

ONLINE EXTENSION

JAVA SERVER FACES IN ACTION



Kito D. Mann

Foreword by Ed Burns



MANNING

Online Extension

*JavaServer Faces
in Action*

KITO D. MANN



MANNING

Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please go to www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
209 Bruce Park Avenue
Greenwich, CT 06830

Fax: (203) 661-9018
email: orders@manning.com

©2005 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ② Recognizing the importance of preserving what has been written, it is Manning's policy to have the books they publish printed on acid-free paper, and we exert our best efforts to that end.

All screens shots of Oracle JDeveloper in appendix B are reproduced with the permission of Oracle Corp. Copyright Oracle Corp, 2004.

All screens shots of WebSphere Studio in appendix B are reproduced with the permission of IBM Corp. Copyright IBM Corp, 2004.

 Manning Publications Co. Copyeditor: Liz Welch
209 Bruce Park Avenue Typesetter: Denis Dalinnik
Greenwich, CT 06830 Cover designer: Leslie Haimes

ISBN 1-932394-11-7

Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – VHG – 08 07 06 05 04

online extension

The online extension consists of five chapters in part 5 as well as four appendixes that are not included in the print edition. They are available for free download from the book's web site at www.manning.com/mann. This is the table of contents for the online extension.

PART 5 WRITING CUSTOM COMPONENTS, RENDERERS, VALIDATORS, AND CONVERTERS: EXAMPLES 703

16

UIInputDate: a simple input component 705

16.1 Writing the UIInputDate class 708

Encoding 709 • *Decoding* 715 • *Implementing StateHolder methods* 717

16.2 Registering the component 718

16.3 JSP integration 718

Writing the JSP custom tag 718 • *Validating the tag* 721
Adding the tag to the tag library 722

16.4 Using the component 724

16.5 Summary 726

17

RolloverButton renderer: a renderer with JavaScript support 727

- 17.1 Writing the RolloverButtonRenderer class 729
 - Encoding* 731 ▪ *Decoding* 735
 - Registering the renderer* 736
- 17.2 JSP Integration 737
 - Writing the HtmlBaseTag class* 738 ▪ *Writing the JSP custom tag* 741 ▪ *Validating the tag* 744 ▪ *Adding the tag to the tag library* 745
- 17.3 Using the renderer 748
- 17.4 Wrapping an existing renderer 750
 - Developing the RolloverButtonDecoratorRenderer class* 750
- 17.5 Summary 754

18

UIHeadlineViewer: a composite, data-aware component 756

- 18.1 RSS and the Informa API 758
- 18.2 Using UIData with Informa 763
- 18.3 Subclassing DataModel 765
- 18.4 Writing the UIHeadlineViewer class 768
- 18.5 Registering the component 780
- 18.6 JSP integration 781
 - Writing the JSP custom tag* 781
 - Adding the tag to the tag library* 787
- 18.7 Using the component 789
- 18.8 Summary 793

19

UINavigator: a model-driven toolbar component 794

- 19.1 Writing the model classes 796
- 19.2 Writing the UINavigator class 801
 - Implementing ActionSource methods* 803 ▪ *Overriding UIComponentBase methods* 806 ▪ *Implementing StateHolder methods* 807 ▪ *Developing NavigatorActionListener: a custom ActionListener* 809

19.3	Registering the component	810
19.4	Writing the ToolbarRenderer class	811
	<i>Encoding</i>	811
	<i>Decoding</i>	820
19.5	Registering the renderer	821
19.6	JSP integration	822
	<i>Writing the Navigator_ToolbarTag component tag</i>	822
	<i>Writing the NavigatorItemTag tag handler</i>	826
	<i>Adding the tags to the tag library</i>	831
19.7	Using the component	834
19.8	Summary	838
20	<i>Validator and converter examples</i>	839
20.1	Validator methods vs. validator classes	840
20.2	Developing a validator	840
	<i>Writing the RegularExpressionValidator class</i>	842
	<i>Registering the validator</i>	847
	<i>Integrating with JSP</i>	847
	<i>Using the validator</i>	852
20.3	When custom converters are necessary	854
20.4	Developing a converter	854
	<i>Writing the UserConverter class</i>	856
	<i>Registering the converter</i>	865
	<i>JSP integration</i>	866
	<i>Using the converter</i>	870
20.5	Summary	872
<i>appendix B:</i>	<i>A survey of JSF IDEs and implementations</i>	873
<i>appendix C:</i>	<i>Extending the core JSF classes</i>	935
<i>appendix D:</i>	<i>JSF configuration</i>	958
<i>appendix E:</i>	<i>Time zone, country, language, and currency codes</i>	976
	<i>references</i>	1011
	<i>index</i>	1015

Part 5

Writing custom components, renderers, validators, and converters: examples

P

art 5 builds upon the concepts covered in part 4 with examples of real-world UI components, renderers, validators, and converters.

UIInputDate: a simple input component

This chapter covers

- Developing a simple date input component
- Implementing the direct rendering model
- Writing a custom tag validator

In this chapter, we'll look at how to build a component that collects a date from the user. We've selected an amazingly unique name for it: `UIInputDate`. Figure 16.1 shows four different instances of the component in a web browser. Each instance shows different parts of the date, and the latter one is synchronized with a session-scoped variable.

NOTE Some portions of code in this chapter have been omitted. You can download the full source code for this book from the *JSF in Action* web site (<http://www.manning.com/mann>).

What we're aiming for with `UIInputDate` is functionality that's similar to the date input controls you see on popular web sites like Expedia or Travelocity. The control uses standard HTML drop-downs without JavaScript, so it's guaranteed to work with most browsers. It can be synchronized with a value-binding expression,

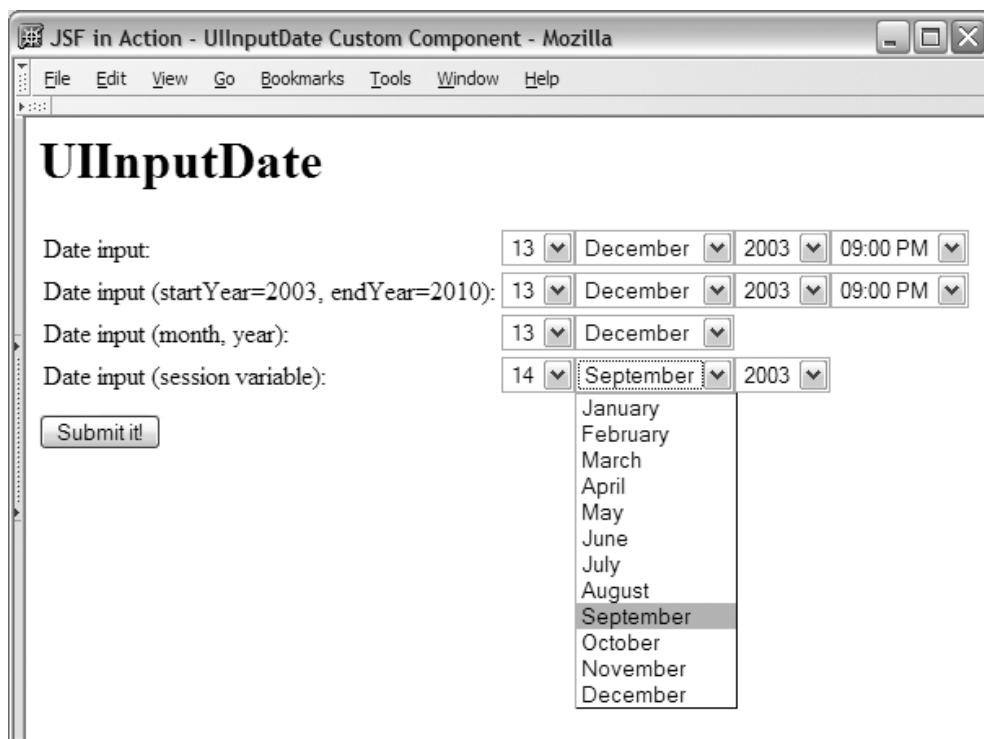


Figure 16.1 The `UIInputDate` component collects a date via drop-down list boxes.

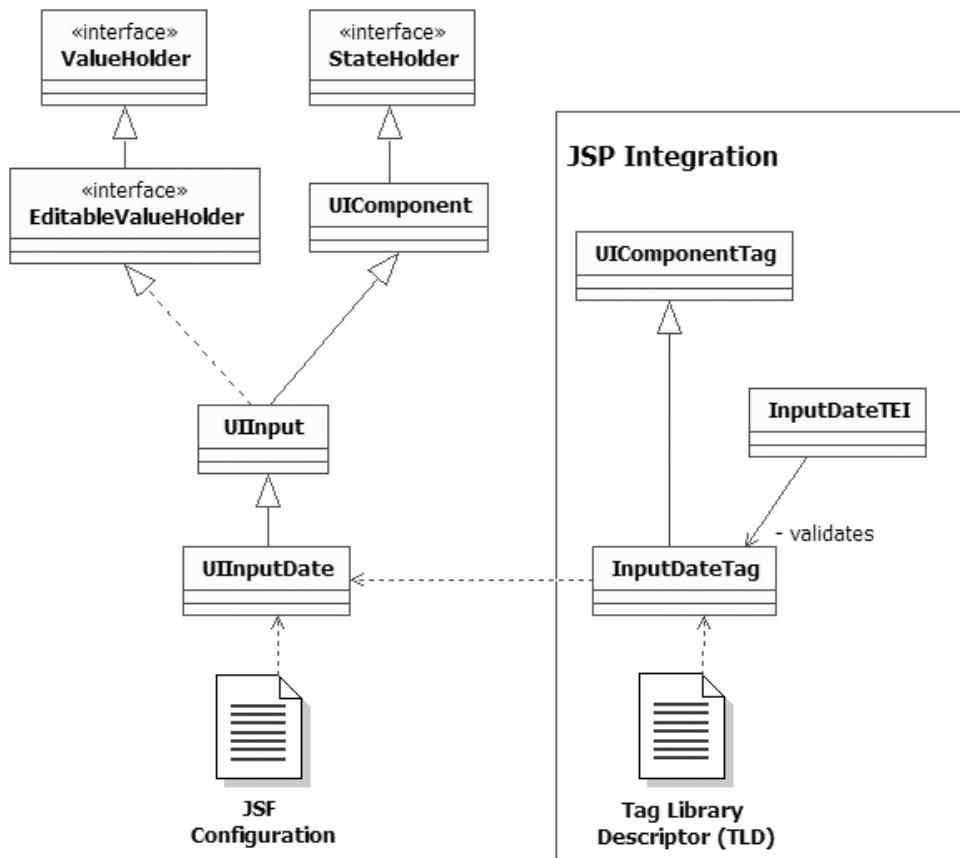


Figure 16.2 Classes and configuration files for the `UIInputDate` component. The component subclasses `UIInput`, which implements the `EditableValueHolder` interface. There is one custom tag handler, `InputDateTag`, which is validated by the `InputDateTEI` class.

and the user can decide which fields (day, month, year, and time) are visible and the range of years shown in the year drop-down list box. Figure 16.2 shows all of the different elements involved in developing this class.

For this component, we'll use the direct implementation rendering model. We'll start by dissecting the `UIInputDate` class and its configuration elements, move on to JSP integration, and then show how to use the component.

16.1 Writing the `UIInputDate` class

Because `UIInputDate` collects user input, we can start by subclassing the `UIInput` class. `UIInput` is a concrete class that has a value, collects user input, and has support for validators and converters. In other words, it's the base implementation of the `EditableValueHolder` interface (and every other input control subclasses it).

A `UIInputDate` is basically a specialized way to input a `Date` object. (It's possible to do this using a standard `UIInput` with a `DateTime` converter, but doing so doesn't automatically enforce the format of the date, nor does it let the user know which dates are valid.) Consequently, the value of the component should be a `Date` object. For convenience to developers, we'll provide a `date` property that is a type-safe alias for the `value` property.

We'd like our component to be able to optionally display different parts of the date—the day, month, year, and time. For each field, we'll provide a `boolean` property that specifies whether to show the field at all. We also need to know the range of values that can be displayed for each field in its drop-down listbox.

We can determine the number of days to show based on the current month. For instance, if the month is currently January, we know there are 31 days. Time also is a no-brainer—there's only 24 hours in the day. Months are easy, too—there are only 12 months in a year. However, it's unclear how many years to show. Should it be just the current year? What if the application developer wants to show this year and the next five years? Or the range 1950 to 1995? To simplify things, we'll provide two additional properties: `startYear` for the first year to show in the drop-down listbox, and `endYear` for the last year to appear. All of these properties are listed in table 16.1.

Table 16.1 The properties for `UIInputDate` to display

Property	Type	Description
<code>date</code>	<code>Date</code>	A type-safe alias for the <code>value</code> property
<code>showDay</code>	<code>boolean</code>	True if the user can edit the day
<code>showMonth</code>	<code>boolean</code>	True if the user can edit the month
<code>showYear</code>	<code>boolean</code>	True if the user can edit the year
<code>showTime</code>	<code>boolean</code>	True if the user can edit the time
<code>startYear</code>	<code>int</code>	The first year to display in the drop-down listbox (if the year is displayed).
<code>endYear</code>	<code>int</code>	The last year to display in the year drop-down listbox (if the year is displayed)

The first step is to declare the component's type and family. For both of these, we'll use "jia.InputDate":

```
public final static String COMPONENT_TYPE = "jia.InputDate";
public final static String COMPONENT_FAMILY = "jia.InputDate";
```

We'll also expose the family constant:

```
public String getFamily()
{
    return COMPONENT_FAMILY;
}
```

The default rendererType for UIInput is Text. In our component, we'd like to handle rendering ourselves, so the rendererType property should be null, which can be handled in the constructor:

```
public UIInputDate()
{
    super();
    setRendererType(null);
}
```

Now that JSF knows we will be handling the rendering duties internally, it's time to get down to business.

16.1.1 Encoding

UIInputDate has no child components, so we can do all of the encoding in encodeBegin. What we're shooting for is one HTML <select> element for each date field, as shown in listing 16.1.

Listing 16.1 Output of UIInputDate (all fields)—this maps to the first UIInputDate shown in figure 16.1

```
<select name="_id0:_id3:day">
<option value="1">01</option>
<option value="2">02</option>
<option value="3">03</option>
<option value="4">04</option>
<option value="5">05</option>
<option value="6">06</option>
<option value="7">07</option>
<option value="8">08</option>
<option value="9">09</option>
<option value="10">10</option>
<option value="11">11</option>
<option value="12">12</option>
<option value="13" selected="true">13</option>
```

```
<option value="14">14</option>
<option value="15">15</option>
<option value="16">16</option>
<option value="17">17</option>
<option value="18">18</option>
<option value="19">19</option>
<option value="20">20</option>
<option value="21">21</option>
<option value="22">22</option>
<option value="23">23</option>
<option value="24">24</option>
<option value="25">25</option>
<option value="26">26</option>
<option value="27">27</option>
<option value="28">28</option>
<option value="29">29</option>
<option value="30">30</option>
<option value="31">31</option>
</select>
<select name="_id0:_id3:month">
    <option value="0">January</option>
    <option value="1">February</option>
    <option value="2">March</option>
    <option value="3">April</option>
    <option value="4">May</option>
    <option value="5">June</option>
    <option value="6">July</option>
    <option value="7">August</option>
    <option value="8">September</option>
    <option value="9">October</option>
    <option value="10">November</option>
    <option value="11" selected="true">December</option>
</select>
<select name="_id0:_id3:year">
    <option value="2003" selected="true">2003</option>
</select>
<select name="_id0:_id3:time">
    <option value="0">12:00 AM</option>
    <option value="1">01:00 AM</option>
    <option value="2">02:00 AM</option>
    <option value="3">03:00 AM</option>
    <option value="4">04:00 AM</option>
    <option value="5">05:00 AM</option>
    <option value="6">06:00 AM</option>
    <option value="7">07:00 AM</option>
    <option value="8">08:00 AM</option>
    <option value="9">09:00 AM</option>
    <option value="10">10:00 AM</option>
    <option value="11">11:00 AM</option>
    <option value="12">12:00 PM</option>
    <option value="13">01:00 PM</option>
```

```
<option value="14">02:00 PM</option>
<option value="15">03:00 PM</option>
<option value="16">04:00 PM</option>
<option value="17">05:00 PM</option>
<option value="18">06:00 PM</option>
<option value="19">07:00 PM</option>
<option value="20">08:00 PM</option>
<option value="21" selected="true">09:00 PM</option>
<option value="22">10:00 PM</option>
<option value="23">11:00 PM</option>
</select>
```

We'll start the method with a couple of checks:

```
public void encodeBegin(FacesContext context)
    throws java.io.IOException
{
    if (!isRendered())
    {
        return;
    }

    if (!showDay && !showMonth && !showYear && !showTime)
    {
        throw new InvalidPropertiesException(
            "All display properties set to false. " +
            "One of the showDay, showMonth, showYear, " +
            "or showTime properties must be set to true.");
    }
}
```

We begin with the obligatory check to make sure the rendered property is true; it's wise to avoid displaying components that aren't supposed to be visible. Next, we throw a custom runtime exception, `InvalidPropertiesException`, if all of the display properties are false. This is an unacceptable condition because there would be nothing to display.

