn	Distinguished Name
o	Organization
ou	Organizational Unit
uid	Unique Identifier

Figure 10–7 shows an example of a directory tree.

How to organize the directory tree, and what information to put in it, can be a matter of intense debate. We do not discuss the issues here. Instead, we simply assume that an organizational scheme has been established and that the directory has been populated with the relevant user data.

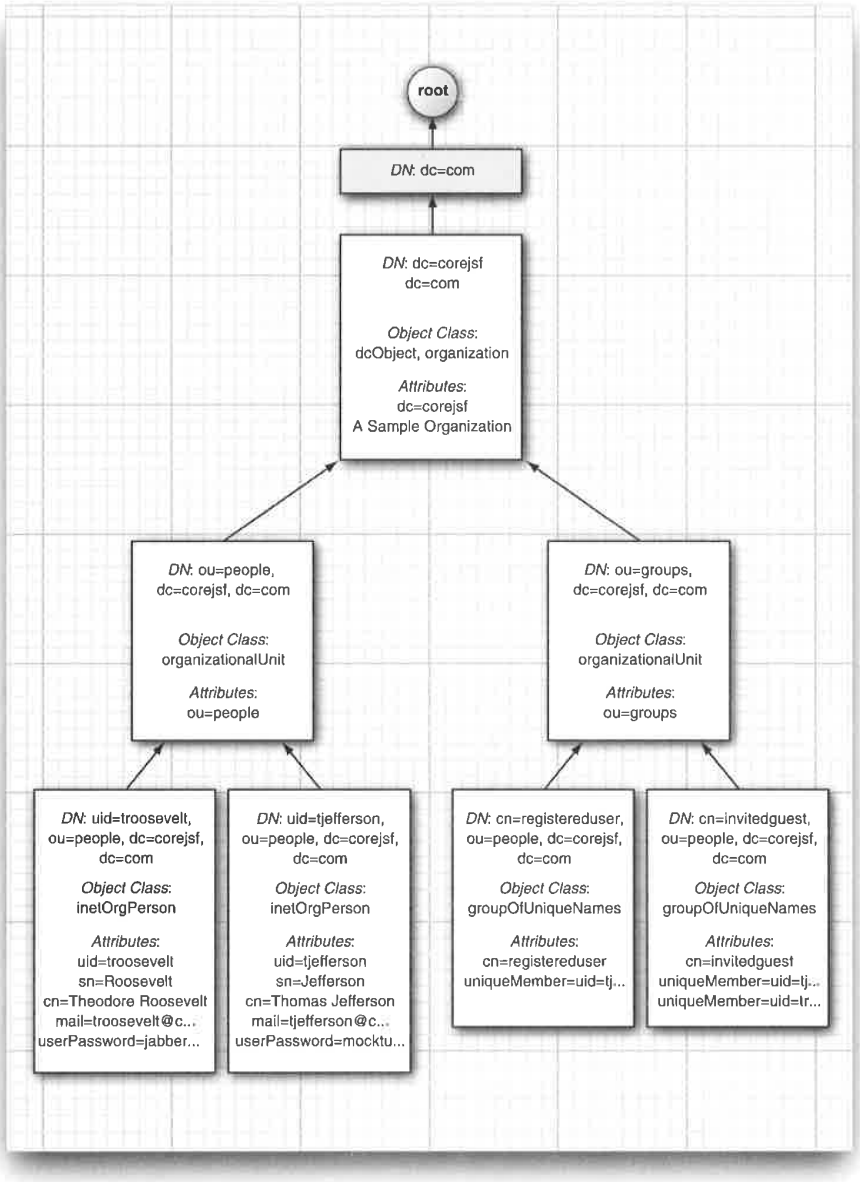


Figure 10–7 A directory tree

Configuring an LDAP Server

You have several options for running an LDAP server to try out the programs in this section. Here are the most popular choices:

- The free OpenLDAP server (http://openldap.org), available for Linux and Windows and built into Mac OS X.
- A high-performance server such as the Sun Java System Directory Server (http://www.sun.com/software/products/directory_srvr/home_directory.html), which is available on a variety of platforms
- Microsoft Active Directory

We give you brief instructions for configuring OpenLDAP. If you use another directory server, the basic steps are similar.

Our sample directory uses the standard object class `inetOrgPerson`. (We use that class because it has useful attributes such as `uid` and `mail`.) You should make sure that your LDAP server recognizes this object class.

If you use OpenLDAP, you need to edit the `slapd.conf` file before starting the LDAP server. Locate the line that includes the `core.schema` file, and add lines to include the `cosine.schema` and `inetorgperson.schema` files. (On Linux, the default location for the `slapd.conf` file is `/usr/local/etc/openldap`. The schema files are in the `schema` subdirectory.)

NOTE: Alternatively, you can make adjustments to our sample data. For example, you can change `inetOrgPerson` to the more commonly available `person`, omit the `uid` and `mail` attributes, and use the `sn` attribute as the login name. If you follow that approach, you will need to change the attributes in the sample programs as well.

In OpenLDAP, edit the `suffix` entry in `slapd.conf` to match the sample data set. This entry specifies the distinguished name suffix for this server. It should read

```
suffix  "dc=corejsf,dc=com"
```

You also need to configure an LDAP user with administrative rights to edit the directory data. In OpenLDAP, add these lines to `slapd.conf`:

```
rootdn  "cn=Manager,dc=corejsf,dc=com"
rootpw  secret
```


We recommend that you specify authorization settings, although they are not strictly necessary for running the examples in this sections. The following settings in slapd.conf permit the Manager user to read and write passwords, and everyone else to read all other attributes.

```
access to attr=userPassword
            by dn.base="cn=Manager,dc=corejsf,dc=com" write
            by * none
access to *
            by dn.base="cn=Admin,dc=corejsf,dc=com" write
            by * read
```

You can now start the LDAP server. On Linux, run /usr/local/libexec/slapd.

Next, populate the server with the sample data. Most LDAP servers allow the import of LDIF (Lightweight Directory Interchange Format) data. LDIF is a humanly readable format that simply lists all directory entries, including their distinguished names, object classes, and attributes. Listing 10–7 shows an LDIF file that describes our sample data:

```
. ldap/misc/sample.ldif
```

For example, with OpenLDAP, you use the ldapadd tool to add the data to the directory:

```
ldapadd -f sample.ldif -x -D "cn=Manager,dc=corejsf,dc=com" -w secret
```

Before proceeding, it is a good idea to double-check that the directory contains the data that you need. We suggest that you download Jarek Gawor's LDAP Browser\Editor from http://www.mcs.anl.gov/~gawor/ldap/. This convenient Java program lets you browse the contents of any LDAP server. Launch the program and configure it with the following options:

- Host: localhost
- Base DN: dc=corejsf,dc=com
- Anonymous bind: unchecked
- User DN: cn=Manager
- Append base DN: checked
- Password: secret

Make sure the LDAP server has started, then connect. If everything is in order, you should see a directory tree similar to that shown in Figure 10–8.

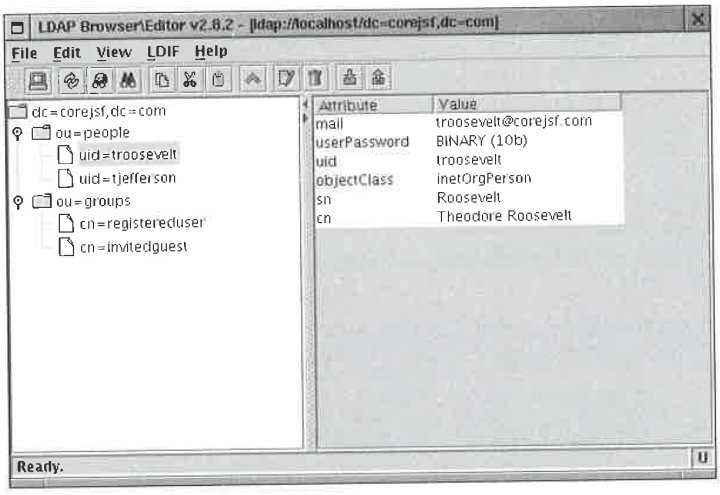


Figure 10–8 Inspecting an LDAP Directory Tree

Listing 10–7 ldap/misc/sample.ldif

```
# Define top-level entry
dn: dc=corejsf,dc=com
objectClass: dcObject
objectClass: organization
dc: corejsf
o: A Sample Organization

# Define an entry to contain people
# searches for users are based on this entry
dn: ou=people,dc=corejsf,dc=com
objectClass: organizationalUnit
ou: people

# Define a user entry for Theodore Roosevelt
dn: uid=troosevelt,ou=people,dc=corejsf,dc=com
objectClass: inetOrgPerson
uid: troosevelt
sn: Roosevelt
cn: Theodore Roosevelt
mail: troosevelt@corejsf.com
userPassword: jabberwock

```

Listing 10–7 ldap/misc/sample.ldif (cont.)

```
23. # Define a user entry for Thomas Jefferson
24. dn: uid=tjefferson,ou=people,dc=corejsf,dc=com
25. objectClass: inetOrgPerson
26. uid: tjefferson
27. sn: Jefferson
28. cn: Thomas Jefferson
29. mail: tjefferson@corejsf.com
30. userPassword: mockturtle
31.
32. # Define an entry to contain LDAP groups
33. # searches for roles are based on this entry
34. dn: ou=groups,dc=corejsf,dc=com
35. objectClass: organizationalUnit
36. ou: groups
37.
38. # Define an entry for the "registereduser" role
39. dn: cn=registereduser,ou=groups,dc=corejsf,dc=com
40. objectClass: groupOfUniqueNames
41. cn: registereduser
42. uniqueMember: uid=tjefferson,ou=people,dc=corejsf,dc=com
43.
44. # Define an entry for the "invitedguest" role
45. dn: cn=invitedguest,ou=groups,dc=corejsf,dc=com
46. objectClass: groupOfUniqueNames
47. cn: invitedguest
48. uniqueMember: uid=troosevelt,ou=people,dc=corejsf,dc=com
49. uniqueMember: uid=tjefferson,ou=people,dc=corejsf,dc=com
```

Accessing LDAP Directory Information

Once you have your LDAP database populated, it is time to connect to it with a Java program. You use the Java Naming and Directory Interface (JNDI), an interface that unifies various directory protocols.

Start by getting a *directory context* to the LDAP directory, with the following incantation:

```
Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, userDN);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext initial = new InitialDirContext(env);
DirContext context = (DirContext) initial.lookup("ldap://localhost:389");
```

Here, we connect to the LDAP server at the local host. The port number 389 is the default LDAP port.

If you connect to the LDAP database with an invalid user/password combination, an AuthenticationException is thrown.

