Custom Components, Converters, and Validators

Topics in This Chapter

Chapter 9

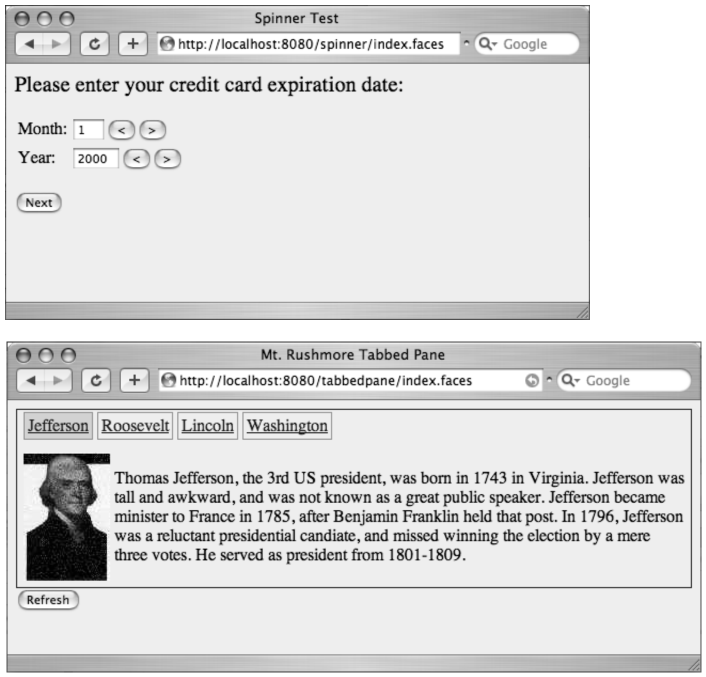
JSF provides a basic set of components for building HTML-based web applications, such as text fields, checkboxes, buttons, and so on. However, most user interface designers want more advanced components, such as calendars, tabbed panes, or navigation trees, that are not part of the standard JSF component 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 spinner and a tabbed pane, shown in Figure 9–1—to illustrate the various aspects of creating custom components.

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 expression to associate a text field's value with a bean property, so you could use value expressions to wire calendar cells to bean properties. JSF standard input components fire value change events when their value changes, so 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 387
- "Calling Converters from External Renderers" on page 393
- "Supporting Value Change Listeners" on page 394
- "Supporting Method Expressions" on page 396



Please enter your credit card expiration date:

Thomas Jefferson, the 3rd US president, was born in 1743 in Virginia. Jefferson was tall and awkward, and was not known as a great public speaker. Jefferson became minister to France in 1785, after Benjamin Franklin held that post. In 1796, Jefferson was a reluctant presidential candiate, and missed winning the election by a mere three votes. He served as president from 1801-1809.

Figure 9–1 The spinner and the tabbed pane

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

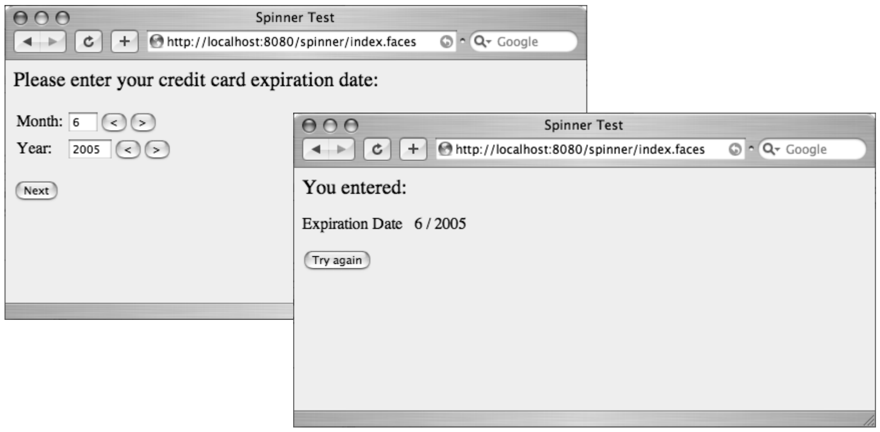
- "Processing `SelectItem` Children" page 411
- "Processing Facets" on page 412
- "Encoding CSS Styles" on page 413
- "Using Hidden Fields" on page 415
- "Saving and Restoring State" on page 415
- "Firing Action Events" on page 418

Classes for Implementing Custom Components

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 into 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.


Spinner Test
http://localhost:8080/spinner/index.faces
Google
Next
Spinner Test
http://localhost:8080/spinner/index.faces
Google
Try again

Please enter your credit card expiration date:

Month: 6

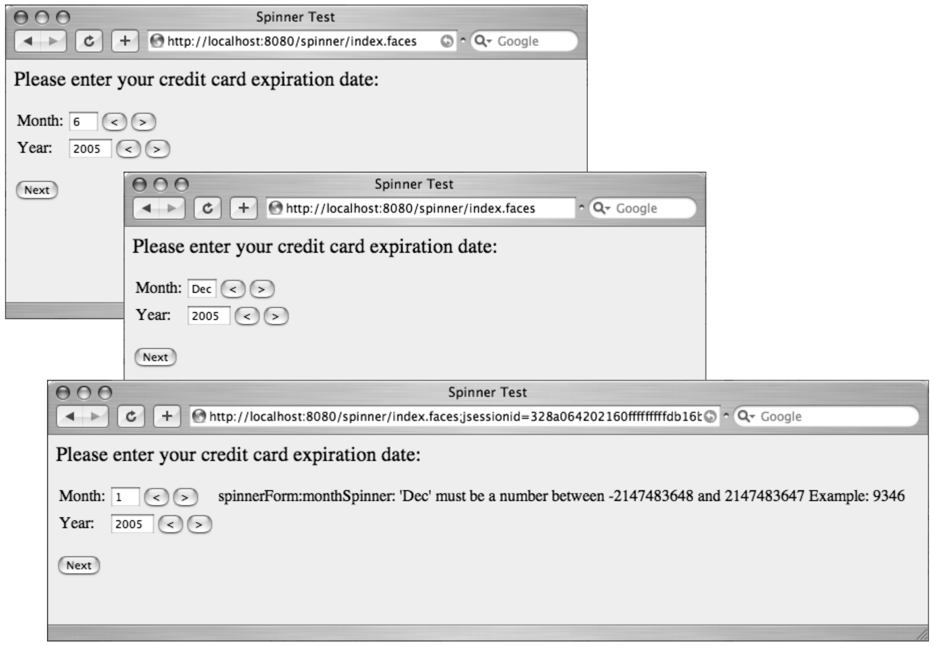
Year: 2005

You entered:

Expiration Date 6 / 2005

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 369.


Spinner Test
http://localhost:8080/spinner/index.faces
Google
Next
Spinner Test
http://localhost:8080/spinner/index.faces
Google
Next
Spinner Test
http://localhost:8080/spinner/index.faces;jsessionid=328a064202160ffffffffdb16t
Google
Next

Please enter your credit card expiration date:

Month: 6

Year: 2005

Please enter your credit card expiration date:

Month: Dec

Year: 2005

Please enter your credit card expiration date:

Month: 1 spinnerForm:monthSpinner: 'Dec' must be a number between -2147483648 and 2147483647 Example: 9346

Year: 2005

Figure 9–3 Handling conversion failures

Here is 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.

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 expression—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.UIComponentELTag` 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 does not 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 372 for more information on tag classes.

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

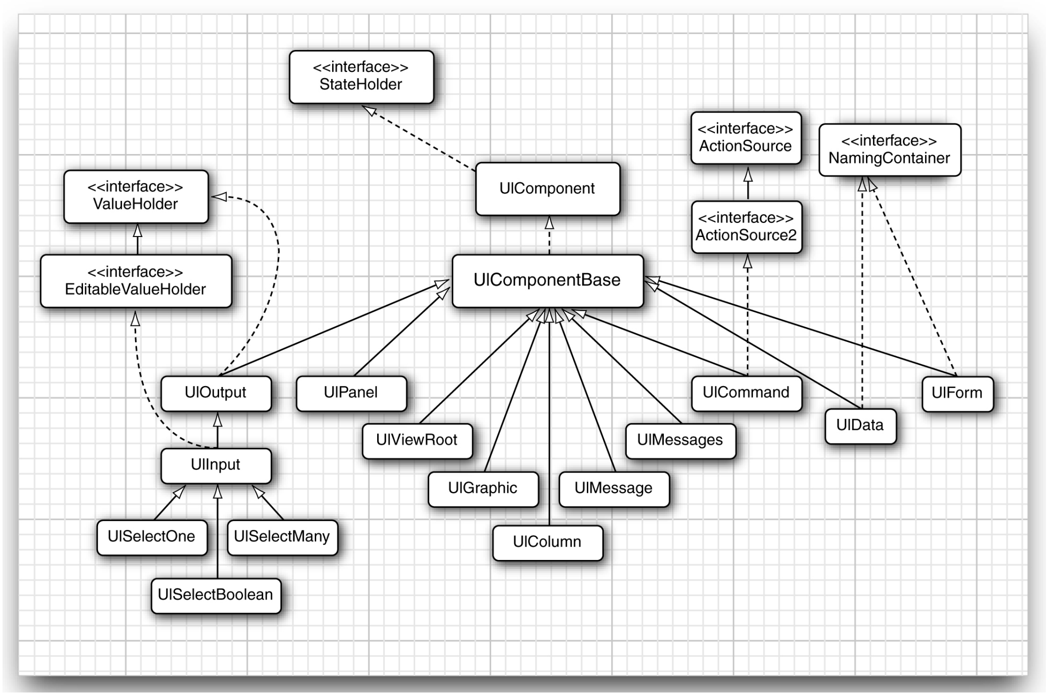
- To *render* the user interface by encoding markup
- To *process user input* by decoding the current HTTP request

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 387, we show you how to implement a separate renderer for the spinner.

A component's `UI` class must extend the `UIComponent` class. That class defines over 40 abstract methods, so you will want to extend an existing class that implements them. 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 `ActionSource2` interface.

NOTE: The `ActionSource2` interface was added in JSF1.2. An `ActionSource` has methods to manage action listeners. An `ActionSource2` additionally manages actions. In JSF 1.1, the `ActionSource` interface handled both actions and action listeners.



<<interface>> StateHolder
<<interface>> ActionSource
<<interface>> NamingContainer
<<interface>> ValueHolder
UIComponent
<<interface>> ActionSource2
<<interface>> EditableValueHolder
UIComponentBase
UIOutput
UIPanel
UICommand
UIForm
UIData
UIViewRoot
UIMessages
UIInput
UIGraphic
UIMessage
UISelectOne
UISelectMany
UIColumn
UISelectBoolean

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.UIComponentELTag`
- `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 expressions*. This is another general-purpose map that you can use to store arbitrary value expressions. For example, if a spinner tag has an attribute `value="#{cardExpirationDate.month}"`, then the component stores a `ValueExpression` object for the given value expression 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.
- `ActionSource2` defines methods for managing 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 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.

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, and validators. In this chapter, we are mostly interested in using the `Application` class to create converters, and to obtain an expression factory for value and method expressions.

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 do not have children, you do not need to implement `encodeChildren`. Since we do not need to worry what gets encoded before or after the children, 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 is 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 do not 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 does not 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 372, the `SpinnerTag` class stores the tag's `size` attribute in the component's value expression map, and the `get` method of the map returned by `getAttributes` evaluates the value expression.)

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.

API **`javax.faces.component.UIComponent`**

- `void encodeBegin(FacesContext context) throws IOException`
  The method called in the Render Response phase of the JSF life cycle, only if the component's renderer type is `null`. This signifies 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.

  As of JSF 1.2, the map also gets attributes that are defined by value expressions. If `get` is called with a name that is not a property or attribute but a key in the component's value expression map, then the value of the associated expression is returned.

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 9–17 on page 419.

NOTE: JSF invokes a component's `encodeChildren` method if the component returns `true` from `getRendersChildren`. Interestingly, it does not 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 can only 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 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 356, 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 are decoding, then we know that the decrement button was activated. If the `decode` method finds a request parameter named *clientId*`.more`, then we know that the increment button was activated.

If neither parameter exists, we know that 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 redisplayed.

`javax.faces.component.UIComponent`

- `void decode(FacesContext context)`

  The method called by JSF 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 cannot 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 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/src/java/com/corejsf/UISpinner.java`

```
package com.corejsf;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.convert.IntegerConverter;

public class UISpinner extends UIInput {
   private static final String MORE = ".more";
   private static final String LESS = ".less";

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

custom-components-VM.fm Page 370 Monday, April 2, 2007 12:10 PM

Listing 9–1 spinner/src/java/com/corejsf/UISpinner.java (cont.)

```java

   public void encodeBegin(FacesContext context) throws IOException {
      ResponseWriter writer = context.getResponseWriter();
      String clientId = getClientId(context);

      encodeInputField(writer, clientId);
      encodeDecrementButton(writer, clientId);
      encodeIncrementButton(writer, clientId);
   }

   public void decode(FacesContext context) {
      Map<String, String> 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));
      }
   }

   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");

```

Listing 9–1 spinner/src/java/com/corejsf/UISpinner.java (cont.)