Here's the rest of the method:

```
ResponseWriter writer = context.getResponseWriter();
Calendar calendar = getCalendar(context, (Date)getValue());

if (showDay)
{
    displaySelectFromCalendar(DAY_KEY, Calendar.DAY_OF_MONTH, "dd",
        calendar, calendar.getMinimum(Calendar.DAY_OF_MONTH),
        calendar.getMaximum(Calendar.DAY_OF_MONTH), context,
        writer);
}
if (showMonth)
```

```

{
    displaySelectFromCalendar(MONTH_KEY, Calendar.MONTH, "MMMM",
        calendar, calendar.getMinimum(Calendar.MONTH),
        calendar.getMaximum(Calendar.MONTH), context, writer);
}
if (showYear)
{
    int startValue = startYear, endValue = endYear;
    if (startYear == -1)
    {
        startValue = calendar.get(Calendar.YEAR);
    }
    if (endYear == -1)
    {
        endValue = calendar.get(Calendar.YEAR);
    }
    displaySelectFromCalendar(YEAR_KEY, Calendar.YEAR,
        "yyyy", calendar, startValue, endValue, context, writer);
}
if (showTime)
{
    displaySelectFromCalendar(TIME_KEY, Calendar.HOUR_OF_DAY,
        "hh:00 a", calendar,
        calendar.getMinimum(Calendar.HOUR_OF_DAY),
        calendar.getMaximum(Calendar.HOUR_OF_DAY),
        context, writer);
}
}
}

```

First, we grab a reference to the `ResponseWriter`, which we'll use for all output. Then, we get a `Calendar` object that wraps the component's current value. `getCalendar` is a utility method that returns a localized `Calendar` instance that's set to either the component's current value, or the current time if the component's value is `null`. It's important to work with a `Calendar` object as opposed to the `Date` object directly, since the `Calendar` has all of the information we need about specific fields like day and year.

The rest of the code simply calls another method, `displaySelectFromCalendar`, for each field that should be displayed. (The constants `DAY_KEY`, `MONTH_KEY`, `YEAR_KEY`, and `TIME_KEY` are strings that are used to generate form field names used during encoding and decoding.) The only nuance is that if `startYear` or `endYear` is set to `-1` (the default value), we use the `Calendar` object's current year for those values. This means that the drop-down listbox defaults to showing one number—the year of the underlying `Date` object.

The job of the `displaySelectFromCalendar` method is to display an HTML `<select>` element for the specified `Calendar` field (`Calendar.DAY`, `Calendar.MONTH`,

`Calendar.YEAR`, or `Calendar.HOUR_OF_DAY`). Here's the code, with the important lines marked in bold:

```

protected void displaySelectFromCalendar(String key,
                                         int calendarField, String dateFormatPattern,
                                         Calendar calendar, int minValue,
                                         int maxValue, FacesContext context,
                                         ResponseWriter writer)
                                         throws IOException
{
    Locale locale = context.getViewRoot().getLocale();
writer.startElement("select", this);
writer.writeAttribute("name", getFieldKey(context, key), null);

    SimpleDateFormat formatter = new SimpleDateFormat(
        dateFormatPattern,
        locale);
    Calendar tempCalendar = Calendar.getInstance(locale);
    tempCalendar.clear();
    tempCalendar.set(calendarField, minValue);
    boolean done = false;
    while (!done)
    {
        int currentFieldValue = tempCalendar.get(calendarField);
displayOption(formatter.format(tempCalendar.getTime()),
            currentFieldValue,
            calendar.get(calendarField) == currentFieldValue,
            writer);
        tempCalendar.roll(calendarField, 1);
        if (calendarField == Calendar.YEAR)
        {
            // years go on forever, so doesn't reset
            done = tempCalendar.get(calendarField) > maxValue;
        }
        else
        {
            // value resets
            done = tempCalendar.get(calendarField) == minValue;
        }
    }
writer.endElement("select");
}

```

The first thing we do is get the `Locale` for the current page through the `viewRoot` property of the `FacesContext`. We'll use this to make sure our all of our output is tailored for the user's culture. We start by writing the opening of the HTML `<select>` element. (The second parameter of `startElement` is the `UIComponent` associated with the element, which is the current `UIInputDate` instance in this case.)

Next, we add a `name` attribute. This attribute is important, because its value is the name of the request parameter value we'll be expecting when we write the `decode` method. The value is retrieved from the `UIInputDate` utility method `getFieldKey`, which just returns the component's `clientId` plus a separator (`NamingContainer.SEPARATOR_CHAR`), followed by the key sent in from the `encodeBegin` method. The key is a constant that maps to the field that's currently being displayed: day, month, year, or time. So, if the `clientId` is `_id1`, and the key is `TIME_KEY`, then the `getFieldKey` method would return `"_id1:time"`.¹ For a key of `YEAR_KEY`, it would return `"_id1:year"`. (The last parameter of `writeAttribute` is the name of the associated UI component property; since the `name` attribute isn't a property, we pass in `null`.)

TIP Whenever you're separating different parts of the name of a form field, you should use `NamingContainer.SEPARATOR_CHAR`. This ensures that your components behave consistently with your JSF implementation's naming conventions.

The rest of this code iterates through all of the necessary values for the selected Calendar field. So, if the field is `Calendar.MONTH`, it starts with January, then selects February, March, April, and so on. During each iteration, it calls `displayOption`, which displays an HTML `<option>` element for the current value. Here's `displayOption`:

```
protected void displayOption(String text, int value,
                            boolean selected, ResponseWriter writer)
                            throws IOException
{
    writer.startElement("option", this);
    writer.writeAttribute("value", new Integer(value), null);
    if (selected)
    {
        writer.writeAttribute("selected", "true", null);
    }
    writer.writeText(text, null);
    writer.endElement("option");
}
```

Pretty simple—it just displays an `<option>` element with the specified text and value, adding the `selected` attribute if necessary.

¹ This is, of course, assuming that the value of `NamingSeparator.SEPARATOR_CHAR` is `:`, which is the case with the reference implementation.

After iterating through all of the possible choices for the current field, we end the `<select>` element. Here's some sample output from calling `displaySelectFromCalendar` for the field `calendar.YEAR` when `startYear` is 2003 and `endYear` is 2004:

```
<select name="_id0:_id6:year">
    <option value="2003" selected="true">2003</option>
    <option value="2004">2004</option>
</select>
```

In this example, the `UIInputDate` instance's `clientId` was `id0:_id06`. Also, the component's date value was already set to 2003. It's easy to see how four calls to `displaySelectFromCalendar` can yield the output shown in listing 16.1.

That's it for encoding. Now, let's see how to decipher the input the component receives back from the user.

16.1.2 Decoding

Before we begin decoding, we must check to make sure that this component can really be updated. Here's the beginning of the `decode` method:

```
public void decode(FacesContext context)
{
    if (!Util.canModifyValue(this))
    {
        return;
    }
```

This little snippet calls the `canModifyValue` method, which returns `true` if the component's `rendered` property is `true` and the `readonly` and `disabled` attributes are set to `false` (if they are present at all). These attributes are commonly used for HTML rendering, and if they're set to `true`, then the component's value shouldn't be changed. The same holds true for the component's `rendered` property, since an invisible component shouldn't be able to collect user input.

Now, on to the meat (or tofu, as the case may be) of the method:

```
Date currentValue = (Date)getValue();
Calendar calendar = getCalendar(context, currentValue);

Map requestParameterMap =
    context.getExternalContext().getRequestParameterMap();
String dayKey = getFieldKey(context, DAY_KEY);
String monthKey = getFieldKey(context, MONTH_KEY);
String yearKey = getFieldKey(context, YEAR_KEY);
String timeKey = getFieldKey(context, TIME_KEY);
if (requestParameterMap.containsKey(dayKey))
{
    calendar.set(Calendar.DAY_OF_MONTH,
```

```

        Integer.parseInt(
            (String)requestParameterMap.get(dayKey)) ;
    }
    if (requestParameterMap.containsKey(monthKey))
    {
        calendar.set(Calendar.MONTH,
            Integer.parseInt(
                (String)requestParameterMap.get(monthKey))) ;
    }
    if (requestParameterMap.containsKey(yearKey))
    {
        calendar.set(Calendar.YEAR,
            Integer.parseInt(
                (String)requestParameterMap.get(yearKey))) ;
    }
    if (requestParameterMap.containsKey(timeKey))
    {
        calendar.set(Calendar.HOUR,
            Integer.parseInt(
                (String)requestParameterMap.get(timeKey))) ;
    }

    setSubmittedValue(calendar.getTime());
}

```

The decoding process is even simpler than encoding. All we have to do update the `submittedValue` based on request parameters. As a matter of fact, the first step is simply to retrieve the component's current value property. This is the value we'll be updating based on the user's input. In order to better manipulate that value, we also create a localized `Calendar` object that represents the `Date` value.

Next, we retrieve the request parameter map from the `ExternalContext`, and initialize all of the keys for each of the fields. Remember, these are the same keys we used for the `name` attribute for the `<select>` elements outputted in the `encode-Begin` method. The keys, which are derived from the client identifier, are the way we map between the encoding and decoding processes (see chapter 2 for more information about client identifiers in general).

For each key, we check to see if there is a corresponding value in the request parameter map; if so, we set the appropriate `Calendar` field to equal that value. In essence, we're updating each field of the component's value based on the user's input. Once that process is complete, we updated the `submittedValue`. In this particular case, the submitted value is derived from converting the `Calendar` back into a `Date` object (which is what `getTime` does). We don't need to update the `value` property explicitly—`UIInput` will update it for us during the Process Validations phase if the submitted value passes validation.

NOTE

In this example, we're performing conversion during the process of decoding. Often, in the `decode` method, you'll set `submittedValue` to the raw `String` received from the request, and then perform conversion in the `validate` method before validation occurs. The key is that `submittedValue` should retain enough data to re-display incorrect data to the user.

That's it for encoding `UIInputDate`. There is, however one more step: state management.

16.1.3 Implementing StateHolder methods

For any component that has a value that you'd like to remember between different requests (which is usually the case), you also need to implement the `StateHolder` interface (introduced in chapter 15). `UIInput` implements this interface already, but some methods must be overridden in order to support the additional properties shown in table 16.1.

The `StateHolder` interface has two methods that are the inverse of each other: `saveState` and `restoreState`. Here's how `UIInputDate` implements them:

```
public Object saveState(FacesContext context)
{
    Object[] values = new Object[7];
    values[0] = super.saveState(context);
    values[1] = showDay ? Boolean.TRUE : Boolean.FALSE;
    values[2] = showMonth ? Boolean.TRUE : Boolean.FALSE;
    values[3] = showYear ? Boolean.TRUE : Boolean.FALSE;
    values[4] = showTime ? Boolean.TRUE : Boolean.FALSE;
    values[5] = new Integer(startYear);
    values[6] = new Integer(endYear);

    return values;
}

public void restoreState(FacesContext context, Object state)
{
    Object[] values = (Object[])state;
    super.restoreState(context, values[0]);
    showDay = ((Boolean)values[1]).booleanValue();
    showMonth = ((Boolean)values[2]).booleanValue();
    showYear = ((Boolean)values[3]).booleanValue();
    showTime = ((Boolean)values[4]).booleanValue();
    startYear = ((Integer)values[5]).intValue();
    endYear = ((Integer)values[6]).intValue();
}
```

The most important thing to note is that we're returning and retrieving an array of `Objects`. The first `Object` in the array is the superclass's state; the other objects represent the properties of this component. It's important to always invoke the

superclass's implementation of the method, and to do so with the same array position in both methods.

This concludes our tour of `UIInputDate`. With a relatively small amount of code, we've been able to create a simple, self-contained, reusable, localizable, date input control. Now, let's configure JSF to recognize its existence.

16.2 Registering the component

We can configure `UIInputDate` with a simple `<component>` element in an application configuration file:

```
<component>
    <description>A simple date entry component.</description>
    <display-name>Input Date</display-name>
    <component-type>jia.InputDate</component-type>
    <component-class>org.jia.components.UIInputDate</component-class>
</component>
```

This configures our `UIInputDate` component (`org.jia.components.UIInputDate`) under the component type `jia.InputDate`, which is the value of the `UIInputDate.COMPONENT_TYPE` constant. For tools, we supply a display name of “Input Date”, as well as a full description. You can also provide an icon and additional metadata for custom components; see chapter 15 for details.

16.3 JSP integration

Once `UIInputDate` has been created and registered with the JSF application, we need to integrate it into the world of JSP. This involves writing a custom tag handler and adding the tag handler to a tag library. We'll then look at how to use `UIInputDate` inside of a JSP.

16.3.1 Writing the JSP custom tag

The tag handler for `UIInputDate` is a simple affair. We'll call this class `org.jia.components.taglib.InputDateTag` and subclass `UIComponentTag`. First, we need to override the `componentType` property:

```
public String getComponentType()
{
    return UIInputDate.COMPONENT_TYPE;
}
```

There's no separate renderer for this component, so we return `null` for the `rendererType` property:

```
public String getRendererType()
{
    return null;
}
```

We also need to define the properties `UIInputDate` defines: `date`, `showDay`, `showMonth`, `showYear`, `showTime`, `startYear`, and `endYear` (listed in table 16.1). The difference between the properties in `UIInputDate` and the ones in `InputDateTag` is that we use the primitive object wrappers in `InputDateTag`. This allows us to be aware of whether the value has been initialized.

In addition to supporting `UIInputDate`'s properties, we want to support the `immediate` property, which is exposed by `UIInput`. Other properties, like `id`, are already supported by `UIComponentTag`.

Once we've added all of these properties, we need to override `UIComponentTag`'s `setProperties` method to transfer the values from the tag handler to the associated component instance:

```
protected void setProperties(UIComponent component)
{
    super.setProperties(component);
    UIInputDate uiDate = (UIInputDate)component;
    if (showDay != null)
    {
        uiDate.setShowDay(showDay.booleanValue());
    }
    if (showMonth != null)
    {
        uiDate.setShowMonth(showMonth.booleanValue());
    }
    if (showYear != null)
    {
        uiDate.setShowYear(showYear.booleanValue());
    }
    if (startYear != null)
    {
        uiDate.setStartYear(startYear.intValue());
    }
    if (endYear != null)
    {
        uiDate.setEndYear(endYear.intValue());
    }
    if (showTime != null)
    {
        uiDate.setShowTime(showTime.booleanValue());
    }
    if (value != null)
    {
        if (isValueReference(value))
```

```

{
    uiDate.setValueBinding("value",
        getFacesContext().getApplication().
        createValueBinding(value));
}
else
{
    uiDate.setValue(value);
}
}
if (immediate != null)
{
    uiDate.setImmediate(immediate.booleanValue());
}
}
}

```

We begin by calling the superclass's `setProperties` method, which is necessary for supporting basic properties such as `id` and `rendered`. Next, for each non-null tag handler property, we set the corresponding property for the associated component. The interesting part is setting the `value` property. Note that we call the `isValueReference` method (from `UIComponentTag`) to see if it's a value-binding expression. (We know it's a value-binding expression if it starts with “`#{`” and ends with “`}`”.) If so, we add a `ValueBinding` instance instead of explicitly setting the `value` property. This is how you support value-binding expressions for tag handler attributes. You can do this for any attribute you wish; for example, the standard components allow value-bindings in properties like `immediate`, as well as renderer-dependent properties like `cellpadding` and `style`.