NOTE: Sun's JNDI tutorial suggests an alternative way to connect to the server:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_PRINCIPAL, userDN);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext context = new InitialDirContext(env);
```

However, it seems undesirable to hardwire the Sun LDAP provider into your code. JNDI has an elaborate mechanism for configuring providers, and you should not lightly bypass it.

To list the attributes of a given entry, specify its distinguished name and then use the getAttributes method:

```
Attributes answer
   = context.getAttributes("uid=troosevelt,ou=people,dc=corejsf,dc=com");
```

You can get a specific attribute with the get method, for example:

```
Attribute commonNameAttribute = answer.get("cn");
```

To enumerate all attributes, you use the NamingEnumeration class. The designers of this class felt that they too could improve on the standard Java iteration protocol, and they gave us this usage pattern:

```
NamingEnumeration attrEnum = answer.getAll();
while (attrEnum.hasMore()) {
    Attribute attr = (Attribute) attrEnum.next();
    String id = attr.getID();
    ...
}
```

Note the use of hasMore instead of hasNext.

Since an attribute can have multiple values, you need to use another NamingEnumeration to list them all:

```
NamingEnumeration valueEnum = attr.getAll();
while (valueEnum.hasMore()) {
    Object value = valueEnum.next();
    ...
}
```

However, if you know that the attribute has a single value, you can call the `get` method to retrieve it:

```
String commonName = (String) commonNameAttribute.get();
```

You now know how to query the directory for user data. Next, let us take up operations for modifying the directory contents.

To add a new entry, gather the set of attributes in a `BasicAttributes` object. (The `BasicAttributes` class implements the `Attributes` interface.)

```
Attributes attrs = new BasicAttributes();
attrs.put("objectClass", "inetOrgPerson");
attrs.put("uid", "alincoln");
attrs.put("sn", "Lincoln");
attrs.put("cn", "Abraham Lincoln");
attrs.put("mail", "alincoln@corejsf.com");
String pw = "redqueen";
attrs.put("userPassword", pw.getBytes());
```

Then call the `createSubcontext` method. Provide the distinguished name of the new entry and the attribute set.

```
context.createSubcontext(
   "uid=alincoln,ou=people,dc=corejsf,dc=com", attrs);
```

CAUTION: When assembling the attributes, remember that the attributes are checked against the schema. Don't supply unknown attributes, and be sure to supply all attributes that are required by the object class. For example, if you omit the `sn` of `person`, the `createSubcontext` method will fail.

To remove an entry, call `destroySubcontext`:

```
context.destroySubcontext(
   "uid=alincoln,ou=people,dc=corejsf,dc=com");
```

Finally, you may want to edit the attributes of an existing entry. You call the method

```
context.modifyAttributes(distinguishedName, flag, attrs);
```

Here, `flag` is one of

```
DirContext.ADD_ATTRIBUTE
DirContext.REMOVE_ATTRIBUTE
DirContext.REPLACE_ATTRIBUTE
```


The attrs parameter contains a set of the attributes to be added, removed, or replaced.

Conveniently, the BasicAttributes(String, Object) constructor constructs an attribute set with a single attribute. For example,

```
context.modifyAttributes(
   "uid=alincoln,ou=people,dc=corejsf,dc=com",
   DirContext.ADD_ATTRIBUTE,
   new BasicAttributes("telephonenumber", "+18005551212"));

context.modifyAttributes(
   "uid=alincoln,ou=people,dc=corejsf,dc=com",
   DirContext.REMOVE_ATTRIBUTE,
   new BasicAttributes("mail", "alincoln@coresjf.com"));

context.modifyAttributes(
   "uid=alincoln,ou=people,dc=corejsf,dc=com",
   DirContext.REPLACE_ATTRIBUTE,
   new BasicAttributes("userPassword", newpw.getBytes()));
```

Finally, when you are done with a context, you should close it:

```
context.close();
```

You now know enough about directory operations to carry out the tasks that you will commonly need when working with LDAP directories. A good source for more advanced information is the JNDI tutorial at http://java.sun.com/products/jndi/tutorial.

However, we are not quite ready to put together a JSF application that uses LDAP. It would be extremely unprofessional to hardcode the directory URL and the manager password into a program. Instead, these values should be specified in a configuration file. The next section discusses various options for the management of configuration parameters. We put the alternatives to work with an application that allows users to self-register on a web site; we use LDAP to store the user information.

javax.naming.directory.InitialDirContext [SDK 1.3]

- InitialDirContext(Hashtable env)

  Constructs a directory context, using the given environment settings. The hash table can contain bindings for Context.SECURITY_PRINCIPAL, Context.SECURITY_CREDENTIALS, and other keys—see the API documentation for the javax.naming.Context interface for details.

javax.naming.Context [SDK 1.3]

- `Object lookup(String name)`
  Looks up the object with the given name. The return value depends on the nature of this context. It commonly is a subtree context or a leaf object.
- `Context createSubcontext(String name)`
  Creates a subcontext with the given name. The subcontext becomes a child of this context. All path components of the name, except for the last one, must exist.
- `void destroySubcontext(String name)`
  Destroys the subcontext with the given name. All path components of the name, except for the last one, must exist.
- `void close()`
  Closes this context.

javax.naming.directory.DirContext [SDK 1.3]

- `Attributes getAttributes(String name)`
  Gets the attributes of the entry with the given name.
- `void modifyAttributes(String name, int flag, Attributes modes)`
  Modifies the attributes of the entry with the given name. The value `flag` is one of `DirContext.ADD_ATTRIBUTE`, `DirContext.REMOVE_ATTRIBUTE`, or `DirContext.REPLACE_ATTRIBUTE`

javax.naming.directory.Attributes [SDK 1.3]

- `Attribute get(String id)`
  Gets the attribute with the given ID.
- `NamingEnumeration getAll()`
  Yields an enumeration that iterates through all attributes in this attribute set.
- `void put(String id, Object value)`
  Adds an attribute to this attribute set.

javax.naming.directory.BasicAttributes [SDK 1.3]

- `BasicAttributes(String id, Object value)`
  Constructs an attribute set that contains a single attribute with the given ID and value.

`javax.naming.directory.Attribute` [SDK 1.3]

- `String getID()`
  Gets the ID of this attribute.
- `Object get()`
  Gets the first attribute value of this attribute if the values are ordered or an arbitrary value if they are unordered.
- `NamingEnumeration getAll()`
  Yields an enumeration that iterates through all values of this attribute.

`javax.naming.NamingEnumeration` [SDK 1.3]

- `boolean hasMore()`
  Returns `true` if this enumeration object has more elements.
- `Object next()`
  Returns the next element of this enumeration.

Managing Configuration Information

Whenever your application interfaces with external services, you need to specify configuration parameters: URLs, usernames, passwords, and so on. You should never hardcode these parameters inside your application classes—doing so would make it difficult to update passwords, switch to alternative servers, and so on.

In the section on database services, you saw a reasonable approach for managing the database configuration. The configuration information is placed inside `server.xml`. The servlet container uses this information to construct a data source and bind it to a well-known name. The classes that need to access the database use JNDI look up the data source.

Placing configuration information into `server.xml` is appropriate for a *global* resource such as a database. This resource can be used by all web applications inside the container. On the other hand, application-specific configuration information should be placed inside `web.xml` or `faces-config.xml`. Using the example of an LDAP connection, we explore all three possibilities.

Configuring a Bean

Whenever you define a bean in faces-config.xml, you can provide initialization parameters by using the managed-property element. Here is how we can initialize a bean that connects to an LDAP directory:

```
<managed-bean>
   <managed-bean-name>userdir</managed-bean-name>
   <managed-bean-class>com.corejsf.UserDirectoryBean</managed-bean-class>
   <managed-bean-scope>application</managed-bean-scope>
   <managed-property>
      <property-name>URL</property-name>
      <value>ldap://localhost:389</value>
   </managed-property>
   <managed-property>
      <property-name>managerDN</property-name>
      <value>cn=Manager,dc=corejsf,dc=com</value>
   </managed-property>
   <managed-property>
      <property-name>managerPassword</property-name>
      <value>secret</value>
   </managed-property>
</managed-bean>
```

You see the familiar managed-bean-name and managed-bean-class elements. However, this bean is given *application scope*. The bean object stays alive for the duration of the entire application, and it can serve multiple sessions. Finally, we used the managed-property settings to initialize the bean. Thus, we achieved our goal of placing these initialization parameters inside a configuration file rather than hardwiring them into the bean code.

Of course, our bean needs setters for these properties:

```
public class UserDirectoryBean {
   private String url;
   private String managerDN;
   private String managerPW;

   public void setManagerDN(String newValue) { managerDN = newValue; }
   public void setManagerPassword(String newValue) { managerPW = newValue; }
   public void setURL(String newValue) { url = newValue; }

   public DirContext getRootContext() throws NamingException { ... }
}
```

When the bean is constructed, the setters are invoked with the values specified in faces-config.xml.

Finally, client code needs to have access to the bean object. For example, suppose the UserBean class wants to connect to the directory:

```
UserDirectoryBean userdir = ... // how?
DirContext context = userdir.connect(dn, pw);
```

To look up a JSF bean, you use its value binding of its name, as in the following statements:

```
FacesContext context = FacesContext.getCurrentInstance();
Application app = context.getApplication();
ValueBinding binding = app.createValueBinding("#{userdir}");
UserDirectoryBean dir = (UserDirectoryBean) binding.getValue(context);
```

In summary, here are the steps for configuring a JSF bean:

1. Place the configuration parameters inside managed-property elements in the faces-config.xml file.
2. Provide property setters for these properties in the bean class.
3. Look up the bean object through its value binding.

This configuration method is straightforward and convenient. However, it is not suitable for configuring objects that should be available to multiple web applications. Moreover, purists might argue that faces-config.xml is intended to describe the logic of a web application, not its interface with external resources, and that web.xml would be more appropriate for the latter. Read on if either of these objections matters to you.

Configuring the External Context

In this section, we assume that your JSF application is launched as a servlet. You can supply parameters in web.xml by providing a set of context-param elements inside the web-app element:

```
<web-app>
   <context-param>
      <param-name>URL</param-name>
      <param-value>ldap://localhost:389</param-value>
   </context-param>
   <context-param>
      <param-name>managerDN</param-name>
      <param-value>cn=Manager,dc=corejsf,dc=com</param-value>
   </context-param>
```



```
    <context-param>
       <param-name>managerPassword</param-name>
       <param-value>secret</param-value>
    </context-param>
    ...
</web-app>
```

To read a parameter, get the *external context* object. That object describes the execution environment that launched your JSF application. If you use a servlet container, then the external context is a wrapper around the `ServletContext` object. The `ExternalContext` class has a number of convenience methods to access properties of the underlying servlet context. The `getInitParameter` method retrieves a context parameter value with a given name.

CAUTION: Do not confuse `context-param` with `init-param`. The latter tag is used for parameters that a servlet can process at startup. It is unfortunate that the method for reading a context parameter is called `getInitParameter`.

Here is the code for getting an LDAP context from configuration parameters in `web.xml`:

```
public DirContext getRootContext() throws NamingException {
   ExternalContext external
      = FacesContext.getCurrentInstance().getExternalContext();

   String managerDN = external.getInitParameter("managerDN");
   String managerPW = external.getInitParameter("managerPassword");
   String url = external.getInitParameter("URL");

   Hashtable env = new Hashtable();
   env.put(Context.SECURITY_PRINCIPAL, managerDN);
   env.put(Context.SECURITY_CREDENTIALS, managerPW);
   DirContext initial = new InitialDirContext(env);

   Object obj = initial.lookup(url);
   if (!(obj instanceof DirContext))
      throw new NamingException("No directory context");
   return (DirContext) obj;
}
```

Follow these steps for accessing resources through the external context:

1. Place the configuration parameters inside `context-param` elements in the `web.xml` file.
2. Use the `ExternalContext` to look up the parameter values.
3. Turn the parameters into objects for your application.

As you can see, this configuration method works at a lower level than the configuration of a JSF bean. The `web.xml` file simply contains an unstructured list of parameters. It is up to you to construct objects that make use of these parameters.

javax.faces.context.FacesContext

- `ExternalContext getExternalContext()`
  Gets the external context, a wrapper such as a servlet or portlet context around the execution environment of this JSF application.

javax.faces.context.ExternalContext

- `String getInitParameter(String name)`
  Gets the initialization parameter with the given name.

Configuring a Container-Managed Resource

We now discuss how to specify container-wide resources. The information in this section is specific to Tomcat. Other containers will have similar mechanisms, but the details will differ.

Earlier in this chapter, we showed you how to configure a JDBC data source by specifying the database URL and login parameters in Tomcat's `server.xml` file. We simply used a JNDI lookup to obtain the data source object. This is an attractive method for specifying systemwide resources. Fortunately, Tomcat lets you fit your own resources into the same mechanism.

As with JDBC data sources, you specify a `Resource` and its `ResourceParams` in `server.xml`. For example, here is the configuration information for an LDAP directory.