```
      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");
   }

   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;
   }
}
```

API *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.

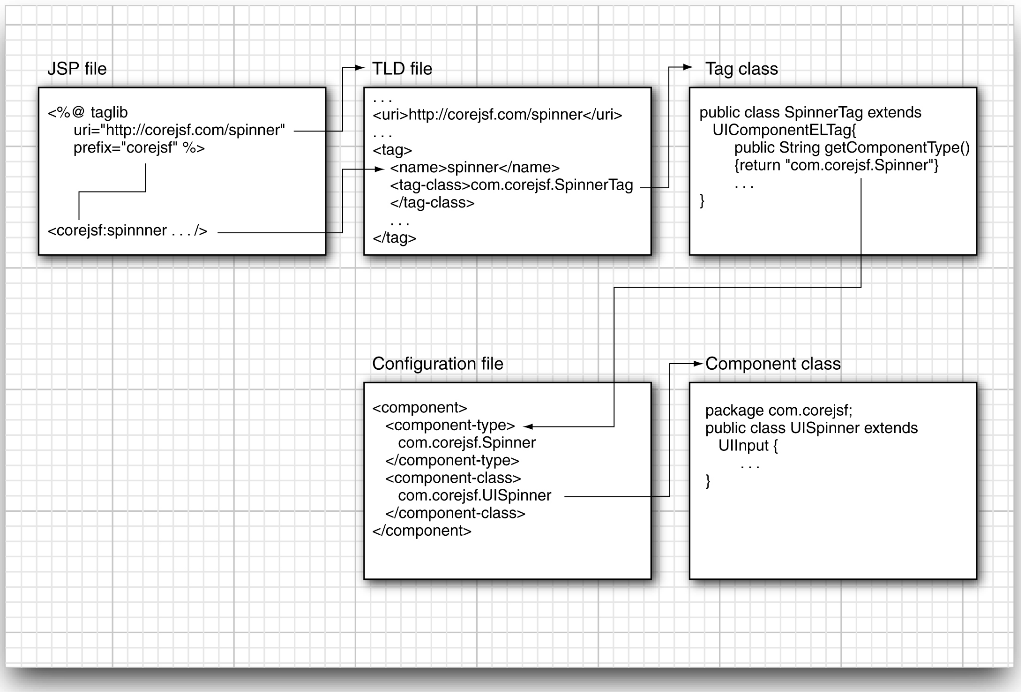
Implementing Custom Component Tags

Now that you have seen how to implement the spinner component, there is one remaining chore: to supply a tag handler. The process is somewhat byzantine, and you may find it helpful to refer to Figure 9–5 as we discuss each step.

NOTE: Custom tags use the JavaServer Pages tag library mechanism. For more information on JSP custom tags, see Chapter 7 of the Java EE 5 tutorial at `http://java.sun.com/javaee/5/docs/tutorial/doc/index.html`.

The TLD File

You need to produce a TLD (tag library descriptor) file that describes one or more tags and their attributes. Place that file into the `WEB-INF` directory. Listing 9–2 shows the TLD file that describes our `spinner` custom tag.



```
<%@ taglib
    uri="http://corejsf.com/spinner"
    prefix="corejsf" %>

<corejsf:spinnner . . . />
```

```
. . .
<uri>http://corejsf.com/spinner</uri>
. . .
<tag>
   <name>spinner</name>
   <tag-class>com.corejsf.SpinnerTag
   </tag-class>
   . . .
</tag>
```

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

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

```
package com.corejsf;
public class UISpinner extends
   UIInput {
      . . .
}
```

Figure 9–5 Locating a custom component

The purpose of the file is to specify the class name for the tag handler (`com.corejsf.SpinnerTag`) and the permitted attributes of the tag (in our case, `id`, `rendered`, `minimum`, `maximum`, `size`, and `value`).

Note the `uri` tag that identifies the tag library:

```
<uri>http://corejsf.com/spinner</uri>
```

This is the URI that you reference in a `taglib` directive of the JSF page, such as

```
<%@ taglib uri="http://corejsf.com/spinner" prefix="corejsf" %>
```

This `taglib` directive is the analog to the directives that define the standard `f` and `h` prefixes in every JSF page.

NOTE: You can choose arbitrary names for the TLD files—only the `.tld` extension matters. The JSF implementation searches for TLD files in the following locations:

- The `WEB-INF` directory or one of its subdirectories
- The `META-INF` directory or any JAR file in the `WEB-INF/lib` directory

The latter is useful if you want to package your converters as reusable JAR files.

NOTE: In the following, we describe TLD files and tag handlers for JSF 1.2. The details are quite different for JSF 1.1. See page 383 for details.

Most attribute definitions in the TLD file contain a `deferred-value` child element, like this:

```
<attribute>
   <description>The spinner minimum value</description>
   <name>minimum</name>
   <deferred-value>
      <type>int</type>
   </deferred-value>
</attribute>
```

This syntax indicates that the attribute is defined by a value expression. The attribute value can be a constant string or a string that contains `#{...}` expressions. The `type` element specifies the Java type of the value expression. In our example, `minimum` is defined by a value expression of type `int`.

Some attributes are specified by method expressions instead of value expressions. For example, to define an action listener, you use the following tag:

```
<attribute>
   <name>actionListener</name>
   <deferred-method>
      <method-signature>
         void actionListener(javax.faces.event.ActionEvent)
      </method-signature>
   </deferred-method>
</attribute>
```

Generally, you want to allow value or method expressions for attributes. One exception is the `id` attribute that is defined as a *runtime expression value*—that is, a JSP expression but not a JSF value expression.

```
<attribute>
   <description>The client id of this component</description>
   <name>id</name>
   <rtexprvalue>true</rtexprvalue>
</attribute>
```

Listing 9–2 spinner/web/WEB-INF/spinner.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
   version="2.1">
   <tlib-version>1.1</tlib-version>
   <short-name>spinner</short-name>
   <uri>http://corejsf.com/spinner</uri>

   <tag>
      <name>spinner</name>
      <tag-class>com.corejsf.SpinnerTag</tag-class>
      <body-content>empty</body-content>
      <attribute>
         <description>A value binding that points to a bean property</description>
         <name>binding</name>
         <deferred-value>
            <type>javax.faces.component.UIComponent</type>
         </deferred-value>
      </attribute>

      <attribute>
         <description>The client id of this component</description>
         <name>id</name>
```

Listing 9–2 spinner/web/WEB-INF/spinner.tld (cont.)

```
         <rtexprvalue>true</rtexprvalue>
      </attribute>

      <attribute>
         <description>Is this component rendered?</description>
         <name>rendered</name>
         <deferred-value>
            <type>boolean</type>
         </deferred-value>
      </attribute>

      <attribute>
         <description>The spinner minimum value</description>
         <name>minimum</name>
         <deferred-value>
            <type>int</type>
         </deferred-value>
      </attribute>

      <attribute>
         <description>The spinner maximum value</description>
         <name>maximum</name>
         <deferred-value>
            <type>int</type>
         </deferred-value>
      </attribute>

      <attribute>
         <description>The size of the input field</description>
         <name>size</name>
         <deferred-value>
            <type>int</type>
         </deferred-value>
      </attribute>

      <attribute>
         <description>The value of the spinner</description>
         <name>value</name>
         <required>true</required>
         <deferred-value>
            <type>int</type>
         </deferred-value>
      </attribute>
   </tag>
</taglib>
```

The Tag Handler Class

Together with the TLD file, you need to supply a *tag handler class* for each custom tag. For a component tag, the tag handler class should be a subclass of UIComponentELTag. As you will see later, the tag handlers for custom converters need to subclass ComponentELTag, and custom validator tag handlers need to subclass ValidatorELTag.

Component 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

Now we look at the implementation of the SpinnerTag class:

```
public class SpinnerTag extends UIComponentELTag {
   private ValueExpression minimum;
   private ValueExpression maximum;
   private ValueExpression size;
   private ValueExpression value;
   ...
}
```

The spinner tag class has an instance field for each attribute. The tag class should keep all attributes as ValueExpression objects.

NOTE: The id, binding, and rendered attributes are handled by the UIComponentELTag superclass.

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; }
```

CAUTION: When the getRendererType method of the tag handler returns null, you must call setRendererType in the component constructor. If the component renders itself, call setRendererType(null).

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

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

When the tag is processed in the JSF page, the tag attribute value is converted to a ValueExpression object, and the setter method is called. Getter methods are not needed.

Tag handlers 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 expressions, not properties:

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

   component.setValueExpression("size", size);
   component.setValueExpression("minimum", minimum);
   component.setValueExpression("maximum", maximum);
   component.setValueExpression("value", value);
}
```

Later, you can evaluate the value expression simply by using the attributes map. For example,

```
Integer minimum = (Integer) component.getAttributes().get("minimum");
```

The get method of the attributes map checks that the component has a value expression with the given key, and it evaluates it.

Finally, you need to define a release method that resets all instance fields to their defaults:

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

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

This method is necessary because the JSF implementation may cache tag handler objects and reuse them for parsing tags. If a tag handler is reused, it should not have leftover settings from a previous tag.

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

Listing 9–3 contains the complete code for the `SpinnerTag` tag handler.

Listing 9–3 spinner/src/java/com/corejsf/SpinnerTag.java

```java
package com.corejsf;

import javax.el.ValueExpression;
import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentELTag;

public class SpinnerTag extends UIComponentELTag {
   private ValueExpression minimum = null;
   private ValueExpression maximum = null;
   private ValueExpression size = null;
   private ValueExpression value = null;

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

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

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

      component.setValueExpression("size", size);
      component.setValueExpression("minimum", minimum);
      component.setValueExpression("maximum", maximum);
      component.setValueExpression("value", value);
   }

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

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

javax.faces.webapp.UIComponentELTag JSF 1.2

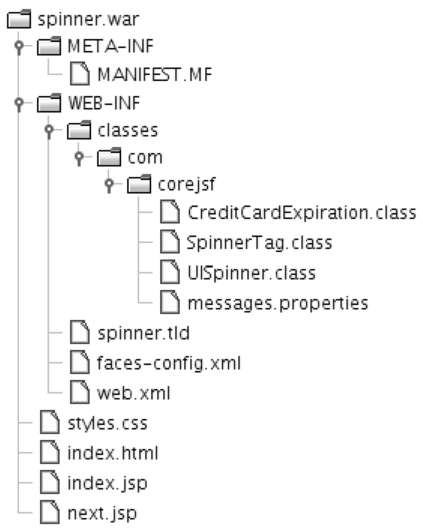
- `void setProperties(UIComponent component)`
  Transfers tag attribute values to component properties, attributes, or both. Custom components must call the superclass `setProperties` method to make sure that properties are set for the attributes `UIComponentELTag` supports: `binding`, `id`, and `rendered`.
- `void release()`
  Clears the state of this tag so that it can be reused.

javax.faces.component.UIComponent

- `void setValueExpression(String name, ValueExpression expr)` JSF 1.2
  If the expression is a constant, without `#{...}` expressions, then it is evaluated and the (`name`, `value`) pair is put into the component's attribute map. Otherwise, the (`name`, `expr`) pair is put into the component's value expression map.

The Spinner Application

After a number of different perspectives of the spinner component, it is 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 356. The directory structure is shown in Figure 9–6 and the code is shown in Listings 9–4 through 9–9.



```
spinner.war
  META-INF
    MANIFEST.MF
  WEB-INF
    classes
      com
        corejsf
          CreditCardExpiration.class
          SpinnerTag.class
          UISpinner.class
          messages.properties
    spinner.tld
    faces-config.xml
    web.xml
  styles.css
  index.html
  index.jsp
  next.jsp
```

Figure 9–6 Directory structure for the spinner example

Listing 9–4 spinner/web/index.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <%@ taglib uri="http://corejsf.com/spinner" prefix="corejsf" %>
   <f:view>
      <head>
         <link href="styles.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="#{msgs.windowTitle}"/></title>
      </head>

      <body>
         <h:form id="spinnerForm">
            <h:outputText value="#{msgs.creditCardExpirationPrompt}"
               styleClass="pageTitle"/>
            <p/>
            <h:panelGrid columns="3">
               <h:outputText value="#{msgs.monthPrompt}"/>
               <corejsf:spinner value="#{cardExpirationDate.month}"
                  id="monthSpinner" minimum="1" maximum="12" size="3"/>
               <h:message for="monthSpinner"/>
               <h:outputText value="#{msgs.yearPrompt}"/>
               <corejsf:spinner value="#{cardExpirationDate.year}"
                  id="yearSpinner" minimum="1900" maximum="2100" size="5"/>
               <h:message for="yearSpinner"/>
            </h:panelGrid>
            <p/>
            <h:commandButton value="#{msgs.nextButtonPrompt}" action="next"/>
         </h:form>
      </body>
   </f:view>
</html>
```

Listing 9–5 spinner/web/next.jsp

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

   <f:view>
      <head>
         <link href="styles.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="#{msgs.windowTitle}"/></title>
      </head>
```

Listing 9–5 spinner/web/next.jsp (cont.)