TIP In most cases, the `value` property (at a minimum) should support value-binding expressions. However, we recommend enabling every exposed property for value-binding expressions; this is the approach we take in later chapters.

Finally, all of the instance variables should be cleared in the `release` method:

```

public void release()
{
    super.release();
    showDay = null;
    showMonth = null;
    showYear = null;
    showTime = null;
    startYear = null;
    endYear = null;
    value = null;
    imediate = null;
}

```

After calling the superclass's `release` method, we simply set all of the instance variables to `null`.

This completes the requirements for `InputDateTag`. Now we just need to validate it.

16.3.2 Validating the tag

`UIInputDate` has a very simple requirement for its properties: `showDay`, `showMonth`, `showYear`, and `showTime` can't all be `false`. This requirement is enforced in the component's `encodeBegin` method, and it also needs to be enforced at the JSP level. You can enforce tag-level restrictions by extending the `javax.servlet.jsp.tagext.TagExtraInfo` class. The `InputDateTEI` class is shown in listing 16.2.

Listing 16.2 InputDateTEI.java: Validates InputDateTag

```
package org.jia.components.taglib;

import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;

public class InputDateTEI extends TagExtraInfo
{
    public InputDateTEI()
    {
    }

    public boolean isValid(TagData tagData) ← Subclasses
    {                                         TagExtraInfo
        return (isTrue(tagData.getAttribute("showDay")) ||
                isTrue(tagData.getAttribute("showMonth")) ||
                isTrue(tagData.getAttribute("showYear")) ||
                isTrue(tagData.getAttribute("showTime")));
    }

    protected boolean isTrue(Object booleanAttribute)
    {
        return (booleanAttribute == null ||
                booleanAttribute == TagData.REQUEST_TIME_VALUE ||
                booleanAttribute.toString().equalsIgnoreCase("true"));
    }
}
```

Returns true
if attributes
are valid

Note that the `isTrue` method returns `true` even if the attribute isn't set. (The `TagData.REQUEST_TIME_VALUE` constant means that the attribute has been set, but it's an expression, so it hasn't been evaluated yet.) This way, unless all four properties

are set to `false`, we still consider the tag valid. However, this doesn't mean the component itself won't complain if its properties were all set to `false` in some other manner.

Now that we've written that tag validator, let's add it (and the tag itself) to the tag library.

16.3.3 Adding the tag to the tag library

Adding `InputDateTag` to our custom tag library is simple. It's a typical tag handler entry that exposes all of the tag handler properties that we defined, plus the `id` and `rendered` properties defined in `UIComponentTag`. The full entry is shown in listing 16.3 as part of the JSF in Action tag library.

Listing 16.3 Tag library entry for InputDateTag

```
<?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>
  <!-- Tag Library Description Elements -->
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>JSF in Action Custom Tags</short-name>
  <uri>jsf-in-action-components</uri>
  <description>
    Sample custom components, renderers, validators, and
    converters from JSF in Action.
  </description>
  <!-- Tag declarations -->
  ...
  <tag>
    <name>inputDate</name>
    <tag-class>
      org.jia.components.taglib.InputDateTag
    </tag-class>
    <attribute>
      <name>id</name>
      <required>false</required>
      <rtpexprvalue>false</rtpexprvalue>
    </attribute>
    <attribute>
      <name>rendered</name>
      <required>false</required>
      <rtpexprvalue>false</rtpexprvalue>
    </attribute>
    <attribute>
      <name>binding</name>
```

① Implemented by
`UIComponentTag`

```
<required>false</required>
<rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>showDay</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>showMonth</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>showYear</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>startYear</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>endYear</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>showTime</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>value</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>immediate</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
</tag>
...
</taglib>
```



Implemented by
UIComponentTag

The name of this tag is `inputDate`. The first two attributes, referenced by ①, are from `UIComponentTag`. All of the other attributes are from `InputDateTag`. Note that we always set `<rtxprvalue>` to `false`.

Now that we've developed the component tag and its validator, let's look at using it.

16.4 Using the component

In the simplest case, we can invoke `UIInputDate` with a single tag and no attributes:

```
<jia:inputDate/>
```

This creates a `UIInputDate` instance with all of the default property values. The `value` property will default to the date and time when it was displayed. This tag is equivalent to the following:

```
<jia:inputDate showDay="true" showMonth="true" showYear="true"
    showTime="true" startYear="-1" endYear="-1"/>
```

Either one of these tag declarations will produce the HTML output shown in listing 16.1, assuming that the current time is December 13th, 2003, at around 9:00 P.M. In a browser, it looks like figure 16.3.



Figure 16.3 Default usage of `UIInputDate`.

By default, the `startYear` is equal to the current year, and so is the `endYear`. So, in the year drop-down listbox in figure 16.3, there's only one choice: 2003. We can specify the range for the year drop-down listbox by using real values for `startYear` and `endYear` rather than -1:

```
<jia:inputDate startYear="2003" endYear="2010"/>
```

This initializes the component with all of the default values, but specifies a year range of 2003 to 2010. In a browser, it looks like figure 16.4.

Now, the year drop-down listbox allows the user to select any year in between the `startYear` of 2003 and the `endYear` of 2010.

Often, showing all four fields is overkill. We can hide any of them by setting their corresponding `show` property to `false`. The following tag declaration hides the year and time:

```
<jia:inputDate showYear="false" showTime="false"/>
```

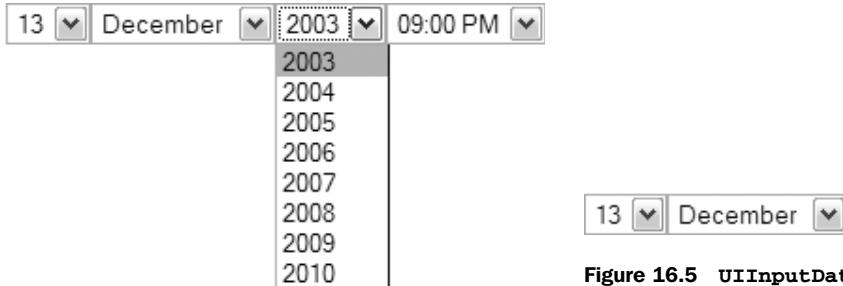


Figure 16.4 `UIInputDate` with a specific date range.

Figure 16.5 `UIInputDate` with `showYear` and `showTime` set to `false`.

In a browser, it looks like figure 16.5.

All of the examples so far aren't associated with a bean; we've just been seeing the local value of the component, which was initialized to the current date. In the real world, you would want to associate this component with a model object. Here's an example that does just that:

```
<jia:inputDate value="#{myBean.tradeDate}" showDay="true"
    showMonth="true" showYear="true" showTime="false"
    startYear="2003" endYear="2004"/>
```

This example displays the day, month, and year, with the year choices ranging from 2003 and 2004. The value displayed will always be synchronized with the Date object referenced by the value-binding expression "`#{myBean.tradeDate}`". Assuming that the date is set to September 14, 2003, figure 16.6 shows what the previous tag declaration would look like in a browser.



Figure 16.6 `UIInputDate` associated with a backing bean property and `showTime` set to `false`.

Of course, any edits made in the component will be synchronized with the underlying Date object.

That's it for `UIInputDate`. We were able to create a useful input control with a fairly small amount of effort.

16.5 Summary

In this chapter, we developed a simple date input control called `UIInputDate`. This control is a wrapper around a `java.util.Calendar` object. It handles all encoding and decoding internally, and exposes properties that control which part of the date should be displayed. Since `UIInputDate` adds additional properties, we also had to override methods of the `StateHolder` interface to ensure that they would be saved and restored properly.

Integrating the component with JSP required a simple tag handler, `InputDateTag`. We also provided an additional class, `InputDateTEI`, for validating the tag handler. Both of these classes were declared in a JSP tag library.

`UIInputDate` handles encoding and decoding duties internally. In the next chapter, we change gears and examine developing a renderer for the standard `UICommand` component.

17

RolloverButton renderer: a renderer with JavaScript support

This chapter covers

- Developing a custom renderer
- Outputting JavaScript
- Decorating an existing renderer

In chapter 8, we showed you how to build a login page with JavaScript events that achieved a rollover effect for the Submit button. The JavaScript in that page was handcoded; if we wanted to have that effect on other buttons, we'd have to either copy the JavaScript from page to page, or move it into a separate file. Every time we wanted an image rollover effect, we'd have to manually reference the function in the button's `mouseout` and `mouseover` event handlers. Sound familiar?

With a component-oriented technology like JSF, it should be possible to encapsulate the JavaScript functionality into the component itself, so that the front-end developer doesn't have to worry about the details when we're assembling the user interface. It should be possible to simply place a button on a page and specify the normal and rollover image locations without any additional work. This shifts the burden of maintaining JavaScript libraries to the component developer's arena. So let's tackle the job of developing a `UICommand` component that supports an image rollover.

At first glance, it seems like making a rollover-enabled button would require subclassing `UICommand`. If you think about it, though, the only difference between a normal button and a rollover button is the way the component is displayed. In other words, there are no behavioral changes; the changes are solely in their renderer domain.

Whenever you need to display a component in a different way, you're better off developing a separate `Renderer` class. `Renderers` are specifically tailored for encoding and decoding a component, and nothing more. (For more information about renderer basics, see chapter 15.)

Ideally, we'd like our renderer to function exactly like the standard `Button` renderer, except that it should also support an extra image that will be displayed when the user places the mouse over the button. Figure 17.1 shows our new renderer, called `RolloverButton` at work.

BY THE WAY

If you're wondering why we don't subclass the `Button` renderer instead of writing a new renderer from scratch, it's because the JSF specification doesn't specify any `Renderer` subclasses. So, there is no standard `Button` renderer class to subclass. You can use a vendor-specific class, but do so only if it's well documented and you're not afraid of vendor lock-in. If you want to reuse an existing renderer's functionality, your best bet is to wrap the renderer class. We show this technique in section 17.4.

As figure 17.1 shows, our renderer supports the standard `Button` renderer functionality. The only difference is that it supports rollover functionality. All of the elements necessary for developing this renderer are shown in figure 17.2.

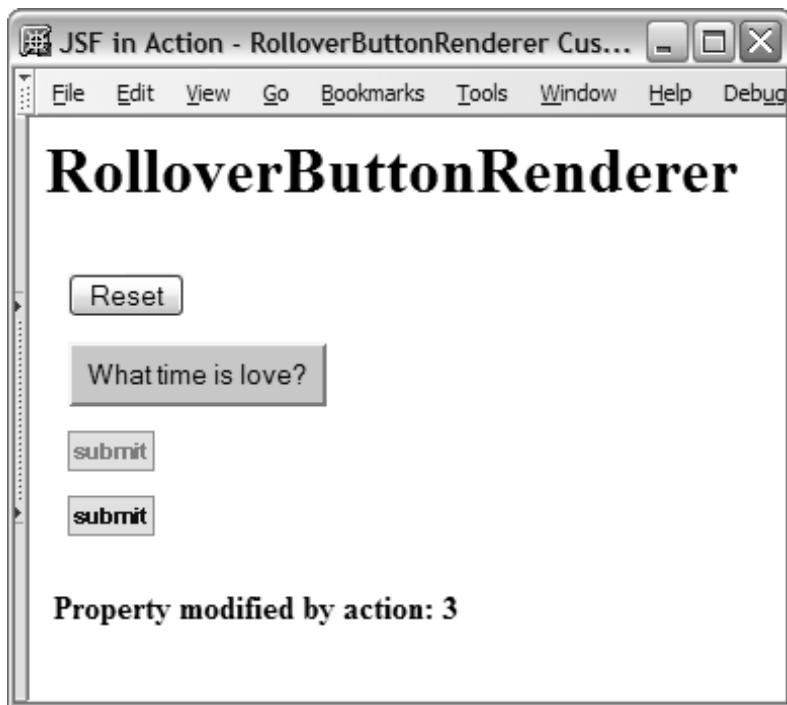


Figure 17.1 The *RolloverButton* renderer adds rollover functionality to a *UICommand* component. Some of the buttons in this example execute an action that modifies a backing bean property.

The first step, of course, is to write the renderer class itself.

NOTE Some portions of listings in this chapter have been omitted. You can download the full source code for this book from the JSF in Action web site (<http://www.manning.com/mann>).

17.1 Writing the *RolloverButtonRenderer* class

Developing renderers is easier than developing components with rendering functionality because you're only focusing on encoding and decoding. In simpler cases like this one, you only need to override two methods: `encodeBegin` and `decode`. Our goal is to output an HTML `<input>` element with the `type` attribute set to either `submit`, `reset`, or `image`, depending on what attributes are set. If a

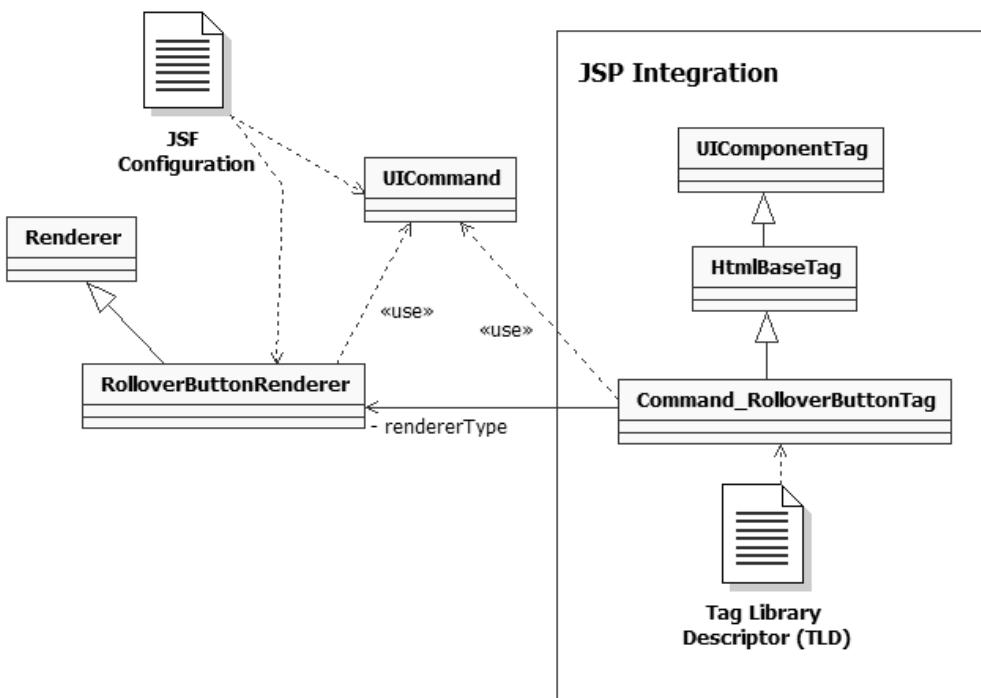


Figure 17.2 Classes and configuration files for the `RolloverButton` renderer. The `RolloverButtonRenderer` class subclasses `Renderer` and works with `UICommand`. It is integrated with JSP through the `Command_RolloverButtonTag` class, which subclasses `HtmlBaseTag`, which in turn subclasses `UIComponentTag`.

rollover image attribute is set, we want to output JavaScript that will change the image attribute in the onmouseover and onmouseout event handlers.