```
<Resource name="ldap/mydir" auth="Container"
   type="javax.naming.directory.DirContext"/>

<ResourceParams name="ldap/mydir">
   <parameter>
      <name>factory</name>
      <value>com.corejsf.DirContextFactory</value>
   </parameter>
   <parameter>
      <name>URL</name>
      <value>ldap://localhost:389</value>
   </parameter>
   <parameter>
      <name>java.naming.security.principal</name>
```



```
        <value>cn=Manager,dc=corejsf,dc=com</value>
    </parameter>
    <parameter>
        <name>java.naming.security.credentials</name>
        <value>secret</value>
    </parameter>
</ResourceParams>
```

However, Tomcat has no standard "factory" for LDAP directories. This class uses the custom factory `com.corejsf.DirContextFactory`. All factories need to implement the `ObjectFactory` interface type and implement the `getObjectInstance` method.

```
public class DirContextFactory implements ObjectFactory {

    public Object getObjectInstance(Object obj,
        Name n, Context nameCtx, Hashtable environment)
        throws NamingException {
        ...
    }
}
```

This method, defined in glorious generality, can be used to produce any object from arbitrary configuration information. There is quite a bit of variability in how the parameters are used, but fortunately we only need to understand what parameters Tomcat supplies when requesting a resource. Tomcat places the configuration parameters into a `Reference` object, a kind of hash table on a megadose of steroids. Our factory simply places the parameters into a plain hash table and then gets the directory context—see Listing 10–8 for the complete source code.

NOTE: The class `com.sun.jndi.ldap.LdapCtxFactory` (which is explicitly invoked in Sun's JNDI tutorial) also implements the `ObjectFactory` interface. Could you use that class as a factory for LDAP connections in Tomcat's `server.xml` file? Sadly, the answer is no. The `getObjectInstance` method of `com.sun.jndi.ldap.LdapCtxFactory` expects an `Object` parameter that is either an URL string, an array of URL strings, or a `Reference` object containing values with key `"URL"`. The other environment settings must be provided in the `Hashtable` parameter. That's not what Tomcat supplies.

Note that we simply use the standard JNDI environment names for the principal and credentials in the `server.xml` file. The `Context` interface constants that we used previously are merely shortcuts for the environment names. For example, `Context.SECURITY_PRINCIPAL` is the string `"java.naming.security.principal"`. (Admittedly,

the constant names aren't much shorter, but they are safer. If you misspell the constant, then the compiler will warn you. If you misspell the environment name, your application will mysteriously fail.)

Now that you have completed the configuration, the remainder is smooth sailing. Your program simply accesses the resource through its JNDI name:

```
public DirContext getRootContext() throws NamingException {
   Context ctx = new InitialContext();
   return (DirContext) ctx.lookup("java:comp/env/ldap/mydir");
}
```

In summary, here are the steps for configuring a container-wide resource:

1. Place the configuration parameters inside the `ResourceParams` section that has the same name as the `Resource` element for your resource.
2. If you use a custom resource factory, deploy the class so that the container can load it (for example, in a JAR file that you place inside the `common/lib` directory).
3. Look up the resource object through its JNDI name.

Listing 10–8 ldap3/misc/com/corejsf/DirContextFactory.java

```
1. package com.corejsf;
2.
3. import java.util.Enumeration;
4. import java.util.Hashtable;
5. import javax.naming.Context;
6. import javax.naming.Name;
7. import javax.naming.NamingException;
8. import javax.naming.RefAddr;
9. import javax.naming.Reference;
10. import javax.naming.directory.DirContext;
11. import javax.naming.directory.InitialDirContext;
12. import javax.naming.spi.ObjectFactory;
13.
14. public class DirContextFactory implements ObjectFactory {
15.    public Object getObjectInstance(Object obj,
16.       Name n, Context nameCtx, Hashtable environment)
17.       throws NamingException {
18.
19.       Hashtable env = new Hashtable();
20.       String url = null;
21.       Reference ref = (Reference) obj;
22.       Enumeration addrs = ref.getAll();
23.       while (addrs.hasMoreElements()) {
```


Listing 10–8 ldap3/misc/com/corejsf/DirContextFactory.java (cont.)

```
            RefAddr addr = (RefAddr) addrs.nextElement();
            String name = addr.getType();
            String value = (String) addr.getContent();
            if (name.equals("URL")) url = value;
            else env.put(name, value);
        }
        DirContext initial = new InitialDirContext(env);
        if (url == null) return initial;
        else return initial.lookup(url);
    }
}
```

NOTE: Compile this file, place it inside a JAR file, and put the JAR file into the common/lib directory of Tomcat.

```
cd corejsf-examples/ch10/ldap3/misc
javac com/corejsf/DirContextFactory.java
jar cvf tomcat/common/lib/dirctxfactory.jar com/corejsf/*.class
```

Remember to restart the server.

Creating an LDAP Application

We now put together a complete application that stores user information in an LDAP directory.

The application simulates a news web site that gives users free access to news as long as they provide some information about themselves. We do not actually provide any news. We simply provide a screen to log in (Figure 10–9) and a separate screen to register for the service (Figure 10–10). Upon successful login, users can read news and update their personal information (Figure 10–11).

The update screen is similar to the registration screen, and we do not show it. Figure 10–12 shows the directory structure, and Figure 10–13 shows the page flow between the news service pages.

We provide three versions of this application, with configuration information in faces-config.xml, web.xml, and server.xml, respectively.

All three versions have identical web pages—see Listings 10–9 through 10–12. (We omit the listings of repetitive pages.) The primary difference between the versions is the implementation of the getRootContext method in the UserBean class (Listing 10–13). The first application has a UserDirectoryBean class (Listing 10–14) that is configured in faces-config.xml (Listing 10–15). The second application

makes an ad hoc lookup of servlet initialization parameters. The third version makes a JNDI lookup, using the class of Listing 10–8. See the preceding sections for details. Finally, for completeness, Listing 10–16 contains the code for the `Name` class that is used in the `UserBean` class.

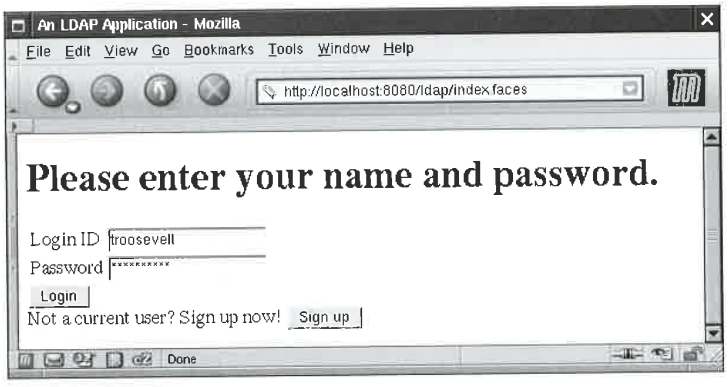


Figure 10–9 Logging In to the News Service

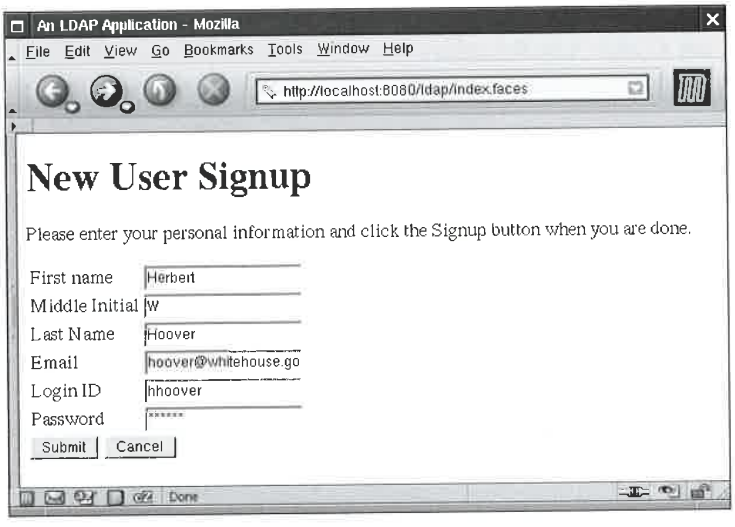


Figure 10–10 Registering for the News Service

Figure 10–11 Main Screen of the News Service

Figure 10–12 The Directory Structure of the LDAP Example

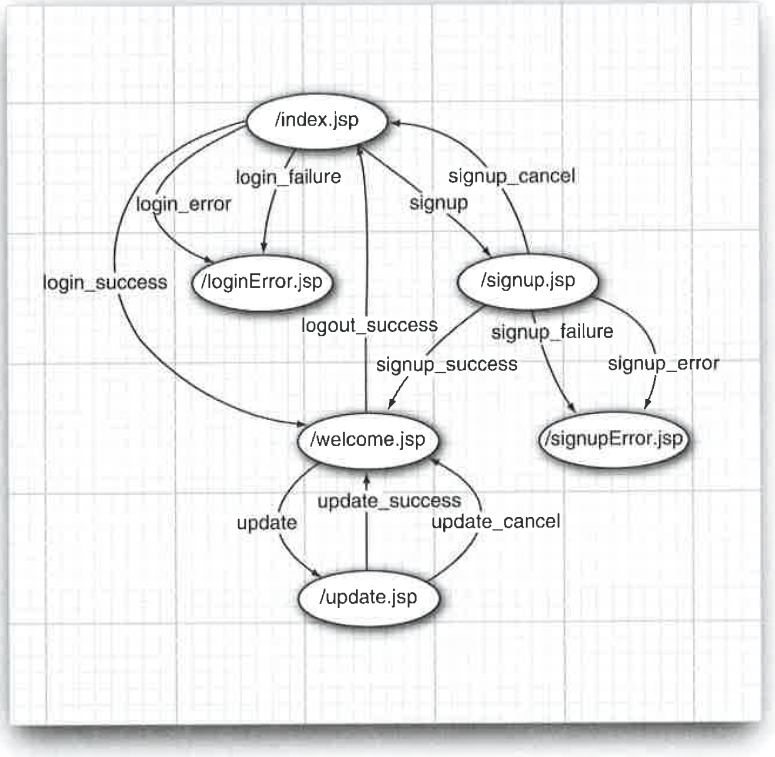


Figure 10–13 Page Flow of the News Service

Listing 10–9 ldap/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.enterNameAndPassword}"/></h1>
12.            <h:panelGrid columns="2">
```