```
      <body>
         <h:form>
            <h:outputText value="#{msgs.youEnteredPrompt}" styleClass="pageTitle"/>
            <p>
            <h:outputText value="#{msgs.expirationDatePrompt}"/>
               <h:outputText value="#{cardExpirationDate.month}"/> /
               <h:outputText value="#{cardExpirationDate.year}"/>
            <p>
            <h:commandButton value="Try again" action="again"/>
         </h:form>
      </body>
   </f:view>
</html>
```

Listing 9–6 spinner/src/java/com/corejsf/CreditCardExpiration.java

```
package com.corejsf;

public class CreditCardExpiration {
   private int month = 1;
   private int year = 2000;

   // PROPERTY: month
   public int getMonth() { return month; }
   public void setMonth(int newValue) { month = newValue; }

   // PROPERTY: year
   public int getYear() { return year; }
   public void setYear(int newValue) { year = newValue; }
}
```

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

```
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">
   <navigation-rule>
      <from-view-id>/index.jsp</from-view-id>
      <navigation-case>
```

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

```
         <from-outcome>next</from-outcome>
         <to-view-id>/next.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

   <navigation-rule>
      <from-view-id>/next.jsp</from-view-id>
      <navigation-case>
         <from-outcome>again</from-outcome>
         <to-view-id>/index.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

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

   <managed-bean>
      <managed-bean-name>cardExpirationDate</managed-bean-name>
      <managed-bean-class>com.corejsf.CreditCardExpiration</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>

   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
   </application>
</faces-config>
```

Listing 9–8 spinner/src/java/com/corejsf/messages.properties

```
windowTitle=Spinner Test
creditCardExpirationPrompt=Please enter your credit card expiration date:
monthPrompt=Month:
yearPrompt=Year:
nextButtonPrompt=Next
youEnteredPrompt=You entered:
expirationDatePrompt=Expiration Date
changes=Changes:
```

Listing 9–9 spinner/web/styles.css

```
body {
   background: #eee;
}
.pageTitle {
   font-size: 1.25em;
}
```

Defining Tag Handlers in JSF 1.1

In JSF 1.1, support for tag handlers was less elegant than in JSF 1.2. This section contains the details. Feel free to skip it if you do not have to write components that are backward compatible with JSF 1.1.

JSF 1.1 provides two separate tag superclasses, `UIComponentTag` and `UIComponentBodyTag`. 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 tags in JSF 1.1 extend `UIComponentBodyTag`: `f:view`, `f:verbatim`, `h:commandLink`, and `h:outputLink`. A spinner component does not process its body, so it would extend `UIComponentTag`.

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

Before JSF 1.2, TLD files used a `DOCTYPE` declaration instead of a schema declaration, like this:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
   <tlib-version>0.03</tlib-version>
   <jsp-version>1.2</jsp-version>
   ...
</taglib>
```

More important, you cannot use any `deferred-value` or `deferred-method` child elements for attributes. Instead, the tag handler class gets passed the expression strings. It must convert them to objects of type `ValueBinding` or `MethodBinding`. These are the JSF 1.1 analogs of the `ValueExpression` and `MethodExpression` classes.

Thus, the tag handler class must define setters for strings, like this:

```
public class SpinnerTag extends UIComponentTag {
   private String minimum;
   private String maximum;
   ...
   public void setMinimum(String newValue) { minimum = newValue; }
   public void setMaximum(String newValue) { maximum = newValue; }
   ...
}
```

In the `setProperties` method, you check whether the string is in fact a value binding. If so, you convert it to an object of type `ValueBinding`, the precursor to the `ValueExpression` class. Otherwise, you convert the string to the appropriate type and set it as a component attribute.

This conversion is rather tedious, and it is useful to define a helper method, such as the following:

```
public void setInteger(UIComponent component, String name, String expr) {
   if (expr == null) return null;
   else if (UIComponentTag.isValueReference(expr)) {
      FacesContext context = FacesContext.getCurrentInstance();
      Application app = context.getApplication();
      ValueBinding binding = app.createValueBinding(expr);
      component.setValueBinding(name, binding);
   }
   else
      component.getAttributes().put(name, new Integer(expr));
}
```

NOTE: The map returned by the `UIComponent.getAttributes` method is smart: It accesses component properties and attributes. For example, if you call the map's `get` method with an attribute whose name is `"value"`, the `getValue` method is called. If the attribute name is `"minimum"`, and there is no `getMinimum` method, the component's attribute map is queried for the entry with key `"minimum"`.

You call the helper method in the `setProperties` method, like this:

```
public void setProperties(UIComponent component) {
   super.setProperties(component);
   setInteger(component, "minimum", minimum);
   setInteger(component, "maximum", maximum);
   ...
}
```

Our helper method assumes that the expression evaluates to an integer. You would need other helper methods `setString`, `setBoolean`, and so on, for other types.

Attributes that define method bindings—such as the four commonly used attributes in Table 9–1—require additional work. 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:

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

Table 9–1 Method Binding Attributes

Attribute Name	Method Parameters	Method for Setting the Binding
`valueChangeListener`	`ValueChangeEvent`	`EditableValueHolder.setValue-ChangeListener`
`validator`	`FacesContext, UIComponent, Object`	`EditableValueHolder.setValidator`
`actionListener`	`ActionEvent`	`ActionSource.setActionListener`
`action`	*none*	`ActionSource.setAction`

Action listeners and validators follow exactly the same pattern. However, actions are slightly more complex. An action can be either 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. If the action is a fixed string, you must construct a `MethodBinding` object whose `getExpressionString` method returns that string:

```
if (UIComponentTag.isValueReference(attributeValue))
   ((ActionSource) component).setMethodBinding(component, "action", attributeValue,
         new Class[] {});
else {
   FacesContext context = FacesContext.getCurrentInstance();
   Application app = context.getApplication();
```

```
      MethodBinding mb = new ActionMethodBinding(attributeValue);
      component.getAttributes().put("action", mb);
   }
```

Here, ActionMethodBinding is the following class (which you must supply in your code):

```
   public class ActionMethodBinding extends MethodBinding implements Serializable {
      private String result;

      public ActionMethodBinding(String result) { this.result = result; }
      public Object invoke(FacesContext context, Object params[]) { return result; }
      public String getExpressionString() { return result; }
      public Class getType(FacesContext context) { return String.class; }
   }
```

Handling method expressions is much simpler in JSF 1.2 (see "Supporting Method Expressions" on page 396 for details).

javax.faces.context.FacesContext

- static FacesContext getCurrentInstance()
  Returns a reference to the current FacesContext instance.
- Application getApplication()
  Returns the Application object associated with this web application.

NOTE: The following methods are all deprecated. You should only use them to implement components that are backward compatible with JSF 1.1.

javax.faces.application.Application

- ValueBinding createValueBinding(String valueBindingExpression)
  Creates a value binding and stores it in the application. The string must be a value binding expression of the form #{...}.
- MethodBinding createMethodBinding(String methodBindingExpression, Class[] arguments)
  Creates a method binding and stores it in the application. The methodBinding-Expression must be a method binding expression. The Class[] represents the types of the arguments passed to the method.

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 "}".

javax.faces.component.EditableValueHolder

- `void setValueChangeListener(MethodBinding m)`
  Sets a method binding for the value change listener of this component. The method must return void and is passed a `ValueChangeEvent`.
- `void setValidator(MethodBinding m)`
  Sets a method binding for the validator of this component. The method must return void and is passed a `ValueChangeEvent`.

javax.faces.component.ActionSource

- `void setActionListener(MethodBinding m)`
  Sets a method binding for the action change listener of this component. The method must return void and is passed an `ActionEvent`.
- `void setAction(MethodBinding m)`
  Sets a method binding for the action of this component. The method can return an object of any type, and it has no parameters.

Revisiting the Spinner

Next, we revisit the spinner listed in the previous section. That spinner has two serious drawbacks. First, the spinner component renders itself, so you could not, 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 the increment or decrement button. Nobody would implement an industrial-strength spinner with those deficiencies. Now we 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 404 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:

1. Define an ID string for your renderer.
2. Declare the renderer in a JSF configuration file.
3. Modify your tag class to return the renderer's ID from the `getRendererType` method.
4. 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 did not work so well. The renderer for `h:inputText` writes an HTML `input` text field. That renderer will not work for `h:outputText`—you do not 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 9–2 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`.

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.

Table 9–2 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
UIMessages	javax.faces.Messages
UIOutput	javax.faces.Output
UIPanel	javax.faces.Panel
UISelectBoolean	javax.faces.SelectBoolean
UISelectMany	javax.faces.SelectMany
UISelectOne	javax.faces.SelectOne

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 used programmatically, 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 is 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 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 will not call that method and the children will be encoded separately.

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 returns the ID string, unchanged.

If you have a component that renders itself, it is 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 the renderer, respectively.

Listing 9–10 spinner2/src/java/com/corejsf/UISpinner.java

```java
package com.corejsf;

import javax.faces.component.UIInput;
import javax.faces.convert.IntegerConverter;

public class UISpinner extends UIInput {
   public UISpinner() {
      setConverter(new IntegerConverter()); // to convert the submitted value
   }
}
```

Listing 9–11 spinner2/src/java/com/corejsf/SpinnerRenderer.java

```java
package com.corejsf;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIComponent;
import javax.faces.component.EditableValueHolder;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.convert.ConverterException;
import javax.faces.render.Renderer;

public class SpinnerRenderer extends Renderer {
   private static final String MORE = ".more";
   private static final String LESS = ".less";

   public Object getConvertedValue(FacesContext context, UIComponent component,
         Object submittedValue) throws ConverterException {
      return com.corejsf.util.Renderers.getConvertedValue(context, component,
         submittedValue);
   }

   public void encodeBegin(FacesContext context, UIComponent spinner)
         throws IOException {
      ResponseWriter writer = context.getResponseWriter();
      String clientId = spinner.getClientId(context);

      encodeInputField(spinner, writer, clientId);
      encodeDecrementButton(spinner, writer, clientId);
      encodeIncrementButton(spinner, writer, clientId);
   }
```

Listing 9–11 spinner2/src/java/com/corejsf/SpinnerRenderer.java (cont.)

```

   public void decode(FacesContext context, UIComponent component) {
      EditableValueHolder spinner = (EditableValueHolder) component;
      Map<String, String> requestMap
         = context.getExternalContext().getRequestParameterMap();
      String clientId = component.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(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 encodeInputField(UIComponent spinner, ResponseWriter writer,
         String clientId) throws IOException {
      writer.startElement("input", spinner);
      writer.writeAttribute("name", clientId, "clientId");

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

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

      writer.endElement("input");
   }

```

Listing 9–11 spinner2/src/java/com/corejsf/SpinnerRenderer.java (cont.)

```
   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;
   }
}
```

Calling Converters from External Renderers

If you compare Listing 9–10 and Listing 9–11 with Listing 9–1, you will 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 is 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 398).

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 the protected `UIInput.getConvertedValue` method, which looks like this in the JSF 1.2 Reference Implementation:

```
// This code is from the javax.faces.component.UIInput class:
public void getConvertedValue(FacesContext context, Object newSubmittedValue)
      throws ConverterException {
   Object newValue = newSubmittedValue;
   if (renderer != null) {
      newValue = renderer.getConvertedValue(context, this, newSubmittedValue);
   } else if (newSubmittedValue instanceof String) {
      Converter converter = getConverterWithType(context); // a private method
      if (converter != null)
        newValue = converter.getAsObject(
           context, this, (String) newSubmittedValue);
   }
   return newValue;
}
```

The private `getConverterWithType` method looks up the appropriate converter for the component value.

Because `UIInput`'s conversion code is buried in protected and private methods, it is not available for a renderer to reuse. Custom components that use converters must duplicate the code—see, for example, the implementation of `com.sun.faces.renderkit.html_basic.HtmlBasicInputRenderer` in the reference implementation. Our `com.corejsf.util.Renderers` class provides the code for use in your own classes.

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>
```

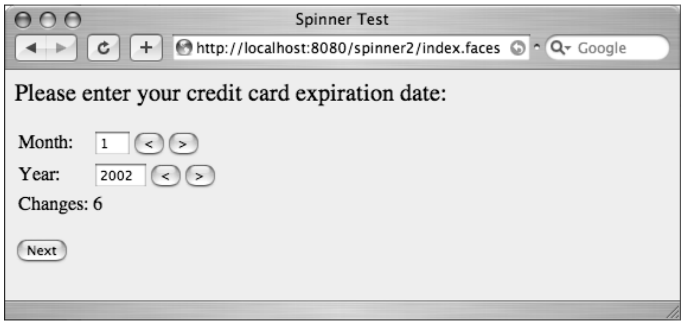
Or you can use a `valueChangeListener` attribute:

```
<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 expression that requires special handling—the topic of the next section, "Supporting Method Expressions."

In the sample program, we demonstrate the value change listener by keeping a count of all value changes that we display on the form (see Figure 9–7).