In addition to handling these special JavaScript duties, we'd like to support the standard Button renderer behavior. We'll call this class `RolloverButtonRenderer` and place it in the `org.jia.components` package.

Unlike components, which generally have properties that affect their behavior, renderers use attributes that are stored dynamically on the component instance. This allows them to remain stateless and consequently efficient. Each Renderer instance can be used for multiple components simultaneously, so they must be thread-safe.

When you're developing a new renderer, you must determine the attributes it needs to understand. For `RolloverButtonRenderer`, we must support the same set of pass-through attributes the `Button` renderer supports. These include `title`,

accessKey, and style, as well as event-handler attributes like onmouseout, onmouseover, onclick, onkeydown, and so on—attributes that aren’t interpreted by code and are simply sent through to the browser. (For simplicity, we’ll put styleClass in this group, even though it’s displayed as the class attribute rather than the styleClass attribute.)

There are also attributes that are specific to RolloverButtonRenderer, shown in table 17.1. (There are quite a few HTML pass-through attributes; there’s no need to list them all here.)

Table 17.1 The attributes for RolloverButtonRenderer.

Property	Type	Description	Button	RolloverButton
type	String	The type for this <input> element. Can be “submit” or “reset”. (Automatically set to “image” if the image property is set.)	X	
image	String	URL of image to be displayed. The type will automatically be set to “submit” if this attribute is set.	X	
rolloverImage	String	URL of image to display when user puts the mouse over the button.		X

The only attribute not currently supported by the Button renderer is rolloverImage. Other than that, we’re duplicating the original Button renderer’s functionality. (For more information on how it works, see the section on `HtmlCommandButton` in chapter 5.)

17.1.1 Encoding

When you’re writing a new renderer, it’s helpful to look at the output you’re aiming to generate first. Duplicating the Button renderer’s output is straightforward—it’s a standard HTML `<input>` element. Providing the rollover functionality requires a little more work, as shown in listing 17.1.

Listing 17.1 Sample output for RolloverButtonRenderer with the rollover image specified

```
<input type="image" title="Submit" name="foo"
       src="images/submit.gif"
       onmouseover="document.getElementsByName('foo')[0].
                   set_image(this,'images/submit_over.gif')"
       onmouseout="document.getElementsByName('foo')[0].
                   set_image(this,'images/submit.gif');"/>
```

```
<script language="JavaScript">
    document.getElementsByName('foo')[0].set_image =
        function(button, img)
    {
        button.src = img;
    }
</script>
```

The first thing you may have noticed is that there's really no need for a function here. We could directly modify the `src` property of the `<input>` element in the `onmouseover` and `onmouseout` event handlers. However, this is a book, and the goal is to teach, so I figured I'd add the function so that you can see a possible way to support JavaScript functions.

Speaking of JavaScript functions, you may also have noticed that the name is a little strange. It's actually declared as a property of the `<input>` element. This is a handy trick that avoids name conflicts. If the function were declared globally, it could conflict with other JavaScript function names on the page. Because our renderer will be responsible for generating this element (and consequently its `name` attribute) as well as the JavaScript code, we can rest better knowing that conflicts will be avoided.

The button itself will be displayed using the image `images/submit.gif`. When the user puts the mouse over the button, the `mouseover` event will fire, and the function will be called, changing the `src` property to `images/submit_over.gif` and thus causing that image to be displayed instead. When the user moves the mouse away from the button, the `mouseout` event will be called, which will call our method and change the `src` property back to `images/submit.gif`. No rocket science here.

Now that we've examined the output and supported attributes of `RolloverButtonRenderer`, let's start developing the `encodeBegin` method. Writing this method is similar to writing `UIComponent`'s version of `encodeBegin`; the only difference is that the `UIComponent` instance is passed in as a parameter. As always, it's best to start by checking to see if the component's `rendered` property is `false`:

```
public void encodeBegin(FacesContext context, UIComponent component)
    throws java.io.IOException
{
    if (!component.isRendered())
    {
        return;
    }
```

If the property is `false`, the component shouldn't be visible, so we should avoid any additional work. We also want to make sure that we have the proper combination of attributes:

```
UICommand button = (UICommand)component;
Map attributes = button.getAttributes();
String clientId = button.getClientId(context);
ResponseWriter writer = context.getResponseWriter();

if ((attributes.get("image") == null) &&
    (attributes.get("value") == null))
{
    throw new InvalidAttributesException(
        "Either the image or value attribute must be set.");
}
```

Here, we declare some variables, and then check to make sure that either the `image` or `value` property has been set. One of these attributes is required, because an `<input>` element needs to either display an image, as defined by the `image` attribute, or a label, as defined by the `value` attribute. If one of these properties isn't set, we throw an `InvalidAttributesException`, which is a custom exception written just for this purpose.

Next, let's start writing the `<input>` element and output some basic attributes:

```
writer.startElement("input", button);
writer.writeAttribute("name", clientId, "id");
Util.writePassthroughAttributes(button.getAttributes(), writer);
```

Here, we start the element, write the `name` attribute, and write the HTML pass-through attributes. The `Util.writePassthroughAttributes` method iterates through an array of all the pass-through attributes, and calls `writer.writeAttribute` for ones that exist in the current component's attribute `Map`.

The next bit of code handles the `image` and `rolloverImage` attributes, if they're set:

```
String imageSrc = getImageSrc(context,
                               (String)attributes.get("image"));
String rolloverImageSrc = getImageSrc(context,
                                       (String)attributes.get(
                                           "rolloverImage"));

if (imageSrc != null)
{
    writer.writeAttribute("type", "image", "type");
    writer.writeAttribute("src", imageSrc, "image");
    if (rolloverImageSrc != null)
    {
        writer.writeAttribute("onmouseover",
            "document.getElementsByName('" + clientId +
            "')[0].set_image(this, '" + rolloverImageSrc +
```

```

        " ')",
        null);
writer.writeAttribute("onmouseout",
    "document.getElementsByName('" + clientId +
')[0].set_image(this, '" + imageSrc + "')",
null);
}
}
}
```

First, we retrieve friendlier versions of the `image` and `rolloverImage` properties. The `getImageSrc` method prepends the image URL with the context path (the web application's name) if it's not an absolute URL.

If an image was indeed set, we go ahead and write the `type` and `src` attributes, hardcoding the `type` as “image” because we know an image should be displayed. If a rollover image was set, we write the `onmouseover` and `onmouseout` attributes. Note that this is the same JavaScript code as shown in listing 17.1, except that we use the `clientId` instead of the venerable “foo” to reference the `<input>` element. Remember, the `clientId` is what we used for the `name` attribute earlier in this method. It should always be used when you're outputting markup that references the component.

Now, let's finish outputting the `<input>` element:

```

else
{
    String type = (String)attributes.get("type");
    if (type != null)
    {
        writer.writeAttribute("type", type, "type");
    }
    else
    {
        writer.writeAttribute("type", "submit", "type");
    }
    String value = (String)attributes.get("value");
    if (value != null)
    {
        writer.writeAttribute("value", value, "value");
    }
}
writer.endElement("input");
```

If the `image` attribute wasn't set, we treat this like a normal `<input>` element and write the specified `type` attribute (defaulting to “submit” if no `type` was set). Next, we write the element's `value` attribute.

Finally, we end the `<input>` element with a call to `ResponseWriter.endElement`. However, we still have to output the JavaScript function itself:

```
if ((imageSrc != null) && (rolloverImageSrc != null))
{
    writer.startElement("script", button);
    writer.writeAttribute("language", "JavaScript", null);
    writer.writeText("document.getElementsByName('" + clientId +
                    "')[0].set_image = function(button, img)",
                    null);
    writer.writeText("{", null);
    writer.writeText("button.src = img;", null);
    writer.writeText("}", null);
    writer.endElement("script");
}
}
```

There's no need to output the JavaScript if no images were specified, so we perform that check first. If both images were specified, it's okay to output the necessary JavaScript. We start with the `<script>` element, and use `ResponseWriter.writeText` to fill in the body of the element with the actual JavaScript code. The `encodeBegin` method is now complete. It can support rendering normal buttons as well as buttons with a simple JavaScript rollover.

NOTE You may have noticed that we accessed component properties (like `value`) and renderer-dependent attributes (like `image`) through the component's `attributes` property. Remember, the `attributes` property returns a `Map` that includes both attributes as well as access to all of the component's properties. Accessing both attributes and properties in the same manner simplifies the process of developing renderers.

This also means that all of the strongly typed HTML pass-through properties available through the HTML components are also available as attributes.

Now, let's examine the opposite process: decoding.

17.1.2 Decoding

Decoding a `UICommand` is much simpler than encoding it. All we want to know is whether the user clicked the button; there's no need to modify the component's `value` at all. Here's the code:

```
public void decode(FacesContext context, UIComponent component)
{
    if (Util.canModifyValue(component))
    {
        String type = (String)component.getAttributes().get("type");
        String clientId = component.getClientId(context);
        Map requestParameterMap =

```

```
        context.getExternalContext().getRequestParameterMap();
if ((type == null || !type.equalsIgnoreCase("reset")) &&
    ((requestParameterMap.get(clientId) != null) ||
     (requestParameterMap.get(clientId + ".x") != null) ||
     (requestParameterMap.get(clientId + ".y") != null)))
{
    component.queueEvent(new ActionEvent(component));
}
}
```

First, we call the `canModifyValue` method, which returns `true` if the component's rendered property is `true`, and there are no HTML `readonly` or `disabled` attributes. If this method returns `false`, it means that the component isn't visible or active, so there's no need to check to see if a user interacted with it.

Next, we check to see if the type of the button is not “reset”, because reset buttons don’t generate ActionEvents. If the button isn’t a reset button, then we need to look for the proper parameter in the `requestParameterMap`. Any parameter name that has the name of the `clientId` indicates that the button was clicked. Also, any parameter name that starts with the `clientId` and ends with either “x” or “y” means that the button was clicked. This is what the browser would send back if an image map was displayed. If the proper parameter has been found, there’s no need to see what the value of the parameter was; all we have to do is enqueue a new `ActionEvent` on the component. This allows the default `ActionListener`, and any additional registered `ActionListeners`, to process the event later in the lifecycle.

This concludes our tour of the `RolloverButtonRenderer` class. You can download the full source code from the book's web site. Now, let's configure this renderer.

17.1.3 Registering the renderer

Instead of declaring a component with the `<component>` tag, we'll need to declare `RolloverButtonRenderer` with the `<renderer>` tag. However, renderers can't exist by themselves—they must be part of a `RenderKit`. In most cases, if your Renderer outputs HTML, you can just add it to the default `RenderKit`:

```
<render-kit>
  <renderer>
    <display-name>Rollover Button</display-name>
    <component-family>javax.faces.Command</component-family>
    <renderer-type>jia.RolloverButton</renderer-type>
    <renderer-class>org.jia.components.RolloverButtonRenderer</renderer-class>
  </renderer>
</render-kit>
```

Notice that we didn't specify the `<render-kit-id>` or `<render-kit-class>` element. This is because we wanted this renderer to be added to the default `RenderKit`. We also specified a display name of "Rollover Button" and the type "jia.RolloverButton". These choices seem logical, since this is essentially a slightly enhanced version of the standard `Button` renderer.

For more on configuring renderers, see chapter 15. Next, we'll examine the JSP side of the coin.

17.2 JSP Integration

In its simplest form, JSP integration includes developing a single custom tag handler and then adding that tag to a tag library. As you write more tag handlers, you'll quickly find that sometimes you need to create a class hierarchy for component tags. This is definitely the case when it comes to supporting HTML pass-through attributes. We've glossed over these details so far, but in general, all tags and renderers that support HTML should support the appropriate HTML attributes. We've supported these attributes in `RolloverButtonRenderer`; now we need to support them in its custom tag.

All HTML elements support a basic set of attributes that include `style`, `class`, `title`, `onmouseout`, `onmouseover`, `onmousemove`, and so on. In order to handle this, we'll start by developing a base tag handler that has properties for all of the common HTML attributes. We'll then subclass this handler and add additional functionality to support the `RolloverButtonRenderer` and `UICommand` combination. (We'll also subclass this tag handler in the next chapter.)

BY THE WAY

If you're wondering why we don't subclass the tag handlers that ship with JSF, it's because the specification itself doesn't define any classes that are more specialized than `UIComponentTag`. This is to allow competitors to handle JSP support in different ways. Unfortunately, it also means that developers have to either build such functionality from scratch or subclass implementation-specific classes. If you don't mind vendor lock-in, and your vendor has well-documented tag handler classes you can use, subclassing their classes is a viable alternative to rolling your own.

Another option is code generation. If you specify all of the HTML attributes in an XML document (such as an application configuration file), you could generate the tag handler automatically. Both the reference implementation of JSF [Sun, JSF RI] and MyFaces [MyFaces] use some sort of (unpolished) code-generation technique to implement the standard tag handlers.

17.2.1 Writing the *HtmlBaseTag* class

HtmlBaseTag is a very simple abstract base class; it just exposes properties for the basic HTML element attributes, overrides *setProperties* to add those attributes to the associated component, and overrides *release* to set its instance variables to null. The code is shown in listing 17.2 (we've omitted most of the accessors and mutators for brevity).

Listing 17.2 The *HtmlBaseTag* class supports all of the common HTML attributes

```
package org.jia.components.taglib;

import org.jia.Util;

import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentTag;

import java.util.Map;

public abstract class HtmlBaseTag extends UIComponentTag
{
    private String styleClass;
    private String onclick;
    private String ondblclick;
    private String onkeydown;
    private String onkeypress;
    private String onkeyup;
    private String onmousedown;
    private String onmousemove;
    private String onmouseout;
    private String onmouseover;
    private String onmouseup;
    private String disabled;
    private String alt;
    private String lang;
    private String dir;
    private String tabindex;
    private String accesskey;
    private String title;
    private String style;

    public HtmlBaseTag()
    {
        super();
    }

    // HtmlBaseTag methods
    protected void setProperties(UIComponent component)
    {
```

```
super.setProperties(component);

Map attributes = component.getAttributes();
Util.addAttribute(attributes, "styleClass", styleClass);
Util.addAttribute(attributes, "onclick", onclick);
Util.addAttribute(attributes, "ondblclick", ondblclick);
Util.addAttribute(attributes, "onkeydown", onkeydown);
Util.addAttribute(attributes, "onkeypress", onkeypress);
Util.addAttribute(attributes, "onkeyup", onkeyup);
Util.addAttribute(attributes, "onmousedown", onmousedown);
Util.addAttribute(attributes, "onmousemove", onmousemove);
Util.addAttribute(attributes, "onmouseout", onmouseout);
Util.addAttribute(attributes, "onmouseover", onmouseover);
Util.addAttribute(attributes, "onmouseup", onmouseup);
Util.addAttribute(attributes, "disabled", disabled);
Util.addAttribute(attributes, "alt", alt);
Util.addAttribute(attributes, "lang", lang);
Util.addAttribute(attributes, "dir", dir);
Util.addAttribute(attributes, "tabindex", tabindex);
Util.addAttribute(attributes, "accesskey", accesskey);
Util.addAttribute(attributes, "title", title);
Util.addAttribute(attributes, "style", style);
}

public void release()
{
    super.release();
    styleClass = null;
    onclick = null;
    ondblclick = null;
    onkeydown = null;
    onkeypress = null;
    onkeyup = null;
    onmousedown = null;
    onmousemove = null;
    onmouseout = null;
    onmouseover = null;
    onmouseup = null;
    disabled = null;
    alt = null;
    lang = null;
    dir = null;
    tabindex = null;
    accesskey = null;
    title = null;
    style = null;
}

// Properties

public String getStyleClass()
```

```
{  
    return styleClass;  
}  
  
public void setStyleClass(String styleClass)  
{  
    this.styleClass = styleClass;  
}  
  
...  
  
public String getStyle()  
{  
    return style;  
}  
  
public void setStyle(String style)  
{  
    this.style = style;  
}  
}
```

As you can see, there's nothing particularly exciting about this class. It simply exposes properties for all of the standard HTML attributes, and sets them in the associated component. The `Util.addAttribute` method checks to see if the attribute is `null`; if not, it either creates and adds a new value-binding expression (if the value is an expression), or just sets the attribute's value. This is simply grunt work that's better only performed once. We assume that the associated renderer or component will output these attributes properly.