Listing 10-9 ldap/index.jsp (cont.)

```
13.                <h:outputText value="#{msgs.loginID}"/>
14.                <h:inputText value="#{user.id}"/>
15.
16.                <h:outputText value="#{msgs.password}"/>
17.                <h:inputSecret value="#{user.password}"/>
18.             </h:panelGrid>
19.             <h:commandButton value="#{msgs.login}" action="#{user.login}"/>
20.             <br/>
21.             <h:outputText value="#{msgs.signupNow}"/>
22.             <h:commandButton value="#{msgs.signup}" action="signup"/>
23.          </h:form>
24.       </body>
25.    </f:view>
26. </html>
```

Listing 10-10 ldap/signup.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.newUserSignup}"/></h1>
12.            <p><h:outputText value="#{msgs.newUserSignup_detail}"/></p>
13.            <h:panelGrid columns="2">
14.               <h:outputText value="#{msgs.firstName}"/>
15.               <h:inputText value="#{user.name.first}"/>
16.
17.               <h:outputText value="#{msgs.middleInitial}"/>
18.               <h:inputText value="#{user.name.middle}"/>
19.
20.               <h:outputText value="#{msgs.lastName}"/>
21.               <h:inputText value="#{user.name.last}"/>
22.
23.               <h:outputText value="#{msgs.email}"/>
24.               <h:inputText value="#{user.email}"/>
25.
```


Listing 10-10 ldap/signup.jsp (cont.)

```
26.                <h:outputText value="#{msgs.loginID}"/>
27.                <h:inputText value="#{user.id}"/>
28.
29.                <h:outputText value="#{msgs.password}"/>
30.                <h:inputSecret value="#{user.password}"/>
31.             </h:panelGrid>
32.             <h:commandButton value="#{msgs.submit}" action="#{user.signup}"/>
33.             <h:commandButton value="#{msgs.cancel}" action="signup_cancel"/>
34.          </h:form>
35.       </body>
36.    </f:view>
37. </html>
```

Listing 10-11 ldap/welcome.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.success}"/></h1>
12.            <p>
13.               <h:outputText value="#{msgs.welcome}"/>
14.               <h:outputText value="#{user.name}"/>!
15.            </p>
16.            <p>
17.               <h:outputText value="#{msgs.success_detail}"/>
18.            </p>
19.            <h:commandButton value="#{msgs.update}" action="update"/>
20.            <h:commandButton value="#{msgs.logout}" action="#{user.logout}"/>
21.         </h:form>
22.      </body>
23.   </f:view>
24. </html>
```


Listing 10–12 ldap/loginError.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.loginError}"/></h1>
12.            <p>
13.               <h:outputText value="#{msgs.loginError_detail}"/>
14.            </p>
15.            <p>
16.               <h:commandButton value="#{msgs.tryAgain}" action="login"/>
17.               <h:commandButton value="#{msgs.signup}" action="signup"/>
18.            </p>
19.         </h:form>
20.      </body>
21.   </f:view>
22. </html>
```

Listing 10–13 ldap/WEB-INF/classes/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.util.logging.Level;
4. import java.util.logging.Logger;
5. import javax.faces.application.Application;
6. import javax.faces.context.FacesContext;
7. import javax.faces.el.ValueBinding;
8. import javax.naming.NameNotFoundException;
9. import javax.naming.NamingException;
10. import javax.naming.directory.Attributes;
11. import javax.naming.directory.BasicAttributes;
12. import javax.naming.directory.DirContext;
13.
14. public class UserBean {
15.    private Name name;
16.    private String id;
17.    private String email;
18.    private String password;
```


Listing 10–13 ldap/WEB-INF/classes/com/corejsf/UserBean.java (cont.)

```
19.   private Logger logger = Logger.getLogger("com.corejava");
20.
21.   public UserBean() { name = new Name(); }
22.
23.   public DirContext getRootContext() throws NamingException {
24.      FacesContext context = FacesContext.getCurrentInstance();
25.      Application app = context.getApplication();
26.      ValueBinding binding = app.createValueBinding("#{userdir}");
27.      UserDirectoryBean dir =
28.         (UserDirectoryBean) binding.getValue(context);
29.      return dir.getRootContext();
30.   }
31.
32.   public Name getName() { return name; }
33.   public void setName(Name newValue) { name = newValue; }
34.
35.   public String getEmail() { return email; }
36.   public void setEmail(String newValue) { email = newValue; }
37.
38.   public String getId() { return id; }
39.   public void setId(String newValue) { id = newValue; }
40.
41.   public String getPassword() { return password; }
42.   public void setPassword(String newValue) { password = newValue; }
43.
44.   public String login() {
45.      try {
46.         DirContext context = getRootContext();
47.         try {
48.            String dn = "uid=" + id + ",ou=people,dc=corejsf,dc=com";
49.            Attributes userAttributes = context.getAttributes(dn);
50.            String cn = (String) userAttributes.get("cn").get();
51.            name.parse(cn);
52.            email = (String) userAttributes.get("mail").get();
53.            byte[] pw = (byte[])
54.               userAttributes.get("userPassword").get();
55.            if (password.equals(new String(pw)))
56.               return "login_success";
57.            else
58.               return "login_failure";
59.         } finally {
60.            context.close();
61.         }
62.      }
63.      catch (NamingException ex) {
```


Listing 10–13 ldap/WEB-INF/classes/com/corejsf/UserBean.java (cont.)

```
64.         logger.log(Level.SEVERE, "loginAction", ex);
65.         return "login_error";
66.      }
67.   }
68.
69.   public String signup() {
70.      try {
71.         DirContext context = getRootContext();
72.         try {
73.            String dn = "uid=" + id + ",ou=people,dc=corejsf,dc=com";
74.
75.            try {
76.               context.lookup(dn);
77.               return "signup_failure";
78.            }
79.            catch (NameNotFoundException ex) {}
80.
81.            Attributes attrs = new BasicAttributes();
82.            attrs.put("objectClass", "inetOrgPerson");
83.            attrs.put("uid", id);
84.            attrs.put("sn", name.getLast());
85.            attrs.put("cn", name.toString());
86.            attrs.put("mail", email);
87.            attrs.put("userPassword", password.getBytes());
88.            context.createSubcontext(dn, attrs);
89.         } finally {
90.            context.close();
91.         }
92.      }
93.      catch (NamingException ex) {
94.         logger.log(Level.SEVERE, "loginAction", ex);
95.         return "signup_error";
96.      }
97.
98.      return "signup_success";
99.   }
100.
101.  public String update() {
102.     try {
103.        DirContext context = getRootContext();
104.        try {
105.           String dn = "uid=" + id + ",ou=people,dc=corejsf,dc=com";
106.           Attributes attrs = new BasicAttributes();
107.           attrs.put("sn", name.getLast());
108.           attrs.put("cn", name.toString());
```

Listing 10–13 ldap/WEB-INF/classes/com/corejsf/UserBean.java (cont.)

```
109.            attrs.put("mail", email);
110.            attrs.put("userPassword", password.getBytes());
111.            context.modifyAttributes(dn,
112.               DirContext.REPLACE_ATTRIBUTE, attrs);
113.         } finally {
114.            context.close();
115.         }
116.      }
117.      catch (NamingException ex) {
118.         logger.log(Level.SEVERE, "updateAction", ex);
119.         return "internal_error";
120.      }
121.
122.      return "update_success";
123.   }
124.
125.   public String logout() {
126.      password = "";
127.      return "logout_success";
128.   }
129. }
```

Listing 10–14 ldap/WEB-INF/classes/com/corejsf/UserDirectoryBean.java

```
1. package com.corejsf;
2.
3. import java.util.Hashtable;
4. import javax.naming.Context;
5. import javax.naming.NamingException;
6. import javax.naming.directory.DirContext;
7. import javax.naming.directory.InitialDirContext;
8.
9. public class UserDirectoryBean {
10.    private String url;
11.    private String managerDN;
12.    private String managerPW;
13.
14.    public void setManagerDN(String newValue) { managerDN = newValue; }
15.    public void setManagerPassword(String newValue) {
16.       managerPW = newValue; }
17.    public void setURL(String newValue) { url = newValue; }
18.
```


Listing 10–14 ldap/WEB-INF/classes/com/corejsf/UserDirectoryBean.java (cont.)

```
19.    public DirContext getRootContext() throws NamingException {
20.       Hashtable env = new Hashtable();
21.       env.put(Context.SECURITY_PRINCIPAL, managerDN);
22.       env.put(Context.SECURITY_CREDENTIALS, managerPW);
23.       DirContext initial = new InitialDirContext(env);
24.
25.       Object obj = initial.lookup(url);
26.       if (!(obj instanceof DirContext))
27.          throw new NamingException("No directory context");
28.       return (DirContext) obj;
29.    }
30. }
```

Listing 10–15 ldap/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2.
3. <!DOCTYPE faces-config PUBLIC
4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
6.
7. <faces-config>
8.
9.    <navigation-rule>
10.      <from-view-id>/index.jsp</from-view-id>
11.      <navigation-case>
12.         <from-outcome>login_success</from-outcome>
13.         <to-view-id>/welcome.jsp</to-view-id>
14.      </navigation-case>
15.      <navigation-case>
16.         <from-outcome>login_error</from-outcome>
17.         <to-view-id>/loginError.jsp</to-view-id>
18.      </navigation-case>
19.      <navigation-case>
20.         <from-outcome>login_failure</from-outcome>
21.         <to-view-id>/loginError.jsp</to-view-id>
22.      </navigation-case>
23.      <navigation-case>
24.         <from-outcome>signup</from-outcome>
25.         <to-view-id>/signup.jsp</to-view-id>
26.      </navigation-case>
27.   </navigation-rule>
28.   <navigation-rule>
29.      <from-view-id>/signup.jsp</from-view-id>
```


Listing 10-15 ldap/WEB-INF/faces-config.xml (cont.)