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



Please enter your credit card expiration date:

Month: 1

Year: 2002

Changes: 6

Figure 9–7 Counting the value changes

Supporting Method Expressions

Four commonly used attributes require method expressions (see Table 9–3). You declare them in the TLD file with `deferred-method` elements, such as the following:

```
<attribute>
   <name>valueChangeListener</name>
   <deferred-method>
      <method-signature>
         void valueChange(javax.faces.event.ValueChangeEvent)
      </method-signature>
   </deferred-method>
</attribute>
```

In the tag handler class, you provide setters for `MethodExpression` objects.

```
public class SpinnerTag extends UIComponentELTag {
   ...
   private MethodExpression valueChangeListener = null;

   public void setValueChangeListener(MethodExpression newValue)  {
      valueChangeListener = newValue;
   }
   ...
}
```

Table 9–3 Processing Method Expressions

Attribute Name	`method-signature` Element in TLD	Code in `setProperties` Method
`valueChangeListener`	`void valueChange(javax.faces.event.ValueChangeEvent)`	`((EditableValueHolder) component) .addValueChangeListener(new MethodExpressionValueChangeListener(expr));`
`validator`	`void validate(javax.faces.context.FacesContext, javax.faces.component.UIComponent, java.lang.Object)`	`((EditableValueHolder) component) .addValidator(new MethodExpressionValidator(expr));`
`actionListener`	`void actionListener(javax.faces.event.ActionEvent)`	`((ActionSource) component) .addActionListener(new MethodExpressionActionListener(expr));`
`action`	`java.lang.Object action()`	`((ActionSource2) component). addAction(expr);`

In the `setProperties` method of the tag handler, you convert the `MethodExpression` object to an appropriate listener object and add it to the component:

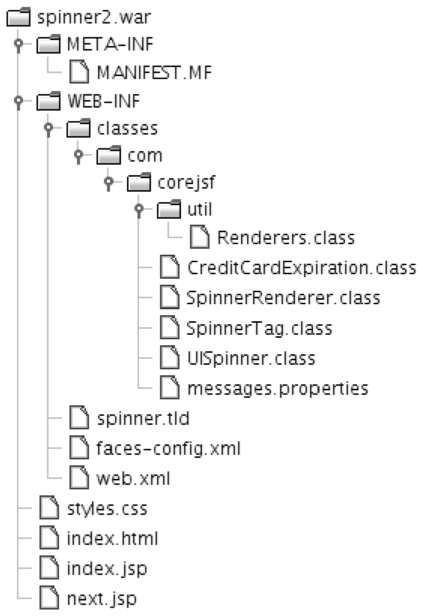
```
public void setProperties(UIComponent component) {
   super.setProperties(component);
   ...
   if (valueChangeListener != null)
      ((EditableValueHolder) component).addValueChangeListener(
            new MethodExpressionValueChangeListener(valueChangeListener));
}
```

Table 9–3 shows how to handle the other method attributes.

NOTE: The `action` attribute value can be either a method expression or a constant. In the latter case, a method is created that always returns the constant value.

The Sample Application

Figure 9–8 shows the directory structure of the sample application. As in the first example, we rely on the core JSF `Renderers` convenience class that contains the code for invoking the converter.



```
spinner2.war
├ META-INF
│  └ MANIFEST.MF
├ WEB-INF
│  ├ classes
│  │  ├ com
│  │  │  ├ corejsf
│  │  │  │  ├ util
│  │  │  │  │  └ Renderers.class
│  │  │  │  ├ CreditCardExpiration.class
│  │  │  │  ├ SpinnerRenderer.class
│  │  │  │  ├ SpinnerTag.class
│  │  │  │  ├ UISpinner.class
│  │  │  │  └ messages.properties
│  ├ spinner.tld
│  ├ faces-config.xml
│  └ web.xml
├ styles.css
├ index.html
├ index.jsp
└ next.jsp
```

Figure 9–8 Directory structure of the revisited spinner example

(The `Renderers` class also contains a `getSelectedItems` method that we need later in this chapter—ignore it for now.) Listing 9–13 contains the revised `SpinnerTag` class, and Listing 9–14 shows the `faces-config.xml` file.

Listing 9–12 spinner2/src/java/com/corejsf/util/Renderers.java

```java
package com.corejsf.util;

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

import javax.el.ValueExpression;
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.model.SelectItem;

public class Renderers {
   public static Object getConvertedValue(FacesContext context,
         UIComponent component, Object submittedValue)
         throws ConverterException {
      if (submittedValue instanceof String) {
         Converter converter = getConverter(context, component);
         if (converter != null) {
            return converter.getAsObject(context, component,
                  (String) submittedValue);
         }
      }
      return submittedValue;
   }

   public static Converter getConverter(FacesContext context,
         UIComponent component) {
      if (!(component instanceof ValueHolder)) return null;
      ValueHolder holder = (ValueHolder) component;

```

Listing 9–12 spinner2/src/java/com/corejsf/util/Renderers.java (cont.)

```
      Converter converter = holder.getConverter();
      if (converter != null)
         return converter;

      ValueExpression expr = component.getValueExpression("value");
      if (expr == null) return null;

      Class targetType = expr.getType(context.getELContext());
      if (targetType == null) return null;
      // Version 1.0 of the reference implementation will not apply a converter
      // if the target type is String or Object, but that is a bug.

      Application app = context.getApplication();
      return app.createConverter(targetType);
   }

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

   @SuppressWarnings("unchecked")
   public static List<SelectItem> getSelectItems(UIComponent component) {
      ArrayList<SelectItem> list = new ArrayList<SelectItem>();
      for (UIComponent child : component.getChildren()) {
         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((SelectItem) value);
            }
         } else if (child instanceof UISelectItems) {
            Object value = ((UISelectItems) child).getValue();
            if (value instanceof SelectItem)
               list.add((SelectItem) value);
            else if (value instanceof SelectItem[])
               list.addAll(Arrays.asList((SelectItem[]) value));
```

Listing 9–12 spinner2/src/java/com/corejsf/util/Renderers.java (cont.)

```
            else if (value instanceof Collection)
               list.addAll((Collection<SelectItem>) value); // unavoidable
            // warning
            else if (value instanceof Map) {
               for (Map.Entry<?, ?> entry : ((Map<?, ?>) value).entrySet())
                  list.add(new SelectItem(entry.getKey(),
                        "" + entry.getValue()));
            }
         }
      }
      return list;
   }
}
```

Listing 9–13 spinner2/src/java/com/corejsf/SpinnerTag.java

```
package com.corejsf;

import javax.el.MethodExpression;
import javax.el.ValueExpression;
import javax.faces.component.EditableValueHolder;
import javax.faces.component.UIComponent;
import javax.faces.event.MethodExpressionValueChangeListener;
import javax.faces.webapp.UIComponentELTag;

public class SpinnerTag extends UIComponentELTag {
   private ValueExpression minimum = null;
   private ValueExpression maximum = null;
   private ValueExpression size = null;
   private ValueExpression value = null;
   private MethodExpression valueChangeListener = null;

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

   public void setMinimum(ValueExpression newValue) { minimum = newValue; }
   public void setMaximum(ValueExpression newValue) { maximum = newValue; }
   public void setSize(ValueExpression newValue) { size = newValue; }
   public void setValue(ValueExpression newValue) { value = newValue; }
   public void setValueChangeListener(MethodExpression newValue)  {
      valueChangeListener = newValue;
   }

```

Listing 9–13 spinner2/src/java/com/corejsf/SpinnerTag.java (cont.)

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

      component.setValueExpression("size", size);
      component.setValueExpression("minimum", minimum);
      component.setValueExpression("maximum", maximum);
      component.setValueExpression("value", value);
      if (valueChangeListener != null)
         ((EditableValueHolder) component).addValueChangeListener(
               new MethodExpressionValueChangeListener(valueChangeListener));
   }

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

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

Listing 9–14 spinner2/web/WEB-INF/faces-config.xml

```
<?xml version="1.0"?>

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">

   <navigation-rule>
      <from-view-id>/index.jsp</from-view-id>
      <navigation-case>
         <from-outcome>next</from-outcome>
         <to-view-id>/next.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

```

Listing 9–14 spinner2/web/WEB-INF/faces-config.xml (cont.)

```
   <navigation-rule>
      <from-view-id>/next.jsp</from-view-id>
      <navigation-case>
         <from-outcome>again</from-outcome>
         <to-view-id>/index.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

   <managed-bean>
      <managed-bean-name>cardExpirationDate</managed-bean-name>
      <managed-bean-class>com.corejsf.CreditCardExpiration</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>

   <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>

   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
   </application>
</faces-config>
```

API *javax.faces.component.EditableValueHolder*

- `void addValueChangeListener(ValueChangeListener listener)` **JSF 1.2**
  Adds a value change listener to this component.
- `void addValidator(Validator val)` **JSF 1.2**
  Adds a validator to this component.

javax.faces.component.ActionSource

- `void addActionListener(ActionListener listener)` JSF 1.2

  Adds an action listener to this component.

javax.faces.component.ActionSource2 JSF 1.2

- `void addAction(MethodExpression m)`

  Adds an action to this component. The method has return type `String` and no parameters.

javax.faces.event.MethodExpressionValueChangeListener JSF 1.2

- `MethodExpressionValueChangeListener(MethodExpression m)`

  Constructs a value change listener from a method expression. The method must return void and is passed a `ValueChangeEvent`.

javax.faces.validator.MethodExpressionValidator JSF 1.2

- `MethodExpressionValidator(MethodExpression m)`

  Constructs a validator from a method expression. The method must return void and is passed a `FacesContext`, a `UIComponent`, and an `Object`.

javax.faces.event.MethodExpressionActionListener JSF 1.2

- `MethodExpressionActionListener(MethodExpression m)`

  Constructs an action listener from a method expression. The method must return void and is passed an `ActionEvent`.

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`

- `ValueExpression getValueExpression(String name)` **JSF 1.2**

  Returns the value expression associated with the given name.

`javax.faces.context.FacesContext`

- `ELContext getELContext()` **JSF 1.2**

  Returns the expression language context.

`javax.el.ValueExpression` **JSF 1.2**

- `Class getType(ELContext context)`

  Returns the type of this value expression.

`javax.faces.application.Application`

- `Converter createConverter(Class targetClass)`

  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`, its interfaces, or its 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 is 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.forms['_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.forms['_id1']['_id1:monthSpinner'].min = 0;
  </script>

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

The second array index is 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 will not 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 do not 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 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 9–15.

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 9–15 spinner-js/src/java/com/corejsf/JSSpinnerRenderer.java

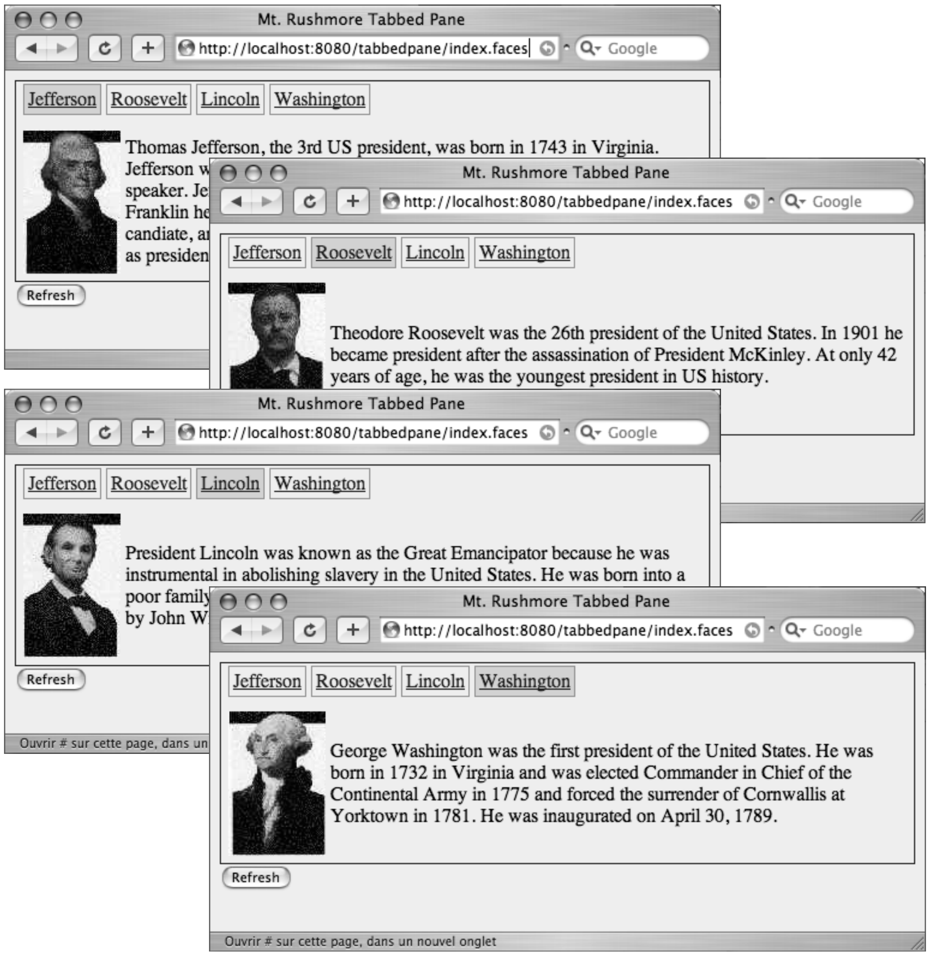
```java
package com.corejsf;

import java.io.IOException;
import java.text.MessageFormat;
import java.util.Map;
import javax.faces.component.EditableValueHolder;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.convert.ConverterException;
import javax.faces.render.Renderer;

public class JSSpinnerRenderer extends Renderer {
   public Object getConvertedValue(FacesContext context, UIComponent component,
         Object submittedValue) throws ConverterException {
      return com.corejsf.util.Renderers.getConvertedValue(context, component,
         submittedValue);
   }

   public void encodeBegin(FacesContext context, UIComponent component)
         throws IOException {
      ResponseWriter writer = context.getResponseWriter();
      String clientId = component.getClientId(context);
      String formId = com.corejsf.util.Renderers.getFormId(context, component);

      UIInput spinner = (UIInput)component;
      Integer min = (Integer) component.getAttributes().get("minimum");
      Integer max = (Integer) component.getAttributes().get("maximum");
      Integer size = (Integer) component.getAttributes().get("size");

      writer.startElement("input", spinner);
      writer.writeAttribute("type", "text", null);
      writer.writeAttribute("name", clientId , null);
      writer.writeAttribute("value", spinner.getValue().toString(), "value");
      if (size != null)
          writer.writeAttribute("size", size , null);
      writer.endElement("input");

      writer.write(MessageFormat.format(
         "<script language=\"JavaScript\">"
         + "document.forms[''{0}''][''{1}''].spin = function (increment) '{'"
         + "var v = parseInt(this.value) + increment;"
         + "if (isNaN(v)) return;"
         + "if (\"min\" in this && v < this.min) return;"
```