BY THE WAY

If you're wondering why we didn't need an analogous base `Renderer` class, it's because `Renderers` don't need to expose these attributes as properties, or set them on the associated `UIComponent` instance. They just have to read the attributes and display them, and this work can be formed in a single method: `Util.writePassthroughAttributes`. This method displays any HTML attributes (basic and element-specific) that are set for a particular `UIComponent` instance. We called this method in `RolloverButtonRenderer`'s `encodeEnd` method.

The next section describes `RolloverButtonRenderer`'s specific tag handler, which subclasses `HtmlBaseTag`.

17.2.2 Writing the JSP custom tag

We'll call this tag handler `CommandRolloverButtonTag` to signify its support for both the `UICommand` class (which is of type `javax.faces.Command`) and the `RolloverButtonRenderer` (which is of type `jia.RolloverButton`). The class will subclass `HtmlBaseTag`, which provides support for common HTML attributes. `HtmlBaseTag`, in turn, subclasses the JSF-provided `UIComponentTag`, which performs most of the basic plumbing.

We can start by defining the `componentType` and `rendererType` properties:

```
public String getComponentType()
{
    return HtmlCommandButton.COMPONENT_TYPE;
}

public String getRendererType()
{
    return "jia.RolloverButton";
}
```

These types map to `HtmlCommandButton` and `RolloverButtonRenderer`, respectively. Even though all of our code is based on `UICommand`, we use the component type for `HtmlCommandButton` so that any developers can access strongly typed properties in their code.

In the previous chapter, the properties exposed by a tag handler were the same as the ones exposed as its associated component. That's because everything we needed was exposed as a component property; there weren't any renderer-specific attributes. However, when you're developing tag handlers for renderer/component pairs, you need to support all of the renderer's attributes as well as the component's properties. Table 17.2 lists all of the properties that this tag handler needs to support.

Table 17.2 The properties for `Command_RolloverButtonTag`. This includes all of the attributes from `RolloverButtonRenderer` and the properties from `UICommand`.

Property	Type	Description	RolloverButton	Command
<code>type</code>	<code>String</code>	The type for this <code><input></code> element. Can be "submit" or "reset". (Automatically set to "image" if the <code>image</code> property is set.)	X	
<code>image</code>	<code>String</code>	URL of image to be displayed. The type will automatically be set to "image" if this attribute is set.	X	

continued on next page

Table 17.2 The properties for Command_RolloverButtonTag. This includes all of the attributes from RolloverButtonRenderer and the properties from UICommand. (continued)

Property	Type	Description	RolloverButton	Command
rolloverImage	String	URL of image to display when user puts the mouse over the button.	X	
value	String	Literal string or value-binding expression to be displayed as the button's label.		X
action	String	Literal string or method-binding expression that specifies specific outcome.		X
actionListener	String	Method-binding expression for an action listener method.		X
immediate	Boolean	True if the component should be processed during the Apply Request Values phase of the request processing lifecycle.		X

This list includes all the attributes supported by RolloverButtonRenderer as well as all of UICommand's properties. We've left out properties of UIComponentBase (like id and rendered) that are already handled by UIComponentTag.

These properties are used in CommandRolloverButtonTag's implementation of setProperties:

```

protected void setProperties(UIComponent component)
{
    super.setProperties(component);

    UICommand command = (UICommand)component;
    Application app = getFacesContext().getApplication();
    Map attributes = command.getAttributes();
    if (value != null)
    {
        if (isValueReference(value))
        {
            command.setValueBinding("value",
                app.createValueBinding(value));
        }
        else
        {
            command.setValue(value);
        }
    }
    if (immediate != null)
    {

```

```

if (isValueReference(immediate))
{
    command.setValueBinding("immediate",
                           app.createValueBinding(immediate));
}
else
{
    command.setImmediate(
        Boolean.valueOf(immediate).booleanValue());
}
}
}

```

We start by setting `UICommand` properties. The `value` and `immediate` properties can accept value-binding expressions, so if the variables are in the proper format (as determined by `isValueReference`, which is a `UIComponentTag` method), we need to add a `ValueBinding` for them; otherwise, we just set the property as usual.

The next step in the `setProperties` method is to process the `action` and `actionListener` properties:

```

if (action != null)
{
    MethodBinding actionBinding = null;
    if (isValueReference(action))
    {
        actionBinding = app.createMethodBinding(action, null);
    }
    else
    {
        actionBinding = new ConstantMethodBinding(action);
    }
    command.setAction(actionBinding);
}
if (actionListener != null)
{
    MethodBinding actionListenerBinding =
        app.createMethodBinding(actionListener,
                               new Class[] { ActionEvent.class });
    command.setActionListener(actionListenerBinding);
}

```

For the `action` property, if it is a method-binding expression, we create a new `MethodBinding` instance based on the property's value. Otherwise, we create a `ConstantMethodBinding` instance using the literal value of the `action` property. `ConstantMethodBinding` is a subclass of `MethodBinding` that just returns the literal value. (In other words, if the `action` property is “foo”, then the corresponding `ConstantMethodBinding` instance’s `invoke` method would simply return “foo”). This is how the reference implementation [Sun, JSF RI] handles static outcomes internally.

Finally, we finish the `setProperties` method by adding the renderer-dependent properties:

```
    Util.addAttribute(app, component, "image", image);
    Util.addAttribute(app, component, "rolloverImage", rolloverImage);
    Util.addAttribute(app, component, "type", type);
}
```

`Util.addAttribute` simply adds the attribute or creates a new `ValueBinding` instance if the value passed in is non-null. Since all of the code in this method creates `ValueBinding` instances if necessary, it's safe to say that all of the attributes of this component tag are value-binding enabled.

The only other step is clearing all of the instance variables with `release`:

```
public void release()
{
    super.release();

    image = null;
    rolloverImage = null;
    action = null;
    value = null;
    immediate = null;
    actionListener = null;
    action = null;
    type = null;
}
```

Note that we're clearing the instance variables for all of the properties listed in table 17.1. The only other methods in this class are getters and setters for these properties, so we'll spare you the details. You can download the full source from the book's web site.

Because `RolloverButtonRenderer` has some restrictions on its attributes that can't be enforced with a normal tag library descriptor, we must also write a tag validator.

17.2.3 Validating the tag

When `RolloverButtonRenderer` executes its `encodeBegin` method, it does a quick check to make sure that either the `image` or `value` attributes are set. The same sort of check should always be performed at the JSP level, so errors can be caught at translation time, when the JSP is compiled.

In JSP, you can validate the entire tag library by subclassing the `TagLibraryValidator` class, or you can validate an individual tag by subclassing the `TagExtraInfo` class. The former method is more powerful, especially if you're checking for dependencies between multiple checks. For simple validation, however,

`TagExtraInfo` will suffice, and that's how we've implemented the validation check for `CommandRolloverButtonTag`. The code is shown in listing 17.3.

Listing 17.3 CommandRolloverButtonTEI.java: Validates CommandRolloverButtonTag

```
package org.jia.components.taglib;

import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;

public class CommandRolloverButtonTEI extends TagExtraInfo
{
    public CommandRolloverButtonTEI()
    {
    }

    public boolean isValid(TagData tagData)
    {
        return ((tagData.getAttribute("image") != null) ||
               (tagData.getAttribute("value") != null));
    }
}
```

All the work happens in the `isValid` method. The `tagData` object passed in contains all of the attributes for the tag, as well as the tag's id (if available). All we do is check to make sure that either the `image` or `value` attributes are set, just as the renderer itself does.

BY THE WAY

If you're wondering why we need to do this check at both the renderer and JSP levels, it's because the two aren't required to be used together. What if, for example, you configured the `RolloverButtonRenderer` in code? You'd still want an exception to be thrown if you didn't set the proper attributes. Having the check at the renderer also helps with debugging the tag handler.

This completes the Java code required to integrate the renderer with JSP. To finish the process, we'll need to add it to the tag library.

17.2.4 Adding the tag to the tag library

The only thing that's unique about the `CommandRolloverButtonTag`'s tag descriptor is the fact that it includes not only all of the properties the class defines itself, but also all of `HtmlBaseTag`'s properties as well. The descriptor is shown in listing 17.4; we've omitted parts for brevity.

Listing 17.4 The tag descriptor for CommandRolloverButtonTag

```
<?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-jsp-taglibrary_1_2.dtd">
<taglib>
    <!-- Tag Library Description Elements -->
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>JSF in Action Custom Tags</short-name>
    <uri>jsf-in-action-components</uri>
    <description>
        Sample custom components, renderers, validatos, and
        converters from JSF in Action.
    </description>
    <!-- Tag declarations -->
    ...
    <tag>
        <name>commandRolloverButton</name>
        <tag-class>
            org.jia.components.taglib.CommandRolloverButtonTag
        </tag-class>
        <tei-class>
            org.jia.components.taglib.CommandRolloverButtonTEI
        </tei-class>
        <attribute>
            <name>id</name>
            <required>false</required>
            <rtpexprvalue>true</rtpexprvalue>
        </attribute>
        <attribute>
            <name>image</name>
            <required>false</required>
            <rtpexprvalue>true</rtpexprvalue>
        </attribute>
        <attribute>
            <name>rolloverImage</name>
            <required>false</required>
            <rtpexprvalue>true</rtpexprvalue>
        </attribute>
        <attribute>
            <name>action</name>
            <required>false</required>
            <rtpexprvalue>true</rtpexprvalue>
        </attribute>
        <attribute>
            <name>actionListener</name>
            <required>false</required>
            <rtpexprvalue>true</rtpexprvalue>
        </attribute>
```

```
<attribute>
    <name>immediate</name>
    <required>false</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<attribute>
    <name>rendered</name>
    <required>false</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<attribute>
    <name>value</name>
    <required>false</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<attribute>
    <name>binding</name>
    <required>false</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<attribute>
    <name>type</name>
    <required>false</required>
    <rteprvalue>true</rteprvalue>
</attribute>
<!-- HTML pass-through attributes -->
<attribute>
    <name>onclick</name>
    <required>false</required>
    <rteprvalue>true</rteprvalue>
</attribute>>
...
<attribute>
    <name>style</name>
    <required>false</required>
    <rteprvalue>true</rteprvalue>
</attribute>
</tag>
</taglib>
```

It's important to remember that whenever you subclass a tag handler, you need to still need to expose all of its properties in your new class's tag library entry.

TIP

If you have a lot of tags that use the same set of attributes, you can use XML entities to avoid repetitive, error-prone typing.

We've now succeeded in duplicating the functionality of the `<h:commandButton>` component tag, with the added ability to support a JavaScript rollover effect. In the next section, we take a look at using the tag.

17.3 Using the renderer

As intended, usage of this component tag is much the same as using the `<h:commandButton>` tag. For example, the following tag displays a simple reset button:

```
<jia:commandRolloverButton type="reset" value="Reset"/>
```

Figure 17.3 shows what this tag looks like in a browser. Since this is a reset button, no action event will be generated.

 **Figure 17.3 RolloverButton renderer displaying a normal reset button.**

The following example registers an action listener method, displays a label whose value comes from a value-binding expression, and uses a CSS class:

```
<jia:commandRolloverButton  
    actionListener="#{testForm.incrementCounter}"  
    value="#{testForm.message}" styleClass="button"/>
```

TestForm is a simple backing bean with some methods and properties that we use in these examples. The message property is equal to "What time is love?", so figure 17.4 shows what this tag looks like in a browser.

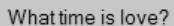


Figure 17.4 RolloverButton renderer with style sheets, executing an action listener.

If you click on the button shown in Figure 17.4, the `incrementCounter(ActionEvent)` method of `testForm` is called, which increments the value of its `counter` property.

The following example illustrates use of the `action` property with an image URL retrieved from a resource bundle:

```
<jia:commandRolloverButton action="#{testForm.incrementCounter}"  
    alt="Submit" title="Submit"  
    image="#{bundle.submitImage}"/>
```

Instead of registering an action listener method, this tag registers an action method. (Both versions of the `incrementCounter` method do the same thing.) It also retrieves a URL for the image from a resource bundle. Figure 17.5 shows what it looks like in a browser.



Figure 17.5 The `RolloverButton` renderer with an image, executing an action method.

Just like the last example, when a user clicks on the button shown in figure 17.5, the `counter` property of `testForm` will be incremented.

All of the previous examples could have used the standard `<h:commandButton>` tag. Using our new renderer's rollover functionality is as simple as adding the `rolloverImage` attribute to the previous example (and using a literal string for the `image` attribute). Here's the new tag, with an extra `style` attribute just for kicks:

```
<jia:commandRolloverButton action="#{testForm.incrementCounter}"  
    image="images/submit.gif"  
    rolloverImage="images/submit_over.gif"  
    alt="Submit" title="Submit"  
    style="margin-bottom: 10px"/>
```

Because the rollover functionality was the main purpose of this exercise, it's worth examining the HTML output. It's shown in listing 17.5.

Listing 17.5 Actual output from `RolloverButton` renderer with a rollover image specified

```
<input name="_id0:_id5" alt="Submit" title="Submit"  
    style="margin-bottom: 10px" type="image"  
    src="images/submit.gif"  
    onmouseover="document.getElementsByName('_id0:_id5')[0].  
        set_image(this, 'images/submit_over.gif')"  
    onmouseout="document.getElementsByName('_id0:_id5')[0].  
        set_image(this, 'images/submit.gif') />  
<script language="JavaScript">  
    document.getElementsByName('_id0:_id5')[0].set_image =  
        function(button, img){ button.src = img; }  
</script>
```

This should look similar to our desired output, which was shown in listing 17.1. In this case, however, we see JSF-generated client identifiers instead of "foo". Also, note that the `alt`, `title`, and `style` attributes were passed through.

Figure 17.6 shows what the HTML looks like in a browser (the rollover image, `submit_over.gif`, is currently showing). This isn't much to look at, but you get the idea. All of these examples are shown together in figure 17.1.



Figure 17.6 The `RolloverButton` renderer with a rollover image showing (the original image is the same as the one shown in figure 17.5).

Even though the results weren't spectacular, it was a decent amount of work for such a simple effect. It would be nice if we could use the standard `Button` renderer and delegate most of our work to it. Fortunately, we can wrap an existing renderer.

17.4 Wrapping an existing renderer

I must confess: developing the `RolloverButton` renderer from scratch was a bit of an unnecessary adventure. It was useful for explaining a lot of the concepts that are involved with writing real-world renderers, but it duplicated a lot of functionality available in the standard `Button` renderer. We couldn't subclass the `Button` renderer because there is no specific class defined by the JSF specification. Any vendor that develops a JSF implementation could have a completely different class for handling the `Button` renderer duties.

For example, the reference implementation's [Sun, JSF RI] class name is `com.sun.faces.renderkit.html_basic.ButtonRenderer`. You could subclass this class, but if you were to use a different implementation, like MyFaces [MyFaces], your code wouldn't be portable. A reasonable solution to this problem is to simply wrap your implementation's default `Button` renderer and work at the level of the `Renderer` abstract base class, which is guaranteed to be available in all JSF implementations.

Let's examine this solution, with the goal of duplicating all of the functionality provided by the `RolloverButton` renderer, with a lot less code (we hope). Because wrapping an existing object is often called the Decorator pattern [GoF], we'll call the class `RolloverButtonDecoratorRenderer`. (Okay, I admit it, I like long names.) Because there is no additional functionality, we'll spare you the additional screenshot; refer to figure 17.1 to see what it looks like.