```
30.     <navigation-case>
31.        <from-outcome>signup_success</from-outcome>
32.        <to-view-id>/welcome.jsp</to-view-id>
33.     </navigation-case>
34.     <navigation-case>
35.        <from-outcome>signup_failure</from-outcome>
36.        <to-view-id>/signupError.jsp</to-view-id>
37.     </navigation-case>
38.     <navigation-case>
39.        <from-outcome>signup_error</from-outcome>
40.        <to-view-id>/signupError.jsp</to-view-id>
41.     </navigation-case>
42.     <navigation-case>
43.        <from-outcome>signup_cancel</from-outcome>
44.        <to-view-id>/index.jsp</to-view-id>
45.     </navigation-case>
46.  </navigation-rule>
47.  <navigation-rule>
48.     <from-view-id>/welcome.jsp</from-view-id>
49.     <navigation-case>
50.        <from-outcome>update</from-outcome>
51.        <to-view-id>/update.jsp</to-view-id>
52.     </navigation-case>
53.     <navigation-case>
54.        <from-outcome>logout_success</from-outcome>
55.        <to-view-id>/index.jsp</to-view-id>
56.     </navigation-case>
57.  </navigation-rule>
58.  <navigation-rule>
59.     <from-view-id>/update.jsp</from-view-id>
60.     <navigation-case>
61.        <from-outcome>update_success</from-outcome>
62.        <to-view-id>/welcome.jsp</to-view-id>
63.     </navigation-case>
64.     <navigation-case>
65.        <from-outcome>update_cancel</from-outcome>
66.        <to-view-id>/welcome.jsp</to-view-id>
67.     </navigation-case>
68.  </navigation-rule>
69.  <navigation-rule>
70.     <navigation-case>
71.        <from-outcome>login</from-outcome>
72.        <to-view-id>/index.jsp</to-view-id>
73.     </navigation-case>
74.     <navigation-case>
```


Listing 10–15 ldap/WEB-INF/faces-config.xml (cont.)

```
        <from-outcome>internal_error</from-outcome>
        <to-view-id>/internalError.jsp</to-view-id>
    </navigation-case>
</navigation-rule>

<managed-bean>
    <managed-bean-name>user</managed-bean-name>
    <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>userdir</managed-bean-name>
    <managed-bean-class>com.corejsf.UserDirectoryBean</managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
    <managed-property>
        <property-name>URL</property-name>
        <value>ldap://localhost:389</value>
    </managed-property>
    <managed-property>
        <property-name>managerDN</property-name>
        <value>cn=Manager,dc=corejsf,dc=com</value>
    </managed-property>
    <managed-property>
        <property-name>managerPassword</property-name>
        <value>secret</value>
    </managed-property>
</managed-bean>

</faces-config>
```

Listing 10–16 ldap/WEB-INF/classes/com/corejsf/Name.java

```
package com.corejsf;

public class Name {
    private String first;
    private String middle;
    private String last;

    public Name() { first = ""; middle = ""; last = ""; }

    public String getFirst() { return first; }
    public void setFirst(String newValue) { first = newValue; }
```

Listing 10–16 ldap/WEB-INF/classes/com/corejsf/Name.java (cont.)

```
12.
13.    public String getMiddle() { return middle; }
14.    public void setMiddle(String newValue) { middle = newValue; }
15.
16.    public String getLast() { return last; }
17.    public void setLast(String newValue) { last = newValue; }
18.
19.    public void parse(String fullName) {
20.       int firstSpace = fullName.indexOf(' ');
21.       int lastSpace = fullName.lastIndexOf(' ');
22.       if (firstSpace == -1) {
23.          first = "";
24.          middle = "";
25.          last = fullName;
26.       }
27.       else {
28.          first = fullName.substring(0, firstSpace);
29.          if (firstSpace < lastSpace)
30.             middle = fullName.substring(firstSpace + 1, lastSpace);
31.          else
32.             middle = "";
33.          last = fullName.substring(lastSpace + 1, fullName.length());
34.       }
35.    }
36.
37.    public String toString() {
38.       StringBuffer buffer = new StringBuffer();
39.       buffer.append(first);
40.       buffer.append(' ');
41.       if (middle.length() > 0) {
42.          buffer.append(middle.charAt(0));
43.          buffer.append(". ");
44.       }
45.       buffer.append(last);
46.       return buffer.toString();
47.    }
48. }
```

Container-Managed Authentication and Authorization

In the preceding sections you saw how a web application can use an LDAP directory to look up user information. It is up to the application to use that information appropriately, to allow or deny users access to certain resources. In this section, we discuss an alternative approach: *container-managed authentica-*

tion. This mechanism puts the burden of authenticating users on the servlet container (such as Tomcat). It is much easier to ensure that security is handled consistently for an entire Web application if the container manages autentication and authorization. The application programmer can then focus on the flow of the web application without worrying about user privileges.

Most of the configuration details in this chapter are specific to Tomcat, but other servlet containers have similar mechanisms.

To protect a set of pages, you specify access control information in the `web.xml` file. For example, the following security constraint restricts all pages in the `protected` subdirectory to authenticated users that have the role `registereduser` or `invitedguest`.

```
<security-constraint>
   <web-resource-collection>
      <url-pattern>/protected/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>registereduser</role-name>
      <role-name>invitedguest</role-name>
    </auth-constraint>
</security-constraint>
```

The role of a user is assigned during authentication. Roles are stored in the user directory together with user names and passwords.

NOTE: If JSF is configured to use a `/faces` prefix for JSF pages, then you must add a corresponding URL pattern to the security constraint, such as `/faces/protected/*` in the preceding example.

Next, you need to specify how users authenticate themselves. The most flexible approach is form-based authentication. Add the following entry to `web.xml`:

```
<login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
      <form-login-page>/login.html</form-login-page>
      <form-error-page>/noauth.html</form-error-page>
   </form-login-config>
</login-config>
```

The form login configuration specifies a web page into which the user types in the username and password. You are free to design any desired appearance for the login page, but you must include a mechanism to submit a request to

j_security_check with request parameters named j_username and j_password. The following form will do the job:

```
<form method="POST" action="j_security_check">
   User name: <input type="text" name="j_username"/>
   Password:  <input type="password" name="j_password"/>
   <input type="submit" value="Login"/>
</form>
```

The error page can be any page at all.

When the user requests a protected resource, the login page is displayed (see Figure 10–14). If the user supplies a valid username and password, then the requested page appears. Otherwise, the error page is shown.

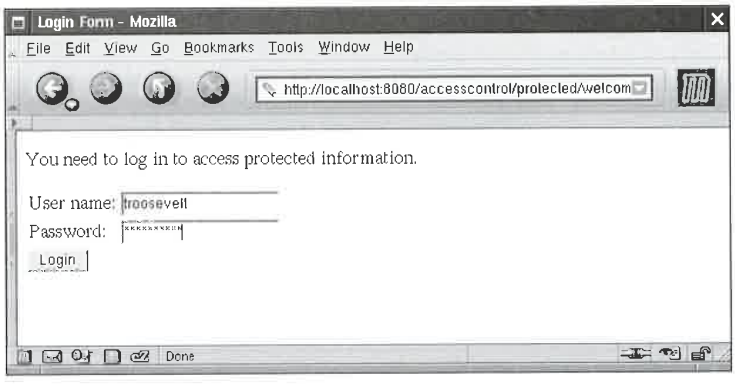


Figure 10–14 Requesting a Protected Resource

NOTE: To securely transmit the login information from the client to the server, you should use SSL. Configuring a server for SSL is beyond the scope of this book. For more information, turn to http://jakarta.apache.org/tomcat/tomcat-5.0-doc/ssl-howto.html.

You can also specify "basic" authentication by placing the following login configuration into web.xml:

```
<login-conf>
   <auth-method>BASIC</auth-method>
   <realm-name>This string shows up in the dialog</realm-name>
</login-conf>
```

In that case, the browser pops up a password dialog (see Figure 10–15). However, a professionally designed web site will probably use form-based authentication.

Figure 10–15 Basic Authentication

The `web.xml` file only describes which resources have access restrictions and which roles are allowed access. It is silent on how users, passwords, and roles are stored. You configure that information by specifying a *realm* for the web application. A realm is any mechanism for looking up user names, passwords, and roles. Tomcat supports several standard realms that access user information from one of the following sources:

- An LDAP directory
- A relational database
- An XML file (by default, `conf/tomcat-users.xml`) that is read when the server starts

To configure a realm, you supply a `Realm` element. Listing10–17 shows a typical example, a JNDI realm.

Listing 10–17 `accesscontrol/META-INF/context.xml`

```
1. <Context path="/accesscontrol" docbase="webapps/accesscontrol.war">
2. <Realm className="org.apache.catalina.realm.JNDIRealm"
3.    debug="99"
4.    connectionURL="ldap://localhost:389"
5.    connectionName="cn=Manager,dc=corejsf,dc=com"
6.    connectionPassword="secret"
7.    userPattern="uid={0},ou=people,dc=corejsf,dc=com"
8.    userPassword="userPassword"
9.    roleBase="ou=groups,dc=corejsf,dc=com"
10.   roleName="cn"
11.   roleSearch="(uniqueMember={0})"/>
12. </Context>
```

The configuration lists the URL and login information and describes how to look up users and roles.

In this example, the `Realm` element is placed inside a `Context` element in the file `META-INF/context.xml`. This is the preferred mechanism for supplying an application-specific realm.

CAUTION: You can also configure a realm in the `Engine` or `Host` element of the `server.xml` file. However, that realm is then used by the manager application in addition to your regular web application. If you want to use the manager application to install your web applications, then you must make make sure that the username and password that you use for installation is included in the realm, with a role of `manager`.

Since the servlet container is in charge of authentication and authorization, there is nothing for you to program. Nevertheless, you may want to have programmatic access to the user information. The `HttpServletRequest` yields a small amount of information, in particular, the name of the user who logged in. You get the request object from the external context:

```
ExternalContext external
      = FacesContext.getCurrentInstance().getExternalContext();
HttpServletRequest request
   = (HttpServletRequest) external.getRequest();
String user = request.getRemoteUser();
```

You can also test whether the current user belongs to a given role. For example,