Listing 9–15 spinner-js/src/java/com/corejsf/JSSpinnerRenderer.java (cont.)

```
         + "if (\"max\" in this && v > this.max) return;"
         + "this.value = v;"
         + "};",
         new Object[] { formId, clientId } ));

      if (min != null) {
         writer.write(MessageFormat.format(
            "document.forms[''{0}''][''{1}''].min = {2};",
            new Object[] { formId, clientId, min }));
      }
      if (max != null) {
         writer.write(MessageFormat.format(
            "document.forms[''{0}''][''{1}''].max = {2};",
            new Object[] { formId, clientId, max }));
      }
      writer.write("</script>");

      writer.startElement("input", spinner);
      writer.writeAttribute("type", "button", null);
      writer.writeAttribute("value", "<", null);
      writer.writeAttribute("onclick",
         MessageFormat.format(
            "document.forms[''{0}''][''{1}''].spin(-1);",
            new Object[] { formId, clientId }),
            null);
      writer.endElement("input");

      writer.startElement("input", spinner);
      writer.writeAttribute("type", "button", null);
      writer.writeAttribute("value", ">", null);
      writer.writeAttribute("onclick",
         MessageFormat.format(
            "document.forms[''{0}''][''{1}''].spin(1);",
            new Object[] { formId, clientId }),
            null);
      writer.endElement("input");
   }

   public void decode(FacesContext context, UIComponent component) {
      EditableValueHolder spinner = (EditableValueHolder) component;
      Map<String, String> requestMap
         = context.getExternalContext().getRequestParameterMap();
      String clientId = component.getClientId(context);
      spinner.setSubmittedValue((String) requestMap.get(clientId));
      spinner.setValid(true);
   }
}
```

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 9–9.


Mt. Rushmore Tabbed Pane
http://localhost:8080/tabbedpane/index.faces
Google
Jefferson
Roosevelt
Lincoln
Washington
Refresh
Ouvrir # sur cette page, dans un nouvel onglet

Thomas Jefferson, the 3rd US president, was born in 1743 in Virginia.
Jefferson w
speaker. Je
Franklin he
candiate, a
as presiden

Theodore Roosevelt was the 26th president of the United States. In 1901 he became president after the assassination of President McKinley. At only 42 years of age, he was the youngest president in US history.

President Lincoln was known as the Great Emancipator because he was instrumental in abolishing slavery in the United States. He was born into a
poor family
by John W

George Washington was the first president of the United States. He was born in 1732 in Virginia and was elected Commander in Chief of the Continental Army in 1775 and forced the surrender of Cornwallis at Yorktown in 1781. He was inaugurated on April 30, 1789.

Figure 9–9 The tabbed pane component

The tabbed pane component differs from the tabbed pane implementation in Chapter 7 in an essential way. The implementation in Chapter 7 was ad hoc, composed of standard JSF tags such as `h:graphicImage` and `h:commandLink`. We will now develop a reusable component that page authors can simply drop into their pages.

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`), the way the standard JSF menu and listbox tags specify menu or listbox items.
- You *specify tabbed pane content with a facet* (which the renderer renders). For example, you could specify the content for the "Washington" tab in Figure 9–9 as `washington`. Then 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.
- 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.

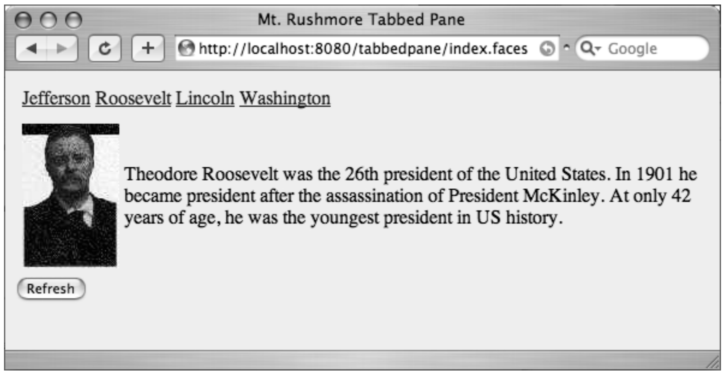
Because the tabbed pane has so many features, there are several ways in which you can use it. Here is a simple use:

```
<corejsf:tabbedPane >
   <f:selectItem itemLabel="Jefferson"  itemValue="jefferson"/>
   <f:selectItem itemLabel="Roosevelt"  itemValue="roosevelt"/>
   <f:selectItem itemLabel="Lincoln"    itemValue="lincoln"/>
   <f:selectItem itemLabel="Washington" itemValue="washington"/>
   <f:facet name="jefferson">
      <h:panelGrid columns="2">
         <h:graphicImage value="/images/jefferson.jpg"/>
         <h:outputText value="#{msgs.jeffersonDiscussion}"/>
      </h:panelGrid>
   </f:facet>
   <!-- three more facets -->
   ...
</corejsf:tabbedPane>
```

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

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

```
<corejsf:tabbedPane styleClass="tabbedPane"
      tabClass="tab" selectedTabClass="selectedTab">
```


Mt. Rushmore Tabbed Pane
http://localhost:8080/tabbedpane/index.faces
Google
Jefferson Roosevelt Lincoln Washington
Refresh

Theodore Roosevelt was the 26th president of the United States. In 1901 he became president after the assassination of President McKinley. At only 42 years of age, he was the youngest president in US history.

Figure 9–10 A plain tabbed pane

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="#{myBean.tabs}"/>
  ...
</corejsf:tabbedPane>
```

Here, the tabs are defined inside a bean.

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="jeffersonTabText"  itemValue="jefferson"/>
  <f:selectItem itemLabel="rooseveltTabText"  itemValue="roosevelt"/>
  <f:selectItem itemLabel="lincolnTabText"    itemValue="lincoln"/>
  <f:selectItem itemLabel="washingtonTabText" itemValue="washington"/>
  ...
</corejsf:tabbedPane>
```

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

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

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, we take a closer look at how it implements advanced features. Here is what we cover in this section:

- "Processing `SelectItem` Children" on page 411
- "Processing Facets" on page 412
- "Encoding CSS Styles" on page 413
- "Using Hidden Fields" on page 415
- "Saving and Restoring State" on page 415
- "Firing Action Events" on page 418

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. You can find the code for the helper method in Listing 9–12 on page 398.

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 415.

Processing Facets

The tabbed pane uses facet names for the content associated with a particular tag. The `encodeEnd` method is responsible for rendering the selected facet:

```
   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);
            }
         }
      }
   }
   ...
}
```

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.

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 is `true` if the component renders its children; otherwise, `false`. 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`.

Encoding CSS Styles

You can support CSS styles in two steps:

1. Add an attribute to the tag library descriptor.
2. 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>
```

Then we 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 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 9–17 on page 419.

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 JSF implementation saves and restores *all* objects in the current view between requests. This includes components, converters, validators, and event listeners. You need to implement state saving for your custom components.

When your application saves the state on the server, then the view objects are held in memory. However, when the state is saved on the client, then the view objects are encoded and stored in a hidden field, in a very long string that looks like this:

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
   value="rO0ABXNyACBjb20uc3VuLmZhY2VzLnV0aWwuVHJlZVN0cnVjdHVyZRRmG0QclWAgAgAETAAI...
   ...4ANXBwcHBwcHBwcHBwcHBwcHBxAH4ANXEAfgA1cHBwcHQABnN1Ym1pdHVxAH4ALAAAAAA=" />
```

Saving state on the client is required to support users who turn off cookies, and it can improve scalability of a web application. Of course, there is a drawback: Voluminous state information is included in every request and response.

The `UITabbedPane` class has an instance field that stores the facet name 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];
    ...
}
```

Here, we assume that the instance field values are serializable. If they are not, then you need to come up with a serializable representation of the component state. (For more information on Java serialization, see Horstmann and Cornell, 2004, 2005. *Core Java™ 2*, vol. 1, chap. 12.)

Listing 9–16 shows how the `UITabbedPane` class saves and restores its state.

NOTE: You may wonder why the implementors did not simply use the standard Java serialization algorithm. However, Java serialization, while quite general, is not necessarily the most efficient format for encoding component state. The JSF architecture allows implementors of JSF containers to provide more efficient mechanisms.

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 `<h:commandButton value="Test State Saving"/>` to index.jsp.
- Run the application and click a tab.
- Click the "Test State Saving" 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.

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

Then you do not need the UITabbedPane class at all. 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>
```

Listing 9–16 tabbedpane/src/java/com/corejsf/UITabbedPane.java

```
package com.corejsf;

import javax.faces.component.UICommand;
import javax.faces.context.FacesContext;

public class UITabbedPane extends UICommand {
   private String content;

```

Listing 9–16 tabbedpane/src/java/com/corejsf/UITabbedPane.java (cont.)

```
   public String getContent() { return content; }
   public void setContent(String newValue) { content = newValue; }

   // Comment out these two methods to see what happens
   // when a component does not properly save its state.
   public Object saveState(FacesContext context) {
      Object values[] = new Object[3];
      values[0] = super.saveState(context);
      values[1] = content;
      return values;
   }

   public void restoreState(FacesContext context, Object state) {
      Object values[] = (Object[]) state;
      super.restoreState(context, values[0]);
      content = (String) values[1];
   }
}
```

javax.faces.component.StateHolder

- `Object saveState(FacesContext context)`
  Returns a `Serializable` object that saves the state of this object.
- `void restoreState(FacesContext context, Object state)`
  Restores the state of this object from the given state object, which is a copy of an object previously obtained from calling `saveState`.
- `void setTransient(boolean newValue)`
- `boolean isTransient()`
  Set and get the transient property. When this property is set, the state is not saved.

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 9–17. The `TabbedPaneTag` class is as boring as ever, and we do not show it here.