17.4.1 Developing the `RolloverButtonDecoratorRenderer` class

The behavior of this class is identical to that of `RolloverButtonRenderer`, as described in section 17.1. That means it also supports the same properties, which are listed in table 17.1.

Encoding

The `encodeBegin` method begins in the same way, as well:

```
public void encodeBegin(FacesContext context, UIComponent component)
                        throws java.io.IOException
{
    if (!component.isRendered())
    {
        return;
    }
```

```
Map attributes = component.getAttributes();
if ((attributes.get("image") == null) &&
    (attributes.get("value") == null))
{
    throw new InvalidAttributesException(
        "Either the image, or value attribute must be set.");
}
```

This code is similar to the code in `RolloverButtonRenderer`. First, we exit the method if the component's `rendered` property is `false`. No need to display a component that's marked as invisible. Next, we grab the component's attributes and check to make sure that either the `image` or `value` attribute is not `null`. One of these attributes is required, or else we'll have nothing to display on the button. If neither one of them is defined, we throw an `InvalidAttributesException`, which is a custom exception that indicates a problem with the current set of attributes.

At this point in `RolloverButtonRenderer`, we declared some variables and wrote out a bunch of attributes, including the HTML pass-through attributes. We then wrote out the attributes for the `onmouseover` and `onmouseout` event handlers (if the `rolloverImage` attribute was set). Because we're delegating to another renderer, however, we can just let the other renderer output all those attributes. And since `onmouseover` and `onmouseout` are HTML pass-through attributes, it will render those as well. We still have to declare some variables, but there's no need to directly write out any attributes; we can just add them to the component:

```
UIComponent button = (UIComponent)component;
String clientId = component.getClientId(context);

String imageSrc = getImageSrc(context,
                               (String)attributes.get("image"));
String rolloverImageSrc =
    getImageSrc(context,
                (String)attributes.get("rolloverImage"));

if (imageSrc != null && rolloverImageSrc != null)
{
    attributes.put("onmouseover",
                  "document.getElementsByName('" + clientId +
                  "')[0].set_image(this, '" + rolloverImageSrc +
                  "')");
    attributes.put("onmouseout",
                  "document.getElementsByName('" + clientId +
                  "')[0].set_image(this, '" + imageSrc + "')");
}
```

First, we declare some variables. Note that we didn't grab a reference to a `ResponseWriter`; we don't need one yet. We then retrieve the URL for the image and roll-

over image. The method `getImageSrc` is exactly the same as it is in `RolloverButtonRenderer`; it simply prefixes the URL with a web application's pathname if it begins with a slash (/).

If an image URL and a rollover image URL were both set, we can get down to business. However, instead of directly writing out the `onmouseover` and `onmouseout` attributes, we just add them to the attribute map. The `Button` renderer will display these attributes, as well as all of the other ones that have been registered on the component.

TIP If you need to pass information from one `encode` method to another, use attributes rather than instance variables. A single `Renderer` instance can be executed in multiple threads, so the component instance itself is the best place to save component-specific state.

Now that all of the properties have been set, we can just delegate encoding to the standard `Button` renderer:

```
{
    getButtonRenderer(context).encodeBegin(context, button);
}
```

`getButtonRenderer` returns the default `Button` renderer; we let it perform the rest of `encodeBegin`. Here's the `getButtonRenderer` method:

```
protected Renderer getButtonRenderer(FacesContext context)
{
    RenderKitFactory rkFactory = (RenderKitFactory)FactoryFinder.
        getFactory(FactoryFinder.RENDER_KIT_FACTORY);

    RenderKit defaultRenderKit = rkFactory.getRenderKit(context,
        RenderKitFactory.HTML_BASIC_RENDER_KIT);

    return defaultRenderKit.getRenderer(UICommand.COMPONENT_FAMILY,
        "javax.faces.Button");
}
```

Here, we grab the `RenderKitFactory` from the `FactoryFinder` class (remember that we can use the `FactoryFinder` to retrieve any JSF factory). Once we've got the `RenderKitFactory`, we can retrieve the default `RenderKit` by using the `RenderKitFactory.HTML_BASIC_RENDER_KIT` constant. From the default `RenderKit`, we can retrieve the `Button` renderer.

NOTE We could store the `Button` renderer in an instance variable, but that would cause problems if the default `Button` renderer is changed at runtime.

You may be wondering what happened to the JavaScript that we rendered in `RolloverButtonRenderer`'s `encodeBegin` method. Unfortunately, we don't know whether the default `Button` renderer outputs the `<input>` element in the `encodeBegin` method or the `encodeEnd` method. The reference implementation [Sun, JSF RI] does all of its work in `encodeBegin`, but it's not safe to assume that every other implementation will do so as well. The JavaScript output must be after the `<input>` element, so it's safer to place it after the wrapped renderer's `encodeEnd`:

```
public void encodeEnd(FacesContext context, UIComponent component)
                      throws java.io.IOException
{
    getButtonRenderer(context).encodeEnd(context, component);

    if (component.getAttributes().get("image") != null &&
        component.getAttributes().get("rolloverImage") != null)
    {
        ResponseWriter writer = context.getResponseWriter();
        writer.startElement("script", component);
        writer.writeAttribute("language", "JavaScript", null);
        writer.writeText("document.getElementsByName('" +
                        component.getClientId(context) +
                        "')[0].set_image = function(component, img)" ,
                        null);
        writer.writeText("{", null);
        writer.writeText("component.src = img;", null);
        writer.writeText("}", null);
        writer.endElement("script");
    }
}
```

After calling the standard renderer's `encodeEnd`, we check to see if both the `image` and `rolloverImage` attributes have been set. If so, then we know we need to finish the rollover rendering job and output the JavaScript code. This should look similar to the output we generated in `RolloverButtonRenderer`'s `encodeBegin` method.

That was the hard part. Because there's no additional functionality to add to the wrapped renderer, we can just delegate to it for the `encodeChildren` method:

```
public void encodeChildren(FacesContext context,
                           UIComponent component)
                           throws java.io.IOException
{
    getButtonRenderer(context).encodeChildren(context, component);
}
```

That's it for encoding. Decoding is a piece of cake.

Decoding

`RolloverButtonRenderer` didn't add any special decoding functionality over and above what the `Button` renderer already does. Consequently, we can just delegate to it for the `decode` method as well:

```
public void decode(FacesContext context, UIComponent component)
{
    getButtonRenderer(context).decode(context, component);
}
```

Nothing terribly exciting here—we just call the corresponding method in the `Button` renderer.

What is exciting, however, is that we've successfully wrapped the `Button` renderer and added a rollover effect. We'll spare you the full listing; you can get it online from the book's web site.

This may not seem like a lot less code than `RolloverButtonRenderer`, but there was no need to call `Util.rendererPassthroughAttributes`, we didn't have to write out any attributes, and we didn't have to write any decoding logic. As a matter of fact, there was no need to worry about the standard functionality at all. This technique can save you a lot of time, especially if you're wrapping a renderer for a more complicated component, like `UIData`.

Configuring this renderer with your application and integrating it with JSP is almost exactly the same as the process for `RolloverButtonRenderer`. As a matter of fact, if you use the same renderer type, `jia.RolloverButtonRenderer`, it is exactly the same. The same goes for usage; refer to section 17.3 for examples.

17.5 Summary

This chapter walked through the process of adding a JavaScript rollover effect to a `UICommand` component. You can't portably subclass standard renderer implementations, so our first approach was to develop an entirely new renderer called `RolloverButton`. This renderer is a replacement for the standard `Button` renderer, and is responsible for all of the same encoding and decoding responsibilities for a `UICommand` component. In addition to `UICommand`'s properties, the component has to support HTML pass-through attributes.

The need to support additional HTML attributes makes JSP integration more complicated, because each attribute must be exposed as a tag handler property. In order to handle this, we built an `HtmlBaseTag` class that supports all of the standard HTML pass-through attributes. Our tag handler, `CommandRolloverButtonTag`,

subclasses `HtmlBaseTag`. We also provided a validator for the tag to make sure that the proper attributes were used together.

The work of writing the renderer itself can be simplified by wrapping the existing `Button` renderer, which is what we did with the `RolloverButtonDecorator` Renderer. However, the new renderer still requires the same level of JSP integration, since you cannot reliably subclass standard tag handler implementations.

Regardless of the method, the result is an easy-to-use button with support for JavaScript rollovers. As a matter of fact, the JSP tag is a drop-in replacement for the standard `<h:commandButton>` tag.

Now that we've examined some simpler components that perform their own rendering, and a new renderer for a standard component, it's time to look at a more complicated component in the next chapter.

18

UIHeadlineViewer: a composite, data-aware component

This chapter covers

- Working with RSS feeds
- Writing a DataModel class
- ISubclassing UIData
- Building a composite component

RSS is all the rage these days. It stands for Really Simple Syndication, RDF Site Summary, or nothing, depending on who you're talking to. Basically, it's an XML-based format for syndicating the contents, or headlines, of a web site. It's commonly used by weblogs ("blogs") and news sites. Just about every web development book I've read recently includes an example of consuming RSS feeds, so I figured this book should be just as hip as the others. In this chapter, we'll build a component that displays RSS feeds from any site on the Internet that publishes such a feed (there are tens of thousands available; see Syndic8 [Syndi8] for a comprehensive directory). Our component, UIHeadlineViewer is shown in figure 18.1.

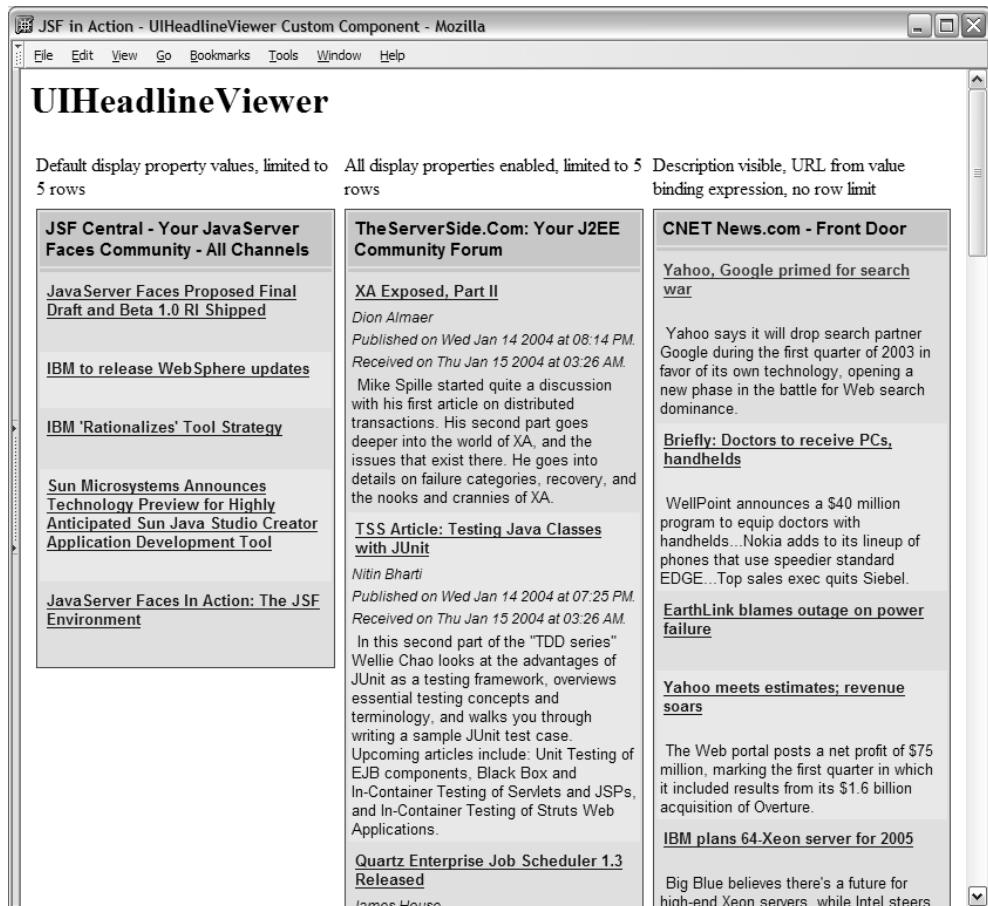


Figure 18.1 UIHeadlineViewer is a component that displays external RSS feeds.

NOTE Some portions of listings in this chapter have been omitted. You can download the full source code for this book from the *JSF in Action* web site at <http://www.manning.com/mann>.

You can see from the figure that it'd be pretty easy to add a `UIHeadlineViewer` to an existing page in your web site. This is a common thing to do—many sites display headlines from other sites, often on their front page. If you have an API that handles the complexities of parsing RSS files, it's entirely possible to display headlines using the standard `UIData` with the `Table` renderer. However, doing so takes a bit of work, especially if you don't know how you want the table to be displayed.

In cases where you want the features of a standard component but you want a specific set of child components or properties, it makes sense to subclass that component and then add the appropriate child components in code. We'll call this a composite component. `UIHeadlineViewer` subclasses `UIData` but adds child components to display the item title, description, and other attributes. Since it subclasses a standard component, we can use it with the standard `Table` renderer, as opposed to developing our own. Using `UIHeadlineViewer` is substantially simpler than using the vanilla `UIData` component.

Because `UIHeadlineViewer` subclasses `UIData`, it's data-aware—it operates on `DataModel` objects. JSF includes `DataModel` objects that wrap collections, result sets, and individual objects. For `UIHeadlineViewer`, we'll create a `DataModel` subclass called `ChannelDataModel` that consumes RSS feeds. Figure 18.2 shows the elements involved with implementing `UIHeadlineViewer`.

Before we delve into the details of implementing these classes, let's examine the underlying API we'll be using to process RSS feeds.

18.1 RSS and the Informa API

Conceptually, RSS feeds are simple. Feeds are organized into channels, and a channel can contain individual items with properties like `title`, `pubDate`, and `description`. A web site may produce multiple channels, such as one for all headlines, and others for specialized topics. For example, listing 18.1 shows a fragment of an RSS published by JSF Central [JSF Central], which is a JSF community site.

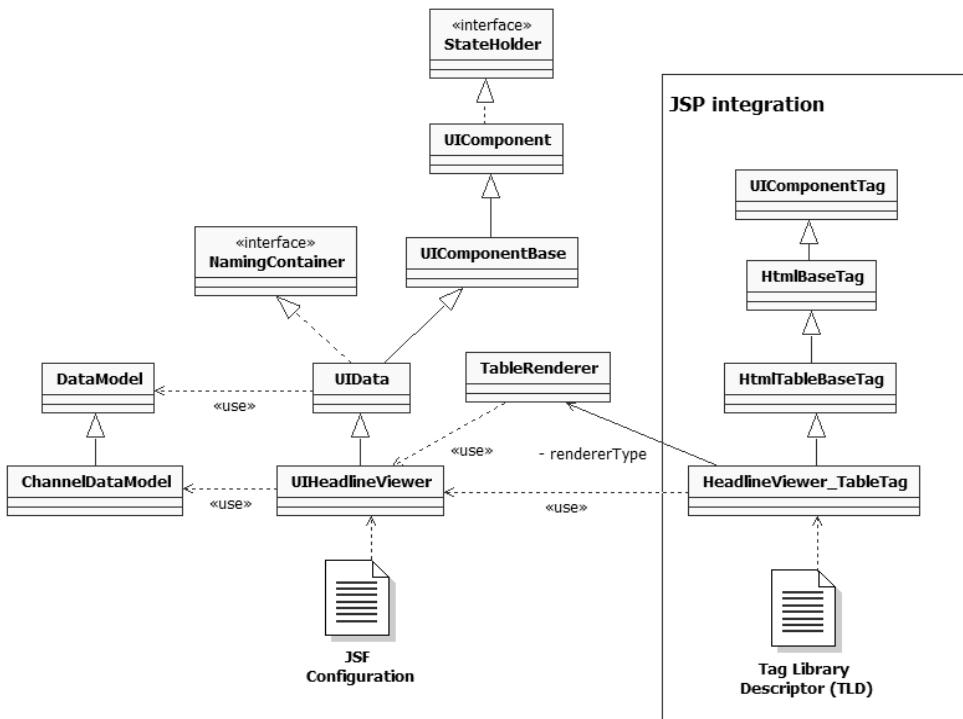


Figure 18.2 Classes and configuration files for the **UIHeadlineViewer** component. The component subclasses **UIData**, and is used with the standard **Table** renderer. It also makes use of a **DataModel** subclass, called **ChannelDataManager**. There is one custom tag handler, **HeadlineViewer_TableTag**, that subclasses **HtmlTableBaseTag** (which has all of the HTML pass-through properties for an HTML table).