```
String role = "admin";
boolean isAdmin = request.isUserInRole(role);
```

NOTE: Currently, there is no specification for logging off or for switching identities when using container-managed security. This is a problem, particularly for testing web applications. Tomcat uses cookies to represent the current user, and you need to quit and restart your browser whenever you want to switch your identity. We resorted to using Lynx for testing because it starts up much faster than a graphical web browser—see Figure 10–16.

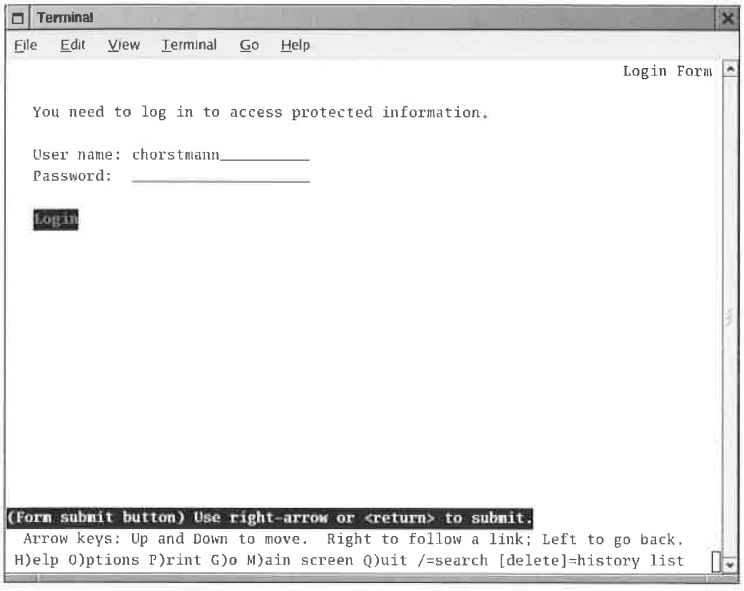


Figure 10–16 Using Lynx for Testing a Web Application

We give you a skeleton application that shows container-managed security at work. When you access the protected resource `protected/welcome.jsp` (Listing 10–18), then the authentication dialog of Listing 10–21 is displayed. You can proceed only if you enter a username and password of a user belonging to the `registereduser` or `invitedguest` role.

Just to demonstrate the servlet API, the welcome page shows the name of the registered user and lets you test for role membership (see Figure 10–16).

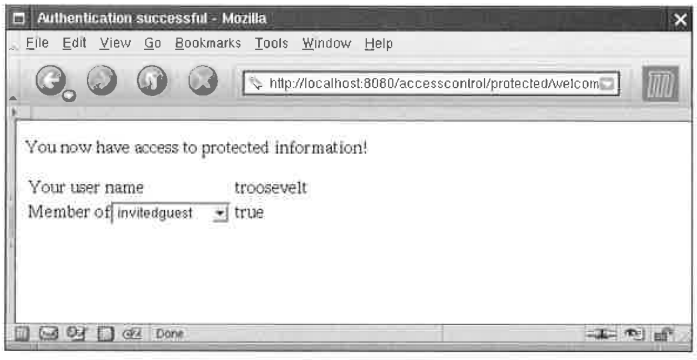


Figure 10–17 Welcome Page of the Authentication Test Application

Listing 10–18 accesscontrol/protected/welcome.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <p><h:outputText value="#{msgs.youHaveAccess}"/></p>
12.            <h:panelGrid columns="2">
13.               <h:outputText value="#{msgs.yourUserName}"/>
14.               <h:outputText value="#{user.name}"/>
15.
16.               <h:panelGroup>
17.                  <h:outputText value="#{msgs.memberOf}"/>
18.                  <h:selectOneMenu onchange="submit()" value="#{user.role}">
19.                     <f:selectItem itemValue="" itemLabel="Select a role"/>
20.                     <f:selectItem itemValue="admin" itemLabel="admin"/>
21.                     <f:selectItem itemValue="manager" itemLabel="manager"/>
22.                     <f:selectItem itemValue="registereduser"
23.                        itemLabel="registereduser"/>
24.                     <f:selectItem itemValue="invitedguest"
25.                         itemLabel="invitedguest"/>
26.                  </h:selectOneMenu>
27.               </h:panelGroup>
28.               <h:outputText value="#{user.inRole}"/>
29.            </h:panelGrid>
30.         </h:form>
31.      </body>
32.   </f:view>
33. </html>
```

Listing 10–19 accesscontrol/WEB-INF/web.xml

```
1. <?xml version="1.0"?>
2. <!DOCTYPE web-app PUBLIC
3.    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4.    "http://java.sun.com/dtd/web-app_2_3.dtd">
5.
6. <web-app>
7.    <servlet>
8.       <servlet-name>Faces Servlet</servlet-name>
```


Listing 10–19 accesscontrol/WEB-INF/web.xml (cont.)

```
9.        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.       <load-on-startup>1</load-on-startup>
11.    </servlet>
12.
13.    <servlet-mapping>
14.       <servlet-name>Faces Servlet</servlet-name>
15.       <url-pattern>*.faces</url-pattern>
16.    </servlet-mapping>
17.
18.    <welcome-file-list>
19.       <welcome-file>index.html</welcome-file>
20.    </welcome-file-list>
21.
22.    <security-constraint>
23.       <web-resource-collection>
24.          <web-resource-name>Protected Pages</web-resource-name>
25.          <url-pattern>/protected/*</url-pattern>
26.       </web-resource-collection>
27.       <auth-constraint>
28.          <role-name>registereduser</role-name>
29.       <role-name>invitedguest</role-name>
30.       </auth-constraint>
31.     </security-constraint>
32.
33.    <login-config>
34.       <auth-method>FORM</auth-method>
35.       <form-login-config>
36.          <form-login-page>/login.html</form-login-page>
37.          <form-error-page>/noauth.html</form-error-page>
38.       </form-login-config>
39.    </login-config>
40.
41.     <security-role>
42.       <role-name>registereduser</role-name>
43.     </security-role>
44.     <security-role>
45.       <role-name>invitedguest</role-name>
46.     </security-role>
47. </web-app>
```

Listing 10–20 accesscontrol/WEB-INF/classes/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.util.logging.Logger;
4. import javax.faces.context.ExternalContext;
5. import javax.faces.context.FacesContext;
6. import javax.servlet.http.HttpServletRequest;
7.
8. public class UserBean {
9.    private String name;
10.   private String role;
11.   private Logger logger = Logger.getLogger("com.corejsf");
12.
13.   public String getName() {
14.      if (name == null) getUserData();
15.      return name == null ? "" : name;
16.   }
17.
18.   public String getRole() { return role == null ? "" : role; }
19.   public void setRole(String newValue) { role = newValue; }
20.
21.   public boolean isInRole() {
22.      ExternalContext context
23.         = FacesContext.getCurrentInstance().getExternalContext();
24.      Object requestObject =  context.getRequest();
25.      if (!(requestObject instanceof HttpServletRequest)) {
26.         logger.severe("request object has type " + requestObject.getClass());
27.         return false;
28.      }
29.      HttpServletRequest request = (HttpServletRequest) requestObject;
30.      return request.isUserInRole(role);
31.   }
32.
33.   private void getUserData() {
34.      ExternalContext context
35.         = FacesContext.getCurrentInstance().getExternalContext();
36.      Object requestObject =  context.getRequest();
37.      if (!(requestObject instanceof HttpServletRequest)) {
38.         logger.severe("request object has type " + requestObject.getClass());
39.         return;
40.      }
41.      HttpServletRequest request = (HttpServletRequest) requestObject;
42.      name = request.getRemoteUser();
43.   }
44. }
```


Listing 10–21 accesscontrol/login.html

```
1. <html>
2.    <head>
3.       <title>Login Form</title>
4.    </head>
5.
6.    <body>
7.       <form method="POST" action="j_security_check">
8.          <p>You need to log in to access protected information.</p>
9.          <table>
10.            <tr>
11.               <td>User name:</td>
12.               <td>
13.                  <input type="text" name="j_username"/>
14.               </td>
15.            </tr>
16.            <tr>
17.               <td>Password:</td>
18.               <td>
19.                  <input type="password" name="j_password"/>
20.               </td>
21.            </tr>
22.         </table>
23.         <input type="submit" value="Login"/>
24.      </form>
25.   </body>
26. </html>
```

Figure 10–18 shows the directory structure of the application. The `web.xml` file in Listing 10–22 restricts access to the `protected` directory. Listing 10–23 contains the page that is displayed when authorization fails. Listing 10–25 contains the protected page. You can find the message strings in Listing 10–26 and the code for the user bean in Listing 10–24.

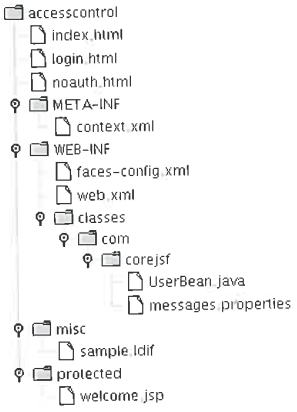


Figure 10–18 Directory Structure of the Access Control Application

Listing 10–22 accesscontrol/WEB-INF/web.xml

```
1. <?xml version="1.0"?>
2. <!DOCTYPE web-app PUBLIC
3.    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
4.    "http://java.sun.com/dtd/web-app_2_3.dtd">
5.
6. <web-app>
7.    <servlet>
8.       <servlet-name>Faces Servlet</servlet-name>
9.       <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
10.      <load-on-startup>1</load-on-startup>
11.   </servlet>
12.
13.   <servlet-mapping>
14.      <servlet-name>Faces Servlet</servlet-name>
15.      <url-pattern>*.faces</url-pattern>
16.   </servlet-mapping>
17.
18.   <welcome-file-list>
19.      <welcome-file>index.html</welcome-file>
20.   </welcome-file-list>
21.
```

Listing 10–22 accesscontrol/WEB-INF/web.xml (cont.)

```
22.    <security-constraint>
23.       <web-resource-collection>
24.          <web-resource-name>Protected Pages</web-resource-name>
25.          <url-pattern>/protected/*</url-pattern>
26.       </web-resource-collection>
27.       <auth-constraint>
28.          <role-name>registereduser</role-name>
29.       <role-name>invitedguest</role-name>
30.       </auth-constraint>
31.    </security-constraint>
32.
33.    <login-config>
34.       <auth-method>FORM</auth-method>
35.       <form-login-config>
36.          <form-login-page>/login.html</form-login-page>
37.          <form-error-page>/noauth.html</form-error-page>
38.       </form-login-config>
39.    </login-config>
40.
41.    <security-role>
42.       <role-name>registereduser</role-name>
43.    </security-role>
44.    <security-role>
45.       <role-name>invitedguest</role-name>
46.    </security-role>
47. </web-app>
```

Listing 10–23 accesscontrol/noauth.html

```
1. <html>
2.    <head>
3.       <title>Authentication failed</title>
4.    </head>
5.
6.    <body>
7.       <p>Sorry--authentication failed. Please try again.</p>
8.    </body>
9. </html>
```

Listing 10–24 accesscontrol/WEB-INF/classes/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.util.logging.Logger;
4. import javax.faces.context.ExternalContext;
5. import javax.faces.context.FacesContext;
6. import javax.servlet.http.HttpServletRequest;
7.
8. public class UserBean {
9.    private String name;
10.   private String role;
11.   private Logger logger = Logger.getLogger("com.corejsf");
12.
13.   public String getName() {
14.      if (name == null) getUserData();
15.      return name == null ? "" : name;
16.   }
17.
18.   public String getRole() { return role == null ? "" : role; }
19.   public void setRole(String newValue) { role = newValue; }
20.
21.   public boolean isInRole() {
22.      ExternalContext context
23.         = FacesContext.getCurrentInstance().getExternalContext();
24.      Object requestObject =  context.getRequest();
25.      if (!(requestObject instanceof HttpServletRequest)) {
26.         logger.severe("request object has type " + requestObject.getClass());
27.         return false;
28.      }
29.      HttpServletRequest request = (HttpServletRequest) requestObject;
30.      return request.isUserInRole(role);
31.   }
32.
33.   private void getUserData() {
34.      ExternalContext context
35.         = FacesContext.getCurrentInstance().getExternalContext();
36.      Object requestObject =  context.getRequest();
37.      if (!(requestObject instanceof HttpServletRequest)) {
38.         logger.severe("request object has type " + requestObject.getClass());
39.         return;
40.      }
41.      HttpServletRequest request = (HttpServletRequest) requestObject;
42.      name = request.getRemoteUser();
43.   }
44. }
```