Listing 9–17 tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java

```
package com.corejsf;

import java.io.IOException;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.faces.component.UIComponent;
import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.event.ActionEvent;
import javax.faces.model.SelectItem;
import javax.faces.render.Renderer;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

// Renderer for the UITabbedPane component

public class TabbedPaneRenderer extends Renderer {
   private static Logger logger = Logger.getLogger("com.corejsf.util");

   // By default, getRendersChildren() returns false, so encodeChildren()
   // won't be invoked unless we override getRendersChildren() to return true

   public boolean getRendersChildren() {
      return true;
   }

   // The decode method gets the value of the request parameter whose name
   // is the client Id of the tabbedpane component. The request parameter
   // is encoded as a hidden field by encodeHiddenField, which is called by
```

Listing 9–17 tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java (cont.)

```java
   // encodeEnd. The value for the parameter is set by JavaScript generated
   // by the encodeTab method. It is the name of a facet or a JSP page.

   // The decode method uses the request parameter value to set the
   // tabbedpane component's content attribute.
   // Finally, decode() queues an action event that's fired to registered
   // listeners in the Invoke Application phase of the JSF lifecycle. Action
   // listeners can be specified with the <corejsf:tabbedpane>'s actionListener
   // attribute or with <f:actionListener> tags in the body of the
   // <corejsf:tabbedpane> tag.

   public void decode(FacesContext context, UIComponent component) {
      Map<String, String> 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);
      }

      component.queueEvent(new ActionEvent(component));
   }

   // The encodeBegin method writes the starting <table> HTML element
   // with the CSS class specified by the <corejsf:tabbedpane>'s styleClass
   // attribute (if supplied)

   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, null);

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

   // encodeChildren() is invoked by the JSF implementation after encodeBegin().
   // The children of the <corejsf:tabbedpane> component are UISelectItem
   // components, set with one or more <f:selectItem> tags or a single
   // <f:selectItems> tag in the body of <corejsf:tabbedpane>
```

Listing 9–17 tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java (cont.)

```

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;
   }

   ResponseWriter writer = context.getResponseWriter();
   writer.startElement("thead", component);
   writer.startElement("tr", component);
   writer.startElement("th", component);

   writer.startElement("table", component);
   writer.startElement("tbody", component);
   writer.startElement("tr", component);

   for (SelectItem item : com.corejsf.util.Renderers.getSelectItems(component))
      encodeTab(context, writer, item, component);

   writer.endElement("tr");
   writer.endElement("tbody");
   writer.endElement("table");

   writer.endElement("th");
   writer.endElement("tr");
   writer.endElement("thead");
   writer.write("\n"); // to make generated HTML easier to read
}

// encodeEnd() is invoked by the JSF implementation after encodeChildren().
// encodeEnd() writes the table body and encodes the tabbedpane's content
// in a single table row.

// The content for the tabbed pane can be specified as either a URL for
// a JSP page or a facet name, so encodeEnd() checks to see if it's a facet;
// if so, it encodes it; if not, it includes the JSP page

public void encodeEnd(FacesContext context, UIComponent component)
      throws java.io.IOException {
   ResponseWriter writer = context.getResponseWriter();
   UITabbedPane tabbedPane = (UITabbedPane) component;
   String content = tabbedPane.getContent();
```

Listing 9–17 tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java (cont.)

```java

      writer.startElement("tbody", component);
      writer.startElement("tr", component);
      writer.startElement("td", component);

      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);
      }

      writer.endElement("td");
      writer.endElement("tr");
      writer.endElement("tbody");

      // Close off the column, row, and table elements
      writer.endElement("table");

      encodeHiddenField(context, writer, component);
   }

   // The encodeHiddenField method is called at the end of encodeEnd().
   // See the decode method for an explanation of the field and its value.

   private void encodeHiddenField(FacesContext context, ResponseWriter writer,
         UIComponent component) throws java.io.IOException {
      // write hidden field whose name is the tabbedpane's client Id
      writer.startElement("input", component);
      writer.writeAttribute("type", "hidden", null);
      writer.writeAttribute("name", component.getClientId(context), null);
      writer.endElement("input");
   }

   // encodeTab, which is called by encodeChildren, encodes an HTML anchor
   // element with an onclick attribute which sets the value of the hidden
   // field encoded by encodeHiddenField and submits the tabbedpane's enclosing
   // form. See the decode method for more information about the hidden field.
```

custom-components-VM.fm Page 423 Monday, April 2, 2007 12:10 PM

Listing 9–17 tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java (cont.)

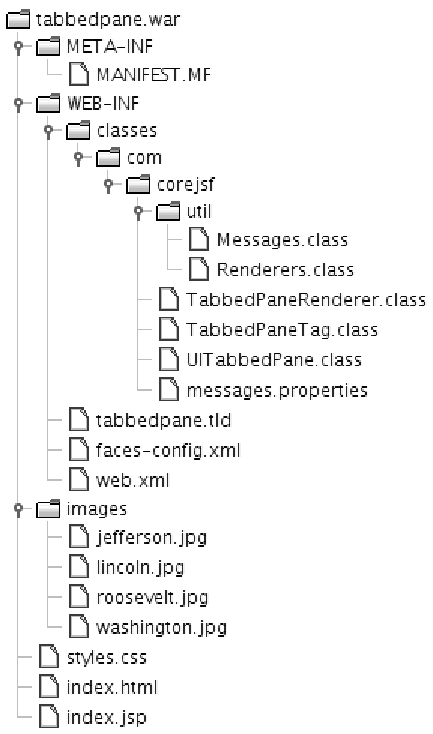
```
   // encodeTab also writes out a class attribute for each tab corresponding
   // to either the tabClass attribute (for unselected tabs) or the
   // selectedTabClass attribute (for the selected tab).

   private void encodeTab(FacesContext context, ResponseWriter writer,
         SelectItem item, UIComponent component) throws java.io.IOException {
      String tabText = getLocalizedTabText(component, item.getLabel());
      String content = (String) item.getValue();

      writer.startElement("td", component);
      writer.startElement("a", component);
      writer.writeAttribute("href", "#", "href");

      String clientId = component.getClientId(context);
      String formId = com.corejsf.util.Renderers.getFormId(context, component);

      writer.writeAttribute("onclick",
      // write value for hidden field whose name is the tabbedpane's client Id

            "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
   }

```

Listing 9–17 tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java (cont.)

```java
   // 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;
   }

   // includePage uses the servlet request dispatcher to include the page
   // corresponding to the selected tab.

   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 (ServletException ex) {
         logger.log(Level.WARNING, "Couldn't load page: " + content, ex);
      } catch (IOException ex) {
         logger.log(Level.WARNING, "Couldn't load page: " + content, ex);
      }
   }
}
```

Using the Tabbed Pane

Figure 9–9 on page 408 shows the `tabbedpane` application. The directory structure for the application is shown in Figure 9–11. Listing 9–18 shows the `index.jsp` page. Listing 9–19 through Listing 9–22 show the tag library descriptor, tag class, faces configuration file, and the style sheet for the tabbed pane application.



```
tabbedpane.war
  META-INF
    MANIFEST.MF
  WEB-INF
    classes
      com
        corejsf
          util
            Messages.class
            Renderers.class
          TabbedPaneRenderer.class
          TabbedPaneTag.class
          UITabbedPane.class
          messages.properties
    tabbedpane.tld
    faces-config.xml
    web.xml
  images
    jefferson.jpg
    lincoln.jpg
    roosevelt.jpg
    washington.jpg
  styles.css
  index.html
  index.jsp
```

Figure 9–11 Directory structure for the tabbed pane example

Listing 9–18 `tabbedpane/web/index.jsp`

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <%@ taglib uri="http://corejsf.com/tabbedpane" prefix="corejsf" %>
   <f:view>
      <head>
         <link href="styles.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="#{msgs.windowTitle}"/></title>
      </head>
      <body>
         <h:form>
```

Listing 9–18 tabbedpane/web/index.jsp (cont.)

```
         <corejsf:tabbedPane styleClass="tabbedPane"
                             tabClass="tab"
                     selectedTabClass="selectedTab">
            <f:facet name="jefferson">
               <h:panelGrid columns="2">
                  <h:graphicImage value="/images/jefferson.jpg"/>
                  <h:outputText value="#{msgs.jeffersonDiscussion}"
                     styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>
            <f:facet name="roosevelt">
               <h:panelGrid columns="2">
                  <h:graphicImage value="/images/roosevelt.jpg"/>
                  <h:outputText value="#{msgs.rooseveltDiscussion}"
                     styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>
            <f:facet name="lincoln">
               <h:panelGrid columns="2">
                  <h:graphicImage value="/images/lincoln.jpg"/>
                  <h:outputText value="#{msgs.lincolnDiscussion}"
                     styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>
            <f:facet name="washington">
               <h:panelGrid columns="2">
                  <h:graphicImage value="/images/washington.jpg"/>
                  <h:outputText value="#{msgs.washingtonDiscussion}"
                     styleClass="tabbedPaneContent"/>
               </h:panelGrid>
            </f:facet>

            <f:selectItem itemLabel="#{msgs.jeffersonTabText}"
               itemValue="jefferson"/>
            <f:selectItem itemLabel="#{msgs.rooseveltTabText}"
               itemValue="roosevelt"/>
            <f:selectItem itemLabel="#{msgs.lincolnTabText}"
               itemValue="lincoln"/>
            <f:selectItem itemLabel="#{msgs.washingtonTabText}"
               itemValue="washington"/>
         </corejsf:tabbedPane>
         <h:commandButton value="Refresh"/>
      </h:form>
   </body>
</f:view>
</html>
```

Listing 9–19 tabbedpane/web/WEB-INF/tabbedpane.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
   version="2.1">
   <description>A library containing a tabbed pane</description>
   <tlib-version>1.1</tlib-version>
   <short-name>tabbedpane</short-name>
   <uri>http://corejsf.com/tabbedpane</uri>
   <tag>
      <description>A tag for a tabbed pane component</description>
      <name>tabbedPane</name>
      <tag-class>com.corejsf.TabbedPaneTag</tag-class>
      <body-content>JSP</body-content>
      <attribute>
         <description>Component id of this component</description>
         <name>id</name>
         <rtexprvalue>true</rtexprvalue>
      </attribute>
      <attribute>
         <description>
            Component reference expression for this component
         </description>
         <name>binding</name>
         <deferred-value>
            <type>javax.faces.component.UIComponent</type>
         </deferred-value>
      </attribute>
      <attribute>
         <description>
            A flag indicating whether or not this component should
            be rendered. If not specified, the default value is true.
         </description>
         <name>rendered</name>
         <deferred-value>
            <type>boolean</type>
         </deferred-value>
      </attribute>
      <attribute>
         <description>The CSS style for this component</description>
         <name>style</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
```

Listing 9–19 tabbedpane/web/WEB-INF/tabbedpane.tld (cont.)

```
      </attribute>
      <attribute>
         <description>The CSS class for this component</description>
         <name>styleClass</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
      <attribute>
         <description>The CSS class for unselected tabs</description>
         <name>tabClass</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
      <attribute>
         <description>The CSS class for the selected tab</description>
         <name>selectedTabClass</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
      <attribute>
         <description>
            The resource bundle used to localize select item labels
         </description>
         <name>resourceBundle</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
      <attribute>
         <description>
            A method expression that's called when a tab is selected
         </description>
         <name>actionListener</name>
         <deferred-method>
            <method-signature>
               void actionListener(javax.faces.event.ActionEvent)
            </method-signature>
         </deferred-method>
      </attribute>
   </tag>
</taglib>
```

Listing 9–20 tabbedpane/src/java/com/corejsf/TabbedPaneTag.java

```java
package com.corejsf;

import javax.el.MethodExpression;
import javax.el.ValueExpression;
import javax.faces.component.ActionSource;
import javax.faces.component.UIComponent;
import javax.faces.event.MethodExpressionActionListener;
import javax.faces.webapp.UIComponentELTag;

// This tag supports the following attributes
//
// binding (supported by UIComponentELTag)
// id (supported by UIComponentELTag)
// rendered (supported by UIComponentELTag)
// style
// styleClass
// tabClass
// selectedTabClass
// resourceBundle
// actionListener

public class TabbedPaneTag extends UIComponentELTag {
   private ValueExpression style;
   private ValueExpression styleClass;
   private ValueExpression tabClass;
   private ValueExpression selectedTabClass;
   private ValueExpression resourceBundle;
   private MethodExpression actionListener;

   public String getRendererType() { return "com.corejsf.TabbedPane"; }
   public String getComponentType() { return "com.corejsf.TabbedPane"; }

   public void setTabClass(ValueExpression newValue) { tabClass = newValue; }
   public void setSelectedTabClass(ValueExpression newValue) {
      selectedTabClass = newValue;
   }
   public void setStyle(ValueExpression newValue) { style = newValue; }
   public void setStyleClass(ValueExpression newValue) {
      styleClass = newValue;
   }
   public void setResourceBundle(ValueExpression newValue) {
      resourceBundle = newValue;
   }
   public void setActionListener(MethodExpression newValue) {
```

Listing 9–20 tabbedpane/src/java/com/corejsf/TabbedPaneTag.java (cont.)

```java
      actionListener = newValue;
   }

   protected void setProperties(UIComponent component) {
      // make sure you always call the superclass
      super.setProperties(component);

      component.setValueExpression("style", style);
      component.setValueExpression("styleClass", styleClass);
      component.setValueExpression("tabClass", tabClass);
      component.setValueExpression("selectedTabClass", selectedTabClass);
      component.setValueExpression("resourceBundle", resourceBundle);
      if (actionListener != null)
         ((ActionSource) component).addActionListener(
               new MethodExpressionActionListener(actionListener));
   }

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

      style = null;
      styleClass = null;
      tabClass = null;
      selectedTabClass = null;
      resourceBundle = null;
      actionListener = null;
   }
}
```

Listing 9–21 tabbedpane/web/WEB-INF/faces-config.xml

```xml
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">
   <navigation-rule>
      <from-view-id>/index.jsp</from-view-id>
         <navigation-case>
            <to-view-id>/welcome.jsp</to-view-id>
         </navigation-case>
   </navigation-rule>
```

Listing 9–21 tabbedpane/web/WEB-INF/faces-config.xml (cont.)

```

   <component>
      <description>A tabbed pane</description>
      <component-type>com.corejsf.TabbedPane</component-type>
      <component-class>com.corejsf.UITabbedPane</component-class>
   </component>

   <render-kit>
      <renderer>
         <component-family>javax.faces.Command</component-family>
         <renderer-type>com.corejsf.TabbedPane</renderer-type>
         <renderer-class>com.corejsf.TabbedPaneRenderer</renderer-class>
      </renderer>
   </render-kit>

   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
   </application>
</faces-config>
```

Listing 9–22 tabbedpane/web/styles.css

```
body {
   background: #eee;
}
.tabbedPane {
   vertical-align: top;
   border: thin solid Blue;
}
.tab {
   padding: 3px;
   border: thin solid CornflowerBlue;
   color: Blue;
}
.selectedTab {
   padding: 3px;
   border: thin solid CornflowerBlue;
   color: Blue;
   background: PowderBlue;
}
```

Implementing Custom Converters and Validators

The custom converters and validators that you saw in Chapter 6 have a shortcoming: They do not allow parameters. For example, we may want to specify a separator character for the credit card converter so that the page designer can choose whether to use dashes or spaces to separate the digit groups. In other words, custom converters should have the same capabilities as the standard `f:convertNumber` and `f:convertDateTime` tags. Specifically, we would like page designers to use tags, such as the following:

```
<h:outputText value="#{payment.card}">
   <corejsf:convertCreditcard separator="-"/>
</h:outputText>
```

To achieve this, we need to implement a custom converter tag. As with custom component tags, custom converter tags require a significant amount of programming, but the payback is a reusable tag that is convenient for page authors.