Listing 18.1 A fragment of an RSS feed from JSF Central

```
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:dc="http://purl.org/dc/elements/1.1/" version="2.0">
  <channel>
    <title>JSF Central - Your JavaServer Faces Community - News</title>
    <link>http://www.jsfcentral.com</link>
    <description>
      JSFCentral.com is a community dedicated to providing developer
      resources, FAQ, news, and information about JavaServer Faces
      technology.
    </description>
    <copyright>
      Copyright (C) 2003 Virtua, Inc. All Rights Reserved. Java,
      JavaServer Faces, and all Java-based marks are trademarks or
      registered trademarks of Sun Microsystems, Inc. in the United
      States and other countries. Virtua, Inc. is independent of Sun
    </copyright>
  </channel>
</rss>
```

```
    Microsystems, Inc.  
</copyright>  
<language>en-us</language>  
<managingEditor>  
    Kito D. Mann (kmann@jsfcentral.com)  
</managingEditor>  
<webMaster>JSFCentral (webmaster@jsfcentral.com) </webMaster>  
<pubDate>22 Dec 2003 13:45 EST</pubDate>  
<item>  
<title>  
    JavaServer Faces Proposed Final Draft and Beta 1.0 RI Shipped  
</title>  
<link>http://www.jsfcentral.com#1700</link>  
<description>  
    Ed Burns announced Friday at 11:00 PM that the newest version  
    of the JavaServer Faces spec has been released. There were many  
    changes to the spec.  
</description>  
<dc:publisher>TheServerSide.com</dc:publisher>  
<dc:creator>Bill Dudney</dc:creator>  
<pubDate>21 Dec 2003 06:11 PST</pubDate>  
</item>  
<item>  
<title>IBM to release WebSphere updates</title>  
<link>http://www.jsfcentral.com#1600</link>  
<description>  
    Sutor said the new version will also add beta support for  
    JavaServer Faces, a proposed standard being developed through  
    the Java Community Process, an organization that Sun Microsystems  
    Inc. established to evolve Java technology. Using the programming  
    model that JavaServer Faces defines, developers can assemble  
    reusable interface components in a Web page and connect them to  
    data sources.  
</description>  
<dc:publisher>Computerworld</dc:publisher>  
<dc:creator>Carol Sliwa</dc:creator>  
<pubDate>15 Dec 2003 00:00 EST</pubDate>  
</item>  
</channel>  
</rss>
```

What makes RSS parsing complicated is the fact that there are several versions, and they all have their particular nuances. Different sites use different versions; some use several. In order to handle all of the complexities of supporting different feeds, it makes sense to use a third-party library. Such a library can normalize all of the feeds into a common object model, which makes our lives easier. For this example, we'll use the open source Informa API [Informa]. Informa has a lot of

useful features, like text searching and persistence. However, for our purposes, we'll just touch on the basics of the API that deal with consuming feeds from URLs and managing them.

NOTE The code in this chapter is based on Informa v0.45. Newer versions may differ.

An RSS channel is represented in Informa by the `ChannelIF` interface. Each `ChannelIF` has properties like `updatedDate`, `title`, `creator`, `description`, and so on. It also has a list of the channel's individual items, which are implemented as `ItemIF` instances. Each `ItemIF` has properties like `title`, `creator`, `date`, `found`, and `description`. `ChannelIF` objects are created by classes that implement the `ChannelBuilderIF` interface. A `ChannelBuilderIF` implementation persists the channel in some way—either in memory or in a data store. You can retrieve a new `ChannelIF` instance from the `RSSParser` class:

```
ChannelIF channel =
    RSSParser.parse(new ChannelBuilder(),
        "http://www.jsfcentral.com/jsfcentral_news.rss");
```

Here, we use `RSSParser` to create a new `ChannelIF` instance based on a new `ChannelBuilder` instance and a specific URL. `ChannelBuilder` is a concrete implementation of `ChannelBuilderIF` that stores new channels in memory.

Once we've retrieved a `ChannelIF` instance, we can access all of the expected channel properties, and also the list of items for the channel:

```
System.out.println("Channel Title: " + channel.getTitle());
Collection items = channel.getItems();
Iterator iterator = items.iterator();
while (iterator.hasNext())
{
    ItemIF item = (ItemIF)iterator.next();
    System.out.println();
    System.out.println("Item Title: " + item.getTitle());
    System.out.println("Item Creator: " + item.getCreator());
    System.out.println("Item Date: " + item.getDate());
    System.out.println("Item Found: " + item.getFound());
    System.out.println("Item Description: " + item.getDescription());
}
```

This code retrieves a list of all items in the channel, and then outputs a few properties of each one. It produces the following output:

```
Channel Title: JSF Central - Your JavaServer Faces Community - News
```

```
Item Title: JavaServer Faces Proposed Final Draft and Beta 1.0 RI  
Shipped  
Item Creator: Bill Dudney  
Item Date: Sun 21 Dec 2003 06:11 GMT-08:00 2003  
Item Found: Mon Jan 12 20:25:04 GMT-05:00 2004  
Item Description: Ed Burns announced Friday at 11:00 PM that the  
newest version of the JavaServer Faces spec has been released.  
There were many changes to the spec.  
  
Item Title: IBM to release WebSphere updates  
Item Creator: Carol Sliwa  
Item Date: Mon 15 Dec 2003 00:00:00 GMT-05:00 2003  
Item Found: Mon Jan 12 20:25:04 GMT-05:00 2004  
Item Description: Sutor said the new version will also add beta  
support for JavaServer Faces, a proposed standard being developed  
through the Java Community Process, an organization that Sun  
Microsystems Inc. established to evolve Java technology. Using the  
programming model that JavaServer Faces defines, developers can  
assemble reusable interface components in a Web page and connect  
them to data sources.
```

...

Note that this matches the content of the RSS feed shown in listing 18.1.

Using the in-memory ChannelBuilder and directly calling RSSParser.parse works fine for this simple example, but if you're constantly displaying the same feed (especially in a web application), reloading an unchanged feed is an unnecessary performance hit. Informa has a FeedManager class that caches ChannelIF instances, and refreshes them only when something has changed, or if the feed is out of date. Here's an example of retrieving a ChannelIF instance using FeedManager instead of RSSParser:

```
FeedManager manager = new FeedManager();  
FeedIF feed = manager.addFeed(  
    "http://www.jsfcentral.com/jsfcentral_news.rss");  
ChannelIF channel = feed.getChannel();
```

By default, FeedManager uses the ChannelBuilder class internally for managing the feeds in memory. The addFeed method returns a FeedIF instance, which provides additional metadata about a channel. We're more interested in the ChannelIF instance itself, which can be retrieved from FeedIF's getChannel method. This code generates the same output as the previous snippet, except that the feed won't be reloaded unless it's necessary.

These are the only classes we'll need to implement UIHeadlineViewer, although it's entirely possible to use more portions of the API to enhance the component's functionality and to develop rich, non-JSF functionality.

18.2 Using UIData with Informa

As we stated previously, you can use the raw Informa API with `UIData` to create the output shown in figure 18.1. This is possible because `UIData` can handle `List` objects directly. Let's say we had the following code:

```
FacesContext context = FacesContext.getCurrentInstance();
Application app = context.getApplication();
ChannelIF channel =
    RSSParser.parse(new ChannelBuilder(),
        "http://www.theserverside.com/rss/theserverside-1.0.rdf");
app.createValueBinding("#{requestScope.channel}").
    setValue(context, channel);

ArrayList itemList = new ArrayList(channel.getItems());
app.createValueBinding("#{requestScope.channelItems}").
    setValue(context, itemList);
```

Here, we create a new `ChannelIF` instance for the RSS feed at TheServerSide.com [TheServerSide], which is a great J2EE community. We store the `ChannelIF` instance itself under the key `channel` in the request scope. This will be displayed in the table's header. We also store its items under the separate key `channelItems` in request scope as well, to be displayed in the table's rows. Note that we create a new `ArrayList` instance as opposed to storing the items directly. This is because `ChannelIF`.`getItems` returns a `Collection`, and `UIData` can handle `Lists` but not `Collections`.

With the channel and its items stored in the request, we can set up a `UIData` instance that references them and achieves our desired appearance using the JSP shown in listing 18.2.

Listing 18.2 Using a `UIData` to display an RSS channel stored by the Informa API

```
<h:dataTable headerClass="hviewer-channel-title"
    rowClasses="hviewer-item-even, hviewer-item-odd"
    rows="5"
    styleClass="hviewer"
    value="#{channelItems}"
    var="item">

    <f:facet name="header">
        <h:outputText value="#{channel.title}" />
    </f:facet>
    <h:column>
        <h:panelGrid columns="1" cellpadding="2" cellspacing="0"
            headerClass="hviewer-item-title"
            columnClasses="hviewer-item-header">
            <f:facet name="header">
                <h:outputLink value="#{item.link}">
```

```
<h:outputText value="#{item.title}" />
</h:outputLink>
</f:facet>
<h:outputText value="#{item.creator}" />
<h:outputText value="#{item.date}" >
    <f:convertDateTime
        pattern="'Published on
            ' EEE MMM dd yyyy 'at' hh:mm a.'/>
</h:outputText>
<h:outputText value="#{item.found}" >
    <f:convertDateTime
        pattern="'Received on
            ' EEE MMM dd yyyy 'at' hh:mm a.'/>
</h:outputText>
<h:outputText value="#{item.description}"
    styleClass="hviewer-item-description">
</h:outputText>
</h:panelGrid>
</h:column>
</h:dataTable>
```

The code in listing 18.2 outputs a table with a single column. The header has a `UIOutput` that displays the channel's title, but the `UIData` control itself is associated with `channelItems`, which is the `ArrayList` of `ItemIF` instances. As the component iterates through the list, each item will be stored under the key `item`. The embedded panel displays a single item, using a header for the item's title and `UIOutput` components for individual fields. The code also uses the `DateTime` converter to format the date fields properly for display.

Using `UIData` this way works just fine, but it requires a bit of work. Not only do you have to know the Informa API, but a specific component configuration is harder to reuse in multiple places on the same page, or on different pages. You can always create a separate JSP file and include it when necessary, but that makes parameterization more difficult, especially if you want a different URL and perhaps different styles on different pages. (Parameterization is easier, however, with JSP 2.0 tag files.)

Our `UIHeadlineViewer` component will encapsulate all of the components that are declared in listing 18.2. All front-end developers will need to do is specify CSS styles and which fields they want visible. They won't have to worry about the Informa API either; all they'll need is the URL of the RSS feed. Our component will also optionally cache feeds. Let's start building it by subclassing `DataModel`, which `UIData` uses to abstract the underlying data format.

18.3 Subclassing DataModel

In the example in the previous section, we associated `UIData`'s value with an `ArrayList` created from the `Collection` of items that the channel returned. In general, this is a reasonable way to associate new data types with `UIData` components. When you set `UIData`'s value to equal a `List`, it internally creates an instance of `ListDataModel`, which is a subclass of `DataModel`. `DataModel` is an abstract class that represents a standard way to access different rows of data. JSF also has `DataModel` subclasses that wrap arrays (`ArrayDataModel`), JSTL Result objects (`ResultDataModel`), JDBC ResultSets (`ResultSetDataModel`), and any other individual object (`ScalarDataModel`). All of these are in the `javax.faces.model` package.

For times when converting your items to a list isn't enough, it's sometimes necessary to write your own `DataModel` subclass. Doing so gives you more control over how data navigation behaves, insulates the component from API changes, and also adds room for extra features like caching. In this section, we'll build a `DataModel` instance that works with `ChannelIF` instances, called `ChannelDataModel`. `ChannelDataModel` isn't a whole lot different than `ListDataModel` under the covers, but it should give you an idea about how easy it is to create your own `DataModel` subclasses.

The first step is to create two constructors—a zero argument constructor (required), and one that accepts the wrapped data type:

```
public ChannelDataModel()
{
    this(null);
}

public ChannelDataModel(ChannelIF channel)
{
    super();
    rowIndex = -1;
    setWrappedData(channel);
}
```

Our data type is a `ChannelIF` instance, so the second constructor accepts that type as an argument. `rowIndex` is an instance variable that represents the current row. `setWrappedData` is the mutator for the `wrappedData` property, which is one of the abstract `DataModel` properties we must implement:

```
public void setWrappedData(Object data)
{
    if (data != null)
    {
        if (!(data instanceof ChannelIF))
        {
            throw new IllegalArgumentException(
                "Wrapped data must be an instance of ChannelIF");
        }
    }
}
```

```

        "Only ChannelIF instances can be wrapped; " +
        " received a " + data.getClass().getName() +
        " instance instead.");
    }
    channel = (ChannelIF)data;
    items = new ArrayList(channel.getItems());
    setRowIndex(0);
}
else
{
    channel = null;
    items = null;
}
}

public Object getWrappedData()
{
    return channel;
}

```

For the setter method, if the data type isn't a `ChannelIF` instance, we throw a new `IllegalArgumentException`, since this class doesn't know how to handle any other types. The method's goal is to set up the `channel` and `items` instance variables, and set the row index to 0 (the first row). Note that we store the items as an `ArrayList`, even though `ChannelIF.getItems` returns a `Collection`. This is because we need to access an individual item by its index, which isn't possible with a `Collection`.

We also need to implement the `rowIndex` property:

```

public void setRowIndex(int rowIndex)
{
    if (rowIndex < -1)
    {
        throw new IllegalArgumentException(
            "rowIndex must be -1 or greater.");
    }
    if (channel != null && this.rowIndex != rowIndex)
    {
        this.rowIndex = rowIndex;
        Object rowData = null;
        if (isRowAvailable())
        {
            rowData = getRowData();
        }
        DataModelListener[] listeners = getDataModelListeners();
        for (int i = 0; i < listeners.length; i++)
        {
            listeners[i].rowSelected(
                new DataModelEvent(this, rowIndex, rowData));
        }
    }
}

```

```

}

public int getRowIndex()
{
    return rowIndex;
}

```

In `setRowIndex`, we only begin processing if the channel isn't null, the new value is greater than -1, and the new value is different than the old value. Then, we update the `rowIndex` instance variable and broadcast a new `DataModelEvent` to all interested `DataModelListeners`. (`DataModel` provides a full implementation of `getDataModelListeners` as well as `addDataModelListener` and `removeDataModelListener`.) Supporting `DataModelListeners` is one of the key `DataModel` requirements; this is necessary for developers to perform actions based on the currently selected row. (The `setRowIndex` method is actually called anytime a renderer iterates through the data set; it's not analogous to the user selecting a row.)

`availableRow` and `rowData` are read-only properties that must be implemented by `DataModel` subclasses. Here's `isRowAvailable`:

```

public boolean isRowAvailable()
{
    return (channel != null && rowIndex > -1 &&
            rowIndex < items.size());
}

```

If the channel isn't null, and the `rowIndex` is between 0 and the maximum size of the list, the currently selected row is available.

The `rowData` property is trivial as well:

```

public Object getRowData()
{
    if (channel == null)
    {
        return null;
    }
    return items.get(rowIndex);
}

```

If the channel isn't null, we just return the item available at the current index.