Listing 10–25 accesscontrol/protected/welcome.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <p><h:outputText value="#{msgs.youHaveAccess}"/></p>
12.            <h:panelGrid columns="2">
13.               <h:outputText value="#{msgs.yourUserName}"/>
14.               <h:outputText value="#{user.name}"/>
15.
16.               <h:panelGroup>
17.                  <h:outputText value="#{msgs.memberOf}"/>
18.                  <h:selectOneMenu onchange="submit()" value="#{user.role}">
19.                     <f:selectItem itemValue="" itemLabel="Select a role"/>
20.                     <f:selectItem itemValue="admin" itemLabel="admin"/>
21.                     <f:selectItem itemValue="manager" itemLabel="manager"/>
22.                     <f:selectItem itemValue="registereduser"
23.                        itemLabel="registereduser"/>
24.                     <f:selectItem itemValue="invitedguest"
25.                         itemLabel="invitedguest"/>
26.                  </h:selectOneMenu>
27.               </h:panelGroup>
28.               <h:outputText value="#{user.inRole}"/>
29.            </h:panelGrid>
30.         </h:form>
31.      </body>
32.   </f:view>
33. </html>
```

Listing 10–26 accesscontrol/WEB-INF/classes/com/corejsf/messages.properties

```
1. title=Authentication successful
2. youHaveAccess=You now have access to protected information!
3. yourUserName=Your user name
4. memberOf=Member of
```

javax.servlet.HttpServletRequest

- `String getRemoteUser() [Servlet 2.2]`
  Gets the name of the user who is currently logged in, or `null` if there is no such user.
- `boolean isUserInRole(String role) [Servlet 2.2]`
  Tests whether the current user belongs to the given role.

Using Web Services

When a web application needs to get information from an external source, it typically uses a remote procedure call mechanism. In recent years, *web services* have emerged as a popular technology for this purpose.

Technically, a web service has two components:

- A server that can be accessed with the SOAP (Simple Object Access Protocol) transport protocol
- A description of the service in the WSDL (Web Service Description Language) format

Fortunately, you can use web services, even if you know nothing at all about SOAP and just a little about WSDL.

To make web services easy to understand, we look at a concrete example: the Amazon Web Services, described at http://www.amazon.com/gp/aws/landing.html. The Amazon Web Services allow a programmer to interact with the Amazon system for a wide variety of purposes. For example, you can get listings of all books with a given author or title, or you can fill shopping carts and place orders. Amazon makes these services available for use by companies that want to sell items to their customers, using the Amazon system as a fulfillment backend. To run our example program, you will need to sign up with Amazon and get a free developer token that lets you connect to the service.

You also need to download and install the Java Web Services Developer Pack (JWSDP) from http://java.sun.com/webservices/webservicespack.html.

NOTE: You may already use the JWSDP for the examples in this book—it bundles JSF, Tomcat, and Ant. If so, there is no need to reinstall it. If you use standalone versions of JSF, Tomcat, and Ant, you can install the JWSDP and continue to use the standalone versions to run this example.

A primary attraction of web services is that they are language-neutral. We will access the Amazon Web Services by using the Java programming language, but other developers can equally well use C++ or Visual Basic. The WSDL descriptor describes the services in a language-independent manner. For example, the WSDL for the Amazon Web Services (located at http://soap.amazon.com/schemas3/AmazonWebServices.wsdl) describes an `AuthorSearchRequest` operation as follows:

```
<operation name="AuthorSearchRequest">
   <input message="typens:AuthorSearchRequest"/>
   <output message="typens:AuthorSearchResponse"/>
</operation>
...
<message name="AuthorSearchRequest">
   <part name="AuthorSearchRequest" type="typens:AuthorRequest"/>
</message>
<message name="AuthorSearchResponse">
   <part name="return" type="typens:ProductInfo"/>
</message>
```

Elsewhere, it defines the data types. Here is the definition of `AuthorRequest`:

```
<xsd:complexType name="AuthorRequest">
   <xsd:all>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="page" type="xsd:string"/>
      <xsd:element name="mode" type="xsd:string"/>
      <xsd:element name="tag" type="xsd:string"/>
      <xsd:element name="type" type="xsd:string"/>
      <xsd:element name="devtag" type="xsd:string"/>
      <xsd:element name="sort" type="xsd:string" minOccurs="0"/>
      <xsd:element name="locale" type="xsd:string" minOccurs="0"/>
      <xsd:element name="keywords" type="xsd:string" minOccurs="0"/>
      <xsd:element name="price" type="xsd:string" minOccurs="0"/>
   </xsd:all>
</xsd:complexType>
```

When this description is translated into the Java programming language, the `AuthorRequest` type becomes a class.

```
public class AuthorRequest {
   public AuthorRequest(String author, String page, String mode, String tag,
      String type, String devtag, String sort, String locale, String keyword,
         String price) { ... }
   public String getAuthor() { ... }
   public void setAuthor(String newValue) { ... }
   public String getPage() { ... }
   public void setPage(String) { ... }
   ...
}
```

To call the search service, construct an AuthorRequest object and call the authorSearchRequest of a "port" object.

```
AmazonSearchPort asp = (AmazonSearchPort)
   (new AmazonSearchService_Impl().getAmazonSearchPort());
AuthorRequest req = new AuthorRequest(name,
   "1", "books", "", "lite", "", token, "", "", "");
ProductInfo pinfo = asp.authorSearchRequest(req);
```

The port object translates the Java object into a SOAP message, passes it to the Amazon server, and translates the returned message into a ProductInfo object. The port classes are automatically generated.

NOTE: The WSDL file does not specify *what* the service does. It only specifies the parameter and return types.

To generate the required Java classes, place into an empty directory a config.xml file with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
   xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
      <wsdl
         location="http://soap.amazon.com/schemas3/AmazonWebServices.wsdl"
         packageName="com.corejsf.amazon" />
</configuration>
```

Then run these commands:

```
jwsdp/jaxrpc/bin/wscompile.sh -import config.xml
jwsdp/jaxrpc/bin/wscompile.sh -gen config.xml
jar cvf aws.jar .
```

Here, *jwsdp* is the directory into which you installed the JWSDP, such as /usr/local/jwsdp-1.3 or c:\jwsdp-1.3. (As usual, Windows users need to use \ instead of /.)

Place the resulting JAR file into the WEB-INF/lib directory of any JSF application that uses the Amazon Web Services.

NOTE: If you like, you can also run the wscompile program from inside Ant. See the *jwsdp*/jaxrpc/samples/HelloWorld directory for an example.

Our sample application is straightforward. The user specifies an author name and clicks the "Search" button (see Figure 10–19).

Figure 10–19 Searching for Books with a Given Author

We simply show the first page of the response in a data table (see Figure 10–20). This shows that the web service is successful. We leave it as the proverbial exercise for the reader to extend the functionality of the application.

First Author	Title	Publisher	Publication Date
Lewis Carroll	Euclid and His Modern Rivals	Dover Pubns	29 March, 2004
David Carroll	World of Darkness: Time of Judgment (World of Darkness)	White Wolf Publishing Inc.	March, 2004
Lewis Carroll	Alicia En El Pais De Las Maravillas/Alice in Wonderland	Editorial Juventud, S.A.	October, 2001
Lewis Carroll	Ninas	Lumen Espana	August, 1998
Lewis Carroll	Alicia En El Pais De Las Maravillas, a Traves Del Espejo Y LA Caza Del Snark/Alice's Adventures in Wonderland, Alice Through the Mirror and the hunti	Edhasa	April, 2002
Lewis Carroll	Alicia En El Pais de Las Maravillas - A Traves del	Ediciones Catedra S.A.	August, 1999
Lewis Carroll	Through the Looking-Glass, and What Alice Found There (Collected Works of Lewis Carroll)	Classic Books	May, 2000
Lewis Carroll	The New Belfry of Christ Church, Ocford (Collected Works of Lewis Carroll)	Classic Books	May, 2000
Lectorum Publications	A Traves Del Espejo Trebol	Everest De Ediciones Y Distribucion	January, 2002
Lewis Carroll	The Vision of the Three T's (Collected Works of Lewis Carroll)	Classic Books	May, 2000

Figure 10–20 A Search Result

Figure 10–21 shows the directory structure of the application. Note the JAR file in the `WEB-INF/lib` directory.

You need different build files for this application since a large number of additional libraries are required for the SOAP calls—see Listings 10–27 and 10–28. As always, you need to customize `build.properties`. If you use the JWSDP instead of the standalone Tomcat server, you don't need to include the `jwsdp-shared` files in the `copy` target.

The bean class in Listing 10–29 contains the call to the web service. The call returns an object of type `ProductInfo`. We stash away the `Details` array contained in the returned object.

Note how the developer token is set in `faces-config.xml` (Listing 10–30). Be sure to supply your own ID in that file.

Listings 10–31 through 10–33 show the JSF pages. The `result.jsp` page contains a data table that displays information from the `Details` array that was returned by the search service.

Finally, Listing 10–34 is the message bundle.

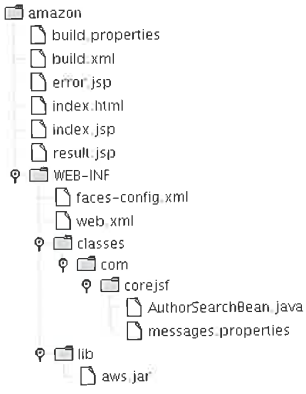


Figure 10–21 Directory Structure of the Web Service Test Application

Listing 10–27 amazon/build.xml

```
1. <project default="install">
2.
3.    <property file="build.properties"/>
4.    <property name="appdir" value="${basedir}"/>
5.    <property name="builddir" value="${appdir}/build"/>
6.    <property name="appname" value="amazon"/>
7.    <property name="warfile" value="${builddir}/${appname}.war"/>
```


Listing 10–27 amazon/build.xml (cont.)

```
8.
9.  <path id="classpath">
10.    <pathelement location="${servlet.api.jar}"/>
11.    <pathelement location="${jsp.api.jar}"/>
12.    <fileset dir="${builddir}/WEB-INF/lib">
13.       <include name="*.jar"/>
14.    </fileset>
15. </path>
16.
17. <target name="init">
18.    <tstamp/>
19. </target>
20.
21. <target name="copy" depends="init"
22.    description="Copy files to build directory.">
23.    <mkdir dir="${builddir}"/>
24.    <copy todir="${builddir}">
25.       <fileset dir="${appdir}">
26.          <exclude name="**/*.java"/>
27.          <exclude name="build/**"/>
28.          <!-- for Eclipse -->
29.          <exclude name="bin/**"/>
30.          <exclude name=".*"/>
31.       </fileset>
32.    </copy>
33.    <copy todir="${builddir}/WEB-INF/lib">
34.       <fileset dir="${jsf.lib.dir}" includes="${jsf.libs}"/>
35.       <fileset dir="${jstl.lib.dir}" includes="${jstl.libs}"/>
36.       <fileset dir="${commons.lib.dir}" includes="${commons.libs}"/>
37.       <fileset dir="${jaxrpc.lib.dir}" includes="*.jar"/>
38.       <fileset dir="${saaj.lib.dir}" includes="*.jar"/>
39.       <fileset dir="${jwsdp-shared.lib.dir}" includes="${jwsdp-shared.libs}"/>
40.    </copy>
41. </target>
42.
43. <target name="compile" depends="copy"
44.    description="Compile source files.">
45.    <javac
46.       srcdir="${appdir}/WEB-INF/classes"
47.       destdir="${builddir}/WEB-INF/classes"
48.       debug="true"
49.       deprecation="true">
50.       <include name="**/*.java"/>
51.       <classpath refid="classpath"/>
52.    </javac>
```