Custom Converter Tags

As with custom component tags, you need to put descriptions of custom converter tags into a TLD file. Place that file into the `WEB-INF` directory. Listing 9–23 shows the TLD file that describes a `convertCreditcard` custom tag.

Listing 9–23 `custom-converter/web/WEB-INF/converter.tld`

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
   version="2.1">
   <tlib-version>1.1</tlib-version>
   <tlib-version>1.1</tlib-version>
   <short-name>converter</short-name>
   <uri>http://corejsf.com/converter</uri>

   <tag>
      <name>convertCreditcard</name>
      <tag-class>com.corejsf.CreditCardConverterTag</tag-class>
      <body-content>empty</body-content>
      <attribute>
         <name>separator</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
   </tag>
</taglib>
```

The entries in this file should be mostly self-explanatory. The purpose of the file is to specify the class name for the tag handler (`com.corejsf.CreditCardConverterTag`) and the permitted attributes of the tag (in our case, `separator`). Note the `uri` tag that identifies the tag library.

The `deferred-value` child element inside the definition of the `separator` attribute indicates that the attribute is defined by a value expression that should yield a string. The attribute value can be a constant string or a string that contains `#{...}` expressions.

You reference the TLD identifier in a `taglib` directive of the JSF page, such as

```
<%@ taglib uri="http://corejsf.com/converter" prefix="corejsf" %>
```

You need to implement a tag handler class that fulfills three purposes:

1. To specify the converter class
2. To gather the tag attributes
3. To configure a converter object, using the gathered attributes

For a converter, the tag handler class should be a subclass of `ConverterELTag`. As you will see later, the handlers for custom validators need to subclass `ValidatorELTag`.

NOTE: Before JSF 1.2, you needed to subclass `ConverterTag` or `ValidatorTag`. These classes are now deprecated.

Your tag handler class must specify a setter method for each tag attribute. For example,

```
public class ConvertCreditCardTag extends ConverterELTag {
   private ValueExpression separator;
   public void setSeparator(ValueExpression newValue) { separator = newValue; }
   ...
}
```

To configure a converter instance with the tag attributes, override the `create-Converter` method. Construct a converter object and set its properties from the tag attributes. For example,

```
public Converter createConverter() throws JspException {
   CreditCardConverter converter = new CreditCardConverter();
   ELContext elContext = FacesContext.getCurrentInstance().getELContext();
   converter.setSeparator((String) separator.getValue(elContext));
   return converter;
}
```

This method sets the `separator` property of the `CreditCardConverter`.

Finally, you need to define a `release` method for each tag handler class that resets all instance fields to their defaults:

```
public void release() {
   separator = null;
}
```

Listing 9–24 shows the complete tag class.

Listing 9–24 custom-converter/src/java/com/corejsf/CreditCardConverterTag.java

```
package com.corejsf;

import javax.el.ELContext;
import javax.el.ValueExpression;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.webapp.ConverterELTag;
import javax.servlet.jsp.JspException;

public class CreditCardConverterTag extends ConverterELTag {
   private ValueExpression separator;

   public void setSeparator(ValueExpression newValue) {
      separator = newValue;
   }

   public Converter createConverter() throws JspException {
      CreditCardConverter converter = new CreditCardConverter();
      ELContext elContext = FacesContext.getCurrentInstance().getELContext();
      converter.setSeparator((String) separator.getValue(elContext));
      return converter;
   }

   public void release() {
      separator = null;
   }
}
```

API **javax.faces.webapp.ConverterELTag** JSF 1.2

- `protected void createConverter()`
  Override this method to create the converter and customize it by setting the properties specified by the tag attributes.

- `void release()`
  Clears the state of this tag so that it can be reused.

`javax.el.ValueExpression` JSF 1.2

- `Object getValue(ELContext context)`
  Gets the current value of this value expression.

`javax.faces.context.FacesContext` JSF 1.0

- `ELContext getELContext()` **JSF 1.2**
  Gets the context for evaluating expressions in the expression language.

Saving and Restoring State

When implementing converters or validators, you have two choices for state saving. The easy choice is to make your converter or validator class serializable. Implement the `Serializable` interface and follow the usual rules for Java serialization.

In the case of the credit card converter, we have a single instance field of type `String`, which is a serializable type. Therefore, we only need to implement the `Serializable` interface:

```
public class CreditCardConverter implements Converter, Serializable { ... }
```

The second choice is to supply a default constructor and implement the `StateHolder` interface. This is more work for the programmer, but it can yield a slightly more efficient encoding of the object state. Frankly, for small objects such as the credit card converter, this second choice is not worth the extra trouble.

In the interest of completeness, we describe the technique, using the standard `DateTimeConverter` as an example.

In the `saveState` method of the `StateHolder` interface, construct a serializable object that describes the instance fields. The obvious choice is an array of objects that holds the instance fields. In the `restoreState` method, restore the instance fields from that object.

```
public class DateTimeConverter implements Converter, StateHolder {
   public Object saveState(FacesContext context) {
     Object[] values = new Object[6];
     values[0] = dateStyle;
     values[1] = locale;
     values[2] = pattern;
     values[3] = timeStyle;
     values[4] = timeZone;
```

```
         values[5] = type;
         return values;
      }
      public void restoreState(FacesContext context, Object state) {
         Object[] values = (Object[]) state;
         dateStyle = (String) values[0];
         locale = (Locale) values[1];
         pattern = (String) values[2];
         timeStyle = (String) values[3];
         timeZone = (TimeZone) values[4];
         type = (String) values[5];
      }
      ...
   }
```

Moreover, the StateHolder interface also requires you to add a transient property. If the property is set, this particular object will not be saved. The property is the analog of the transient keyword used in Java serialization.

```
   public class DateTimeConverter implements Converter, StateHolder {
      private boolean transientFlag; // "transient" is a reserved word
      public boolean isTransient() { return transientFlag; }
      public void setTransient(boolean newValue) { transientFlag = newValue; }
      ...
   }
```

CAUTION: Converters, validators, and event listeners that implement neither the Serializable nor the StateHolder interface are skipped when the view is saved.

NOTE: Here is an easy experiment to verify that converters must save their state. Configure the custom-converter application to save state on the client by adding this parameter to web.xml:

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

Comment out the Serializable interface of the CreditCardConverter class. To the result.jsp page, add the button

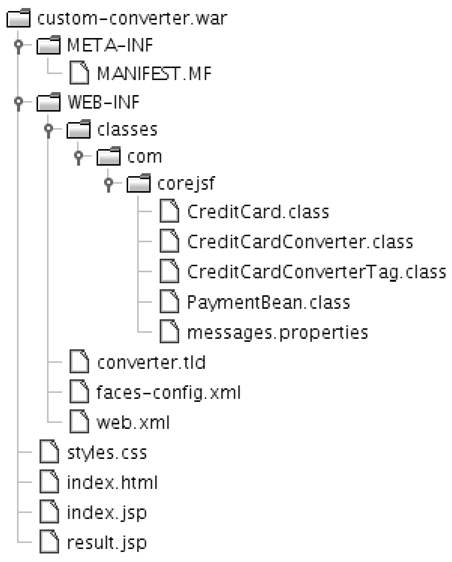
```
<h:commandButton value="Test State Saving"/>
```

Enter a credit card number in index.jsp, click the “Process” button, and see the number formatted with dashes: 4111-1111-1111-1111. Click the “Test State Saving” button and see the dashes disappear.

The Sample Custom Converter Application

This completes the discussion of the custom converter example. Figure 9–12 shows the directory structure. Most files are unchanged from the preceding example. However, `result.jsp` calls the custom converter (see Listing 9–25).

The tag handler is in Listing 9–25. The modified converter and configuration file are in Listings 9–26 and 9–27.



```
custom-converter.war
├ META-INF
│ └ MANIFEST.MF
├ WEB-INF
│ ├ classes
│ │ └ com
│ │   └ corejsf
│ │     ├ CreditCard.class
│ │     ├ CreditCardConverter.class
│ │     ├ CreditCardConverterTag.class
│ │     ├ PaymentBean.class
│ │     └ messages.properties
│ ├ converter.tld
│ ├ faces-config.xml
│ └ web.xml
├ styles.css
├ index.html
├ index.jsp
└ result.jsp
```

Figure 9–12   Directory structure of the custom converter program

Listing 9–25 `custom-converter/web/result.jsp`

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <%@ taglib uri="http://corejsf.com/converter" prefix="corejsf" %>
   <f:view>
      <head>
         <link href="styles.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="#{msgs.title}"/></title>
      </head>
      <body>
         <h:form>
            <h1><h:outputText value="#{msgs.paymentInformation}"/></h1>
```

Listing 9–25 custom-converter/web/result.jsp (cont.)

```
            <h:panelGrid columns="2">
               <h:outputText value="#{msgs.amount}"/>
               <h:outputText value="#{payment.amount}">
                  <f:convertNumber type="currency"/>
               </h:outputText>

               <h:outputText value="#{msgs.creditCard}"/>
               <h:outputText value="#{payment.card}">
                  <corejsf:convertCreditcard separator="-"/>
               </h:outputText>

               <h:outputText value="#{msgs.expirationDate}"/>
               <h:outputText value="#{payment.date}">
                  <f:convertDateTime pattern="MM/yyyy"/>
               </h:outputText>
            </h:panelGrid>
            <h:commandButton value="#{msgs.back}" action="back"/>
         </h:form>
      </body>
   </f:view>
</html>
```

Listing 9–26 custom-converter/src/java/com/corejsf/CreditCardConverter.java

```
package com.corejsf;

import java.io.Serializable;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;

public class CreditCardConverter implements Converter, Serializable {
   private String separator;

   // PROPERTY: separator
   public void setSeparator(String newValue) { separator = newValue; }

   public Object getAsObject(
      FacesContext context,
      UIComponent component,
      String newValue)
```

Listing 9–26 custom-converter/src/java/com/corejsf/CreditCardConverter.java (cont.)

```
      throws ConverterException {
      StringBuilder builder = new StringBuilder(newValue);
      int i = 0;
      while (i < builder.length()) {
         if (Character.isDigit(builder.charAt(i)))
            i++;
         else
            builder.deleteCharAt(i);
      }
      return new CreditCard(builder.toString());
   }

   public String getAsString(
      FacesContext context,
      UIComponent component,
      Object value)
      throws ConverterException {
      // length 13: xxxx xxx xxx xxx
      // length 14: xxxxx xxxx xxxxx
      // length 15: xxxx xxxxxx xxxxx
      // length 16: xxxx xxxx xxxx xxxx
      // length 22: xxxxxx xxxxxxxx xxxxxxxx
      if (!(value instanceof CreditCard))
         throw new ConverterException();
      String v = ((CreditCard) value).toString();
      String sep = separator;
      if (sep == null) sep = " ";
      int[] boundaries = null;
      int length = v.length();
      if (length == 13)
         boundaries = new int[] { 4, 7, 10 };
      else if (length == 14)
         boundaries = new int[] { 5, 9 };
      else if (length == 15)
         boundaries = new int[] { 4, 10 };
      else if (length == 16)
         boundaries = new int[] { 4, 8, 12 };
      else if (length == 22)
         boundaries = new int[] { 6, 14 };
      else
         return v;
      StringBuilder result = new StringBuilder();
      int start = 0;
```

Listing 9–26 custom-converter/src/java/com/corejsf/CreditCardConverter.java (cont.)

```java
      for (int i = 0; i < boundaries.length; i++) {
         int end = boundaries[i];
         result.append(v.substring(start, end));
         result.append(sep);
         start = end;
      }
      result.append(v.substring(start));
      return result.toString();
   }
}
```

Listing 9–27 custom-converter/web/WEB-INF/faces-config.xml

```xml
<?xml version="1.0"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">
   <navigation-rule>
      <from-view-id>/index.jsp</from-view-id>
      <navigation-case>
         <from-outcome>process</from-outcome>
         <to-view-id>/result.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

   <navigation-rule>
      <from-view-id>/result.jsp</from-view-id>
      <navigation-case>
         <from-outcome>back</from-outcome>
         <to-view-id>/index.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

   <converter>
      <converter-id>com.corejsf.CreditCard</converter-id>
      <converter-class>com.corejsf.CreditCardConverter</converter-class>
   </converter>

```

Listing 9–27 custom-converter/web/WEB-INF/faces-config.xml (cont.)

```
   <converter>
      <converter-for-class>com.corejsf.CreditCard</converter-for-class>
      <converter-class>com.corejsf.CreditCardConverter</converter-class>
   </converter>

   <managed-bean>
      <managed-bean-name>payment</managed-bean-name>
      <managed-bean-class>com.corejsf.PaymentBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>

   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
   </application>
</faces-config>
```

Custom Validator Tags

In the preceding sections, you saw how to implement a custom converter that offers page authors the same convenience as the standard JSF tags. In this section, you will see how to provide a custom validator.

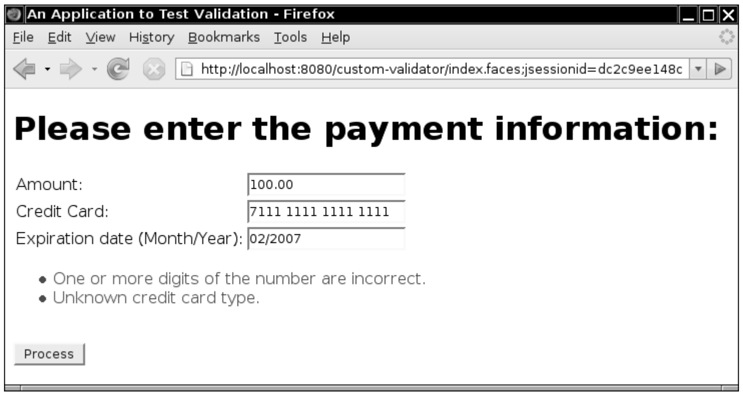
The steps for providing a custom validator are almost the same as those for a custom converter:

1. Produce a TLD file and reference it in your JSF pages.
2. Implement a tag handler class that extends the `ValidatorELTag` class, gathers the attributes that the TLD file advertises, and passes them to a validator object.
3. Implement a validator class that implements the `Validator` interface. Supply the `validate` method in the usual way, by throwing a `ValidatorException` if an error is detected. Implement the `Serializable` or `StateHolder` interface to save and restore the state of validator objects.

As an example, let us do a thorough job validating credit card numbers (see Listing 9–27). We want to carry out three checks:

1. The user has supplied a value.
2. The number conforms to the Luhn formula.
3. The number starts with a valid prefix.

custom-components-VM.fm Page 442 Monday, April 2, 2007 12:10 PM


An Application to Test Validation - Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/custom-validator/index.faces;jsessionid=dc2c9ee148c

Please enter the payment information:

Amount: 100.00
Credit Card: 7111 1111 1111 1111
Expiration date (Month/Year): 02/2007

- One or more digits of the number are incorrect.
- Unknown credit card type.

Figure 9–13 Thoroughly validating a credit card number

A credit card's prefix indicates card type—for example, a prefix between 51 and 55 is reserved for MasterCard, and a prefix of 4 indicates Visa. We could write custom code for this purpose, but instead we chose to implement a more general (and more useful) validator that validates arbitrary regular expressions.

We use that validator in the following way:

```
<corejsf:validateRegex expression="[3-6].*"
   errorDetail="#{msgs.unknownType}"/>
```

The regular expression `[3-6].*` denotes any string that starts with the digits 3 through 6. Of course, we could easily design a more elaborate regular expression that does a more careful check.

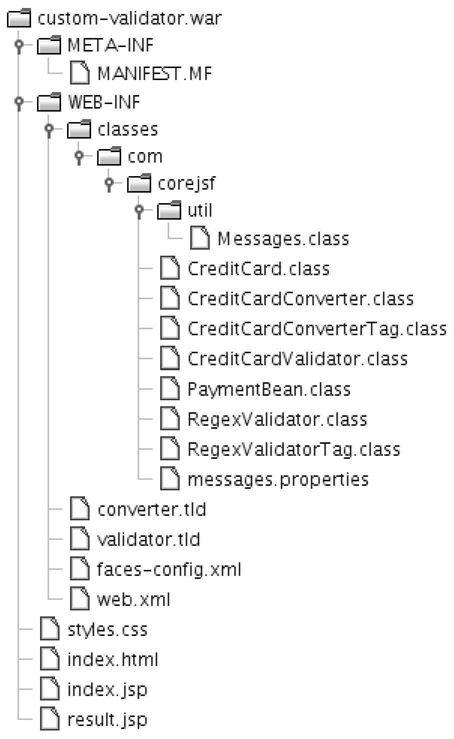
You will find the validator code in Listing 9–28. When reading through the code, keep in mind that the moral of the story here has nothing to do with regular expressions per se. Instead, the story is about what validators do when their component's data is invalid: They generate a faces message, wrap it inside a validator exception, and throw it.

By default, the validator displays an error message that complains about failing to match a regular expression. If your application's audience includes users who are unfamiliar with regular expressions, you will want to change the message. We give you attributes `errorSummmary` and `errorDetail` for this purpose.

We use a custom tag so that we can supply parameters to the validator. Implementing a custom tag for a validator is similar to creating a custom converter tag, which we described earlier in this chapter. However, the custom validator tag must extend the `ValidatorTag` class.

You can find the implementation of the `RegexValidatorTag` class in Listing 9–29.

Figure 9–14 shows the application's directory structure. Listing 9–32 shows the JSF page with the triple validation of the credit card field.



```
custom-validator.war
  META-INF
    MANIFEST.MF
  WEB-INF
    classes
      com
        corejsf
          util
            Messages.class
          CreditCard.class
          CreditCardConverter.class
          CreditCardConverterTag.class
          CreditCardValidator.class
          PaymentBean.class
          RegexValidator.class
          RegexValidatorTag.class
          messages.properties
    converter.tld
    validator.tld
    faces-config.xml
    web.xml
  styles.css
  index.html
  index.jsp
  result.jsp
```

Figure 9–14 Directory structure of the thoroughly validating application

Listing 9–30 shows `faces-config.xml`. Note the mapping of the validator ID to the validator class. The validator tag class is defined in the TLD file (Listing 9–31).

You have now seen how to implement custom tags for components, converters, and validators. We covered all essential issues that you will encounter as you develop custom tags. The code in this chapter should make a good starting point for your own implementations.

API *`javax.faces.webapp.ValidatorTag`*

- `void setValidatorId(String id)`
  Sets the ID of this validator. The ID is used to look up the validator class.

- `protected void createValidator()`
  Override this method to customize the validator by setting the properties specified by the tag attributes.

Listing 9–28 custom-validator/src/java/com/corejsf/RegexValidator.java

```
package com.corejsf;

import java.io.Serializable;
import java.text.MessageFormat;
import java.util.Locale;
import java.util.regex.Pattern;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class RegexValidator implements Validator, Serializable {
   private String expression;
   private Pattern pattern;
   private String errorSummary;
   private String errorDetail;

   public void validate(FacesContext context, UIComponent component,
         Object value) {
      if (value == null) return;
      if (pattern == null) return;
      if(!pattern.matcher(value.toString()).matches()) {
         Object[] params = new Object[] { expression, value };
         Locale locale = context.getViewRoot().getLocale();
         FacesMessage message = com.corejsf.util.Messages.getMessage(
               "com.corejsf.messages", "badRegex", params);
         if (errorSummary != null)
            message.setSummary(
                  new MessageFormat(errorSummary, locale).format(params));
         if (errorDetail != null)
            message.setDetail(
                  new MessageFormat(errorDetail, locale).format(params));
         throw new ValidatorException(message);
      }
   }

   // PROPERTY: expression
   public void setExpression(String newValue) {
      expression = newValue;
      pattern = Pattern.compile(expression);
   }
```

Listing 9–28 custom-validator/src/java/com/corejsf/RegexValidator.java (cont.)

```

   // PROPERTY: errorSummary
   public void setErrorSummary(String newValue) {
      errorSummary = newValue;
   }

   // PROPERTY: errorDetail
   public void setErrorDetail(String newValue) {
      errorDetail = newValue;
   }
}
```

Listing 9–29 custom-validator/src/java/com/corejsf/RegexValidatorTag.java

```
package com.corejsf;

import javax.el.ELContext;
import javax.el.ValueExpression;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.webapp.ValidatorELTag;
import javax.servlet.jsp.JspException;

public class RegexValidatorTag extends ValidatorELTag {
   private ValueExpression expression;
   private ValueExpression errorSummary;
   private ValueExpression errorDetail;

   public void setExpression(ValueExpression newValue) {
      expression = newValue;
   }

   public void setErrorSummary(ValueExpression newValue) {
      errorSummary = newValue;
   }

   public void setErrorDetail(ValueExpression newValue) {
      errorDetail = newValue;
   }

   public Validator createValidator() throws JspException {
      RegexValidator validator = new RegexValidator();
      ELContext elContext = FacesContext.getCurrentInstance().getELContext();

```

Listing 9–29 custom-validator/src/java/com/corejsf/RegexValidatorTag.java (cont.)

```java
      validator.setExpression((String) expression.getValue(elContext));
      if (errorSummary != null)
         validator.setErrorSummary((String) errorSummary.getValue(elContext));
      if (errorDetail != null)
         validator.setErrorDetail((String) errorDetail.getValue(elContext));

      return validator;
   }

   public void release() {
      expression = null;
      errorSummary = null;
      errorDetail = null;
   }
}
```

Listing 9–30 custom-validator/web/WEB-INF/faces-config.xml

```xml
<?xml version="1.0"?>

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
   version="1.2">
   <navigation-rule>
      <from-view-id>/index.jsp</from-view-id>
      <navigation-case>
         <from-outcome>process</from-outcome>
         <to-view-id>/result.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

   <navigation-rule>
      <from-view-id>/result.jsp</from-view-id>
      <navigation-case>
         <from-outcome>back</from-outcome>
         <to-view-id>/index.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

```

Listing 9–30 custom-validator/web/WEB-INF/faces-config.xml (cont.)

```
   <converter>
      <converter-id>com.corejsf.CreditCard</converter-id>
      <converter-class>com.corejsf.CreditCardConverter</converter-class>
   </converter>

   <converter>
      <converter-for-class>com.corejsf.CreditCard</converter-for-class>
      <converter-class>com.corejsf.CreditCardConverter</converter-class>
   </converter>

   <validator>
      <validator-id>com.corejsf.CreditCard</validator-id>
      <validator-class>com.corejsf.CreditCardValidator</validator-class>
   </validator>

   <validator>
      <validator-id>com.corejsf.Regex</validator-id>
      <validator-class>com.corejsf.RegexValidator</validator-class>
   </validator>

   <managed-bean>
      <managed-bean-name>payment</managed-bean-name>
      <managed-bean-class>com.corejsf.PaymentBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>

   <application>
      <resource-bundle>
         <base-name>com.corejsf.messages</base-name>
         <var>msgs</var>
      </resource-bundle>
   </application>
</faces-config>
```

Listing 9–31 custom-validator/web/WEB-INF/validator.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
   version="2.1">
```

Listing 9–31 custom-validator/web/WEB-INF/validator.tld (cont.)

```
   <tlib-version>1.1</tlib-version>
   <short-name>validator</short-name>
   <uri>http://corejsf.com/validator</uri>
   <tag>
      <name>validateRegex</name>
      <tag-class>com.corejsf.RegexValidatorTag</tag-class>
      <body-content>empty</body-content>
      <attribute>
         <name>expression</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
      <attribute>
         <name>errorSummary</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
      <attribute>
         <name>errorDetail</name>
         <deferred-value>
            <type>java.lang.String</type>
         </deferred-value>
      </attribute>
   </tag>
</taglib>
```

Listing 9–32 custom-validator/web/index.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <%@ taglib uri="http://corejsf.com/validator" prefix="corejsf" %>
   <f:view>
      <head>
         <link href="styles.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="#{msgs.title}"/></title>
      </head>
      <body>
         <h:form>
            <h1><h:outputText value="#{msgs.enterPayment}"/></h1>
```

custom-components-VM.fm Page 449 Monday, April 2, 2007 12:10 PM

Listing 9–32 custom-validator/web/index.jsp (cont.)

```
            <h:panelGrid columns="2">
               <h:outputText value="#{msgs.amount}"/>
               <h:inputText id="amount" value="#{payment.amount}">
                  <f:convertNumber minFractionDigits="2"/>
               </h:inputText>

               <h:outputText value="#{msgs.creditCard}"/>
               <h:inputText id="card" value="#{payment.card}" required="true">
                  <f:validator validatorId="com.corejsf.CreditCard"/>
                  <corejsf:validateRegex expression="[3-6].*"
                     errorDetail="#{msgs.unknownType}"/>
               </h:inputText>

               <h:outputText value="#{msgs.expirationDate}"/>
               <h:inputText id="date" value="#{payment.date}">
                  <f:convertDateTime pattern="MM/yyyy"/>
               </h:inputText>
            </h:panelGrid>
            <h:messages styleClass="errorMessage"
               showSummary="false" showDetail="true"/>
            <br/>
            <h:commandButton value="Process" action="process"/>
         </h:form>
      </body>
   </f:view>
</html>
```