Listing 10–27 amazon/build.xml (cont.)

```
53.    </target>
54.
55.    <target name="war" depends="compile"
56.       description="Build WAR file.">
57.       <delete file="${warfile}"/>
58.       <jar jarfile="${warfile}" basedir="${builddir}"/>
59.    </target>
60.
61.    <target name="install" depends="war"
62.       description="Deploy web application.">
63.       <copy file="${warfile}" todir="${tomcat.dir}/webapps"/>
64.    </target>
65.
66. </project>
```

Listing 10–28 amazon/build.properties

```
1. jsf.dir=/usr/local/jsf-1_0
2. tomcat.dir=/usr/local/jakarta-tomcat-5.0.19
3.
4. username=me
5. password=secret
6. manager.url=http://localhost:8080/manager
7.
8. servlet.api.jar=${tomcat.dir}/common/lib/servlet-api.jar
9. jsp.api.jar=${tomcat.dir}/common/lib/jsp-api.jar
10.
11. jsf.lib.dir=${jsf.dir}/lib
12. jstl.lib.dir=${tomcat.dir}/webapps/jsp-examples/WEB-INF/lib
13. commons.lib.dir=${tomcat.dir}/server/lib
14.
15. jsf.libs=jsf-api.jar,jsf-impl.jar
16. jstl.libs=jstl.jar,standard.jar
17. commons.libs=commons-beanutils.jar,commons-digester.jar
18.
19. jwsdp.dir=/home/apps/jwsdp-1.3
20. jaxp.lib.dir=${jwsdp.dir}/jaxp/lib
21. jaxrpc.lib.dir=${jwsdp.dir}/jaxrpc/lib
22. saaj.lib.dir=${jwsdp.dir}/saaj/lib
23. jwsdp-shared.lib.dir=${jwsdp.dir}/jwsdp-shared/lib
24. jaxp.api.jar=${jaxp.lib.dir}/jaxp-api.jar
25. jwsdp-shared.libs=jax-qname.jar,namespace.jar,activa-
tion.jar,jaas.jar,mail.jar,xsdlib.jar,providerutil.jar
```


Listing 10–29 amazon/WEB-INF/classes/com/corejsf/AmazonSearchBean.java

```
1. package com.corejsf;
2.
3. import com.corejsf.amazon.AmazonSearchPort;
4. import com.corejsf.amazon.AmazonSearchService_Impl;
5. import com.corejsf.amazon.AuthorRequest;
6. import com.corejsf.amazon.Details;
7. import com.corejsf.amazon.ProductInfo;
8.
9. public class AuthorSearchBean {
10.    private String name;
11.    private String type;
12.    private Details[] details;
13.    private String token;
14.
15.    public String getName() { return name; }
16.    public void setName(String newValue) { name = newValue; }
17.
18.    public void setToken(String newValue) { token = newValue; }
19.
20.    public String search() {
21.       try{
22.          AmazonSearchPort asp = (AmazonSearchPort)
23.             (new AmazonSearchService_Impl().getAmazonSearchPort());
24.
25.          AuthorRequest req = new AuthorRequest(name,
26.              "1", "books", "", "lite", "", token, "", "", "");
27.          ProductInfo pinfo = asp.authorSearchRequest(req);
28.          details = pinfo.getDetails();
29.          return "success";
30.       } catch(Exception e) {
31.          e.printStackTrace();
32.          return "failure";
33.       }
34.    }
35.
36.    public Details[] getDetails() { return details; }
37. }
```

Listing 10–30 amazon/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2.
3. <!DOCTYPE faces-config PUBLIC
4.   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
5.   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
6.
7. <faces-config>
8.
9.    <navigation-rule>
10.      <from-view-id>/index.jsp</from-view-id>
11.      <navigation-case>
12.         <from-outcome>success</from-outcome>
13.         <to-view-id>/result.jsp</to-view-id>
14.      </navigation-case>
15.      <navigation-case>
16.         <from-outcome>failure</from-outcome>
17.         <to-view-id>/error.jsp</to-view-id>
18.      </navigation-case>
19.   </navigation-rule>
20.   <navigation-rule>
21.      <from-view-id>/result.jsp</from-view-id>
22.      <navigation-case>
23.         <from-outcome>back</from-outcome>
24.         <to-view-id>/index.jsp</to-view-id>
25.      </navigation-case>
26.   </navigation-rule>
27.   <navigation-rule>
28.      <from-view-id>/error.jsp</from-view-id>
29.      <navigation-case>
30.         <from-outcome>continue</from-outcome>
31.         <to-view-id>/index.jsp</to-view-id>
32.      </navigation-case>
33.   </navigation-rule>
34.
35.   <managed-bean>
36.      <managed-bean-name>authorSearch</managed-bean-name>
37.      <managed-bean-class>com.corejsf.AuthorSearchBean</managed-bean-class>
38.      <managed-bean-scope>session</managed-bean-scope>
39.      <managed-property>
40.         <property-name>token</property-name>
41.         <value>Your token goes here</value>
42.      </managed-property>
43.   </managed-bean>
44.
45. </faces-config>
```


Listing 10–31 amazon/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <link href="styles.css" rel="stylesheet" type="text/css"/>
8.          <title><h:outputText value="#{msgs.title}"/></title>
9.       </head>
10.      <body>
11.         <h:form>
12.            <h1><h:outputText value="#{msgs.authorSearch}"/></h1>
13.            <h:outputText value="#{msgs.author}"/>
14.            <h:inputText value="#{authorSearch.name}"/>
15.            <h:commandButton value="#{msgs.search}"
16.               action="#{authorSearch.search}"/>
17.         </h:form>
18.      </body>
19.   </f:view>
20. </html>
```

Listing 10–32 amazon/result.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.searchResult}"/></h1>
12.            <h:dataTable value="#{authorSearch.details}" var="detail"
13.               border="1">
14.               <h:column>
15.                  <f:facet name="header">
16.                     <h:outputText value="#{msgs.author1}"/>
1.
```

Listing 10–32 amazon/result.jsp (cont.)

```
                    </f:facet>
                    <h:outputText value="#{detail.authors[0]}"/>
                 </h:column>
                 <h:column>
                    <f:facet name="header">
                       <h:outputText value="#{msgs.title}"/>
                    </f:facet>
                    <h:outputText value="#{detail.productName}"/>
                 </h:column>
                 <h:column>
                    <f:facet name="header">
                       <h:outputText value="#{msgs.publisher}"/>
                    </f:facet>
                    <h:outputText value="#{detail.manufacturer}"/>
                 </h:column>
                 <h:column>
                    <f:facet name="header">
                       <h:outputText value="#{msgs.pubdate}"/>
                    </f:facet>
                    <h:outputText value="#{detail.releaseDate}"/>
                 </h:column>
              </h:dataTable>
              <h:commandButton value="#{msgs.back}" action="back"/>
           </h:form>
        </body>
     </f:view>
  </html>
```

Listing 10–33 amazon/error.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.    <f:view>
5.       <head>
6.          <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h1><h:outputText value="#{msgs.internalError}"/></h1>
12.            <p><h:outputText value="#{msgs.internalError_detail}"/></p>
13.            <p>
```


Listing 10–33 amazon/error.jsp (cont.)

```
            <h:commandButton value="#{msgs.continue}" action="login"/>
        </p>
      </h:form>
    </body>
  </f:view>
</html>
```

Listing 10–34 amazon/WEB-INF/classes/com/corejsf/messages.properties

```
title=A Faces Application that Invokes a Web Service
authorSearch=Author Search at Amazon
author=Author
format=Format
search=Search
searchResult=Search Result
internalError=Internal Error
internalError_detail=To our chagrin, an internal error has occurred. \
   Please report this problem to our technical staff.
continue=Continue
author1=First Author
title=Title
publisher=Publisher
pubdate=Publication Date
back=Back
```

You have now seen how your web applications can connect to external services, such as databases, directories, and web services. Here are some general considerations to keep in mind.

- Libraries are placed either in the `WEB-INF/lib` directory of the web application or in the `common/lib` directory of the servlet container. You would do the latter only for libraries that are used by many applications, such as JDBC drivers.
- Servlet containers typically provide common services for database connection pooling, authentication realms, and so on. JNDI provides a convenient mechanism for locating the classes that are needed to access these services.
- Configuration parameters can be placed into `faces-config.xml` or `web.xml`. The former is more appropriate for parameters that are intrinsic to the web application; the latter should be used for parameters that are determined at deployment time.

JAVA™ Programming

core JAVASERVER

Without assuming knowledge of JSP and servlets, *Core JavaServer Faces*:

- shows how to build more robust applications and avoid tedious handcoding
- answers questions most developers don't even know to ask
- demonstrates how to use JavaServer Faces with Tiles to build consistent user interfaces
- provides hints, tips, and explicit "how-to" information that allows you to quickly become more productive
- explains how to integrate JavaServer Faces with databases, use directory services, wireless apps, and Web services
- shows how to implement custom components, renderers, converters, and validators
- teaches best practices and good habits like using style sheets and message bundles
- covers all of the JavaServer Faces tags and how to create new tag libraries

JavaServer Faces promises to bring rapid us[illegible] server-side Java. It allows developers to painlessly write server-side applications without worrying about the complexities of dealing with browsers and Web servers. It also automates low-level, boring details like control flow and moving code between Web forms and business logic.

JavaServer Faces was designed to support drag and drop development of server-side applications, but you can also think of it as a conceptual layer on top of servlets and JavaServer Pages (JSP). Experienced JSP developers will find that JavaServer Faces provides much of the plumbing that they currently have to implement by hand. If you already use a server-side framework such as Struts, you will find that JavaServer Faces uses a similar architecture, but is more flexible and extensible. JavaServer Faces also comes with server-side components and an event model, which are fundamentally similar to the same concepts in Swing.

JavaServer Faces is quickly becoming the standard Web-application framework. *Core JavaServer Faces* is the one book you need to master this powerful and time-saving technology.

About the Authors

Cay Horstmann is a Professor of Computer Science at San Jose State University. Previously, he was Vice President and Chief Technology Officer at Preview Systems Inc. and a consultant for major corporations, universities, and organizations on C++, Java, and Internet programming. He is the author of many successful professional and academic books, including the international best-seller *Core Java*, also published by Sun Microsystems Press.

David Geary is the president of Sabreware, Inc., a Java training and consulting company. David has developed object-oriented software for more than 20 years and worked on the Java APIs at Sun Microsystems from 1994 to 1997. He is the author of six Java books, including the *Graphic Java* series, *Advanced JavaServer Pages*, and *Core JSTL*. David is a member of the expert groups for the JavaServer Pages Standard Tag Library (JSTL) and JavaServer Faces; was one of the earliest contributors to the Apache Struts application framework; and wrote test questions for Sun's Web Component Developer.

PRENTICE HALL
Professional Technical Reference
Upper Saddle River, NJ 07458
www.phptr.com

$49.99 U.S.
$71.99 Canada

ISBN 0-13-146305-5