The final property is `rowCount`, which is yet another wrapper for a `List` method:

```

public int getRowCount()
{
    if (channel == null)
    {
        return -1;
    }
    return items.size();
}

```

If the channel is null, we return -1; otherwise, we just return the size of the list.

That's it for ChannelDataModel. For the full source, see the book's web site. Given the simplicity of this process, it's no surprise that the JSF specification refers to DataModel objects as wrappers for other data types. In essence, they're adapters [GoF]—they adapt a specific data type into a known type that UIData (and any other custom components that you or third parties develop) can understand.

Now that we've completed our ChannelDataModel tour, let's move on to the important piece of the puzzle: the UIHeadlineViewer component itself.

18.4 Writing the UIHeadlineViewer class

Our goal with UIHeadlineViewer is essentially to provide a customized version of UIData that can load feeds based solely on the feed's URL. Writing UIHeadlineViewer (in the package `jia.components`) requires the following steps:

- 1 Subclass UIData.
- 2 Create and add child components that mimic the structure defined by the JSP in listing 18.2. That structure is a tree of components, as shown in figure 18.3. Rather than defining these components in JSP, we'll create them in code and add them when our component is created.

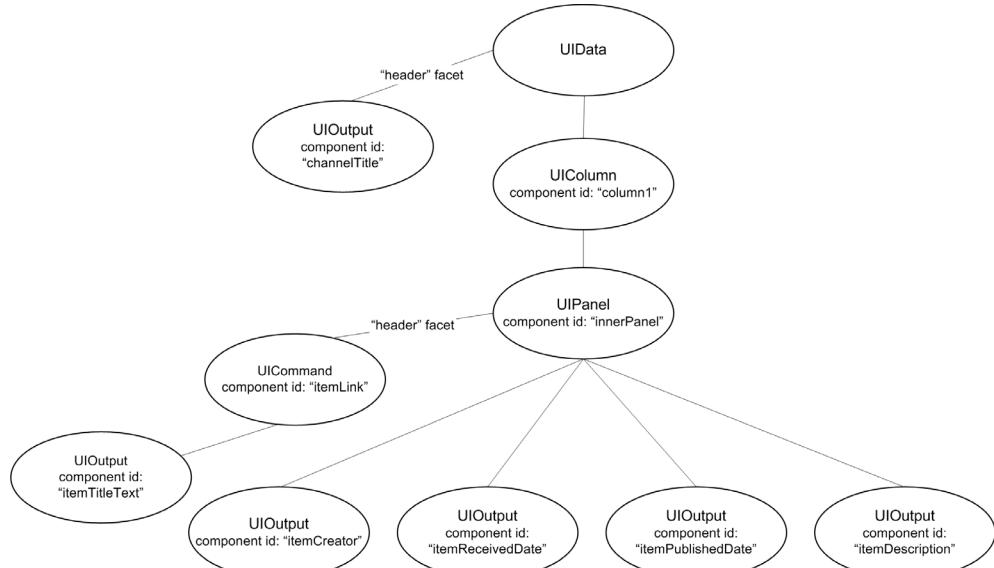


Figure 18.3 `UIHeadlineViewer` is a composite component. It's a `UIData` subclass with a facet and several customized child components.

For each of the primary child components, we'll expose properties that indicate whether they should be visible. This will allow us to easily control whether `UIHeadlineViewer` displays an item's description, published date, creator, and so on. We'll also expose a `URL` property that we'll use to create the `ChannelDataModel` instance that is stored as the component's value. These properties are listed in table 18.1.

Table 18.1 `UIHeadlineViewer` has properties that control which items should be visible.

Property	Type	Description	Default Value
<code>showChannelTitle</code>	<code>boolean</code>	True if the channel title should be displayed.	<code>true</code>
<code>showItemTitle</code>	<code>boolean</code>	True if the item title should be displayed.	<code>true</code>
<code>showItemCreator</code>	<code>boolean</code>	True if an item's creator should be displayed.	<code>false</code>
<code>showItem-ReceivedDate</code>	<code>boolean</code>	True if an item's received date should be displayed.	<code>false</code>
<code>showItem-PublishedDate</code>	<code>boolean</code>	True if an item's published date should be displayed.	<code>false</code>
<code>URL</code>	<code>String</code>	The URL for the RSS feed.	<code>null</code>

The code for `UIHeadlineViewer` is shown in listing 18.3.

Listing 18.3 `UIHeadlineViewer.java`: a `UIData` subclass that displays RSS feeds

```
package org.jia.components;

import org.jia.components.model.ChannelDataModel;
import org.jia.util.Util;

import javax.faces.application.Application;
import javax.faces.component.*;
import javax.faces.context.FacesContext;
import javax.faces.convert.DateTimeConverter;
import javax.faces.el.ValueBinding;

import de.nav.a.informa.core.ChannelIF;
import de.nav.a.informa.impl.basic.ChannelBuilder;
import de.nav.a.informa.parsers.RSSParser;
import de.nav.a.informa.utils.FeedManager;

public class UIHeadlineViewer extends UIData {  

    public final static String COMPONENT_TYPE = "jia.HeadlineViewer";
```

Subclass
1 `UIData`

2 Declare new
component type

```

private UIPanel innerPanel;
private UIOutput itemDescription;
private UIOutput itemPublishedDate;
private UIOutput itemReceivedDate;
private UIOutput itemTitle;
private UIOutput channelTitle;
private UIOutput itemCreator;

private String url;
private Boolean showChannelTitle;
private Boolean showItemCreator;
private Boolean showItemDescription;
private Boolean showItemPublishedDate;
private Boolean showItemReceivedDate;
private Boolean showItemTitle;

public UIHeadlineViewer()
{
    super();
    this.setVar("item");      ←③ Set the row
    addChildrenAndFacets();   variable to "item"
}

// Protected methods

protected void addChildrenAndFacets()
{
    Application app = FacesContext.
        getCurrentInstance().getApplication();

    setChannelTitle(createChannelTitle(app));

    UIColumn column =
        (UIColumn)app.createComponent(
            UIColumn.COMPONENT_TYPE);
    column.setId("column1");
    getChildren().add(column);

    setInnerPanel(createInnerPanel(app));

    setTitle(createItemTitle(app));
    setItemCreator(createItemCreator(app));
    setItemPublishedDate(
        createItemPublishedDate(app));
    setItemReceivedDate(
        createItemReceivedDate(app));
    setDescription(
        createItemDescription(app));
}

// Component creation methods

```

4
Add child
components
and facets

5
Components
for each item

```

protected UIOutput createChannelTitle(Application app)
{
    UIOutput channelTitle =
        (UIOutput)app.createComponent(UIOutput.COMPONENT_TYPE);
    channelTitle.setId("channelTitle");
    return channelTitle;
} ← 6 Give all components an identifier

protected UIPanel createInnerPanel(Application app)
{
    UIPanel innerPanel =
        (UIPanel)app.createComponent(UIPanel.COMPONENT_TYPE);
    innerPanel.setId("innerPanel");
    innerPanel.setRendererType(
        "javax.faces.Grid");
    return innerPanel;
} ↗ 7 Set renderer type if necessary

protected UIOutput createItemTitle(Application app)
{
    UIOutput itemTitle =
        (UIOutput)app.createComponent(UIOutput.COMPONENT_TYPE);
    itemTitle.setId("itemTitle");
    itemTitle.setRendererType("javax.faces.Link");
    itemTitle.setValueBinding("value",
        app.createValueBinding("#{item.link}"));
} ↗ 8 Set value-binding for row components

UIOutput itemTitleText =
    (UIOutput)app.createComponent(UIOutput.COMPONENT_TYPE);
itemTitleText.setValueBinding("value",
    app.createValueBinding("#{item.title}"));
itemTitle.getChildren().add(itemTitleText);

return itemTitle;
}

protected UIOutput createItemCreator(Application app)
{
    UIOutput itemCreator =
        (UIOutput)app.createComponent(UIOutput.COMPONENT_TYPE);
    itemCreator.setId("itemCreator");
    itemCreator.setValueBinding("value",
        app.createValueBinding("#{item.creator}"));
    return itemCreator;
}

protected UIOutput createItemPublishedDate(Application app)
{
    UIOutput itemPublishedDate =
        (UIOutput)app.createComponent(UIOutput.COMPONENT_TYPE);
    itemPublishedDate.setId("itemPublishedDate");
}

```

```

itemPublishedDate.setValueBinding("value",
    app.createValueBinding("#{item.date}"));

DateTimeConverter converter =
    new DateTimeConverter();
converter.setPattern(
    "'Published on ' EEE MMM dd yyyy " +
    "'at' hh:mm a.");
itemPublishedDate.setConverter(converter);

return itemPublishedDate;
}

protected UIOutput createItemReceivedDate(Application app)
{
    UIOutput itemReceivedDate =
        (UIOutput)app.createComponent(UIOutput.COMPONENT_TYPE);
    itemReceivedDate.setId("itemReceivedDate");
    itemReceivedDate.setValueBinding("value",
        app.createValueBinding("#{item.found}"));

    DateTimeConverter converter = new DateTimeConverter();
    converter.setPattern(
        "'Received on ' EEE MMM dd yyyy 'at' hh:mm a.");
    itemReceivedDate.setConverter(converter);

    return itemReceivedDate;
}

protected UIOutput createItemDescription(Application app)
{
    UIOutput itemDescription =
        (UIOutput)app.createComponent(UIOutput.COMPONENT_TYPE);
    itemDescription.setId("itemDescription");
    itemDescription.setValueBinding("value",
        app.createValueBinding("#{item.description}"));
    return itemDescription;
}

// Embedded component properties

public UIOutput getChannelTitle()
{
    return channelTitle;
}

protected void setChannelTitle(UIOutput channelTitle)
{
    this.channelTitle = channelTitle;
    getFacets().put("header", channelTitle); ←⑩ Add channel
    title as a facet
}

```

⑨ Create and add converters for dates

⑩ Add channel title as a facet

```
public UIPanel getInnerPanel()
{
    return innerPanel;
}

protected void setInnerPanel(UIPanel innerPanel)
{
    this.innerPanel = innerPanel;
    findComponent("column1").getChildren().add(innerPanel); | 11 Add inner panel
                                                       as a child of the
                                                       column

public UIOutput getItemTitle()
{
    return itemTitle;
}

protected void setItemTitle(UIOutput itemTitle)
{
    this.itemTitle = itemTitle;
    innerPanel.getFacets().put("header", itemTitle); | 12 Add item title as
                                                       a facet of the
                                                       inner panel
}

public UIOutput getItemCreator()
{
    return itemCreator;
}

protected void setItemCreator(UIOutput itemCreator)
{
    this.itemCreator = itemCreator;
    innerPanel.getChildren().add(itemCreator); | 13 Add per-item
                                                       components
}

public UIOutput getItemPublishedDate()
{
    return itemPublishedDate;
}

protected void setItemPublishedDate(UIOutput itemPublishedDate)
{
    this.itemPublishedDate = itemPublishedDate;
    innerPanel.getChildren().add(itemPublishedDate);
}

public UIOutput getItemReceivedDate()
{
    return itemReceivedDate;
```

```
}

protected void setItemReceivedDate(UIOutput itemReceivedDate)
{
    this.itemReceivedDate = itemReceivedDate;
    innerPanel.getChildren().add(itemReceivedDate);
}

public UIOutput getItemDescription()
{
    return itemDescription;
}

protected void setItemDescription(UIOutput itemDescription)
{
    this.itemDescription = itemDescription;
    innerPanel.getChildren().add(itemDescription);
}

// UIComponent methods

public void encodeBegin(FacesContext context)
    throws java.io.IOException
{
    ChannelDataModel data =
        (ChannelDataModel)getValue();
    if (data != null)
    {
        ChannelIF channel =
            (ChannelIF)data.getWrappedData();
        getChannelTitle().setValue(
            channel.getTitle());
    }
    channelTitle.setRendered
        (getShowChannelTitle());
    itemTitle.setRendered(
        getShowItemTitle());
    itemCreator.setRendered(
        getShowItemCreator());
    itemDescription.setRendered
        (getShowItemDescription());
    itemPublishedDate.setRendered
        (getShowItemPublishedDate());
    itemReceivedDate.setRendered(
        getShowItemReceivedDate());

    super.encodeBegin(context);
}

// Other properties
```

14 Set the value
of the title

15 Set the
properties
before display

```
public String getURL()
{
    return url;
}

public void setURL(String url)
{
    if (url != null)
    {
        ValueBinding binding =
            getFacesContext().getApplication().
                createValueBinding(
                    "#{@UIHeadlineViewerFeedManager}");
        FeedManager manager =
            (FeedManager)binding.getValue(
                getFacesContext());
        ChannelIF channel = null;
        try
        {
            if (manager != null)
            {
                channel = manager.addFeed(url).
                    getChannel();
            }
            else
            {
                channel = RSSParser.parse(
                    new ChannelBuilder(), url);
            }
        }
        catch (Exception e)
        {
            throw new FacesException(
                "Error creating channel from URL", e);
        }
        this.url = url;
        setValue(new ChannelDataModel(channel));
    }
}

public boolean getShowChannelTitle()
{
    return Util.getBooleanProperty(this,
        showChannelTitle,
        "showChannelTitle", true);
}

public void setShowChannelTitle(
    boolean showChannelTitle)
{
```

16 Create new
ChannelDataModel
instance

17 Check for
expressions

```
        this.showChannelTitle =
            new Boolean(showChannelTitle);
    }

    public boolean getShowItemTitle()
    {
        return Util.getBooleanProperty(this, showItemTitle,
                                       "showItemTitle", true);
    }

    public void setShowItemTitle(boolean showItemTitle)
    {
        this.showItemTitle = new Boolean(showItemTitle);
    }

    public boolean getShowItemCreator()
    {
        return Util.getBooleanProperty(this, showItemCreator,
                                       "showItemCreator", false);
    }

    public void setShowItemCreator(boolean showItemCreator)
    {
        this.showItemCreator = new Boolean(showItemCreator);
    }

    public boolean getShowItemPublishedDate()
    {
        return Util.getBooleanProperty(this, showItemPublishedDate,
                                       "showItemPublishedDate", false);
    }

    public void setShowItemPublishedDate(
                boolean showItemPublishedDate)
    {
        this.showItemPublishedDate =
            new Boolean(showItemPublishedDate);
    }

    public boolean getShowItemReceivedDate()
    {
        return Util.getBooleanProperty(this, showItemReceivedDate,
                                       "showItemReceivedDate", false);
    }

    public void setShowItemReceivedDate(boolean showItemReceivedDate)
    {
        this.showItemReceivedDate = new Boolean(showItemReceivedDate);
    }
```

```
public boolean getShowItemDescription()
{
    return Util.getBooleanProperty(this, showItemDescription,
                                   "showItemDescription", false);
}

public void setShowItemDescription(boolean showItemDescription)
{
    this.showItemDescription = new Boolean(showItemDescription);
}

// StateHolder methods

public Object saveState(FacesContext context)
{
    Object[] values = new Object[8];
    values[1] = url;
    values[2] = super.saveState(context);
    values[3] = showChannelTitle;
    values[4] = showItemTitle;
    values[5] = showItemCreator;
    values[6] = showItemPublishedDate;
    values[7] = showItemReceivedDate;
    values[8] = showItemDescription;

    return values;
}

public void restoreState(FacesContext context,
                        Object state)
{
    Object[] values = (Object[])state;
    super.restoreState(context, values[0]);
    url = (String)values[1];
    showChannelTitle = (Boolean)values[2];
    showItemTitle = (Boolean)values[3];
    showItemCreator = (Boolean)values[4];
    showItemPublishedDate = (Boolean)values[5];
    showItemReceivedDate = (Boolean)values[6];
    showItemDescription = (Boolean)values[7];
}
```

18 Save and
restore new
properties

- ➊ Subclassing UIData gives us automatic support for displaying DataModel objects, and ensures that the Table renderer can display this component.
- ➋ Here, we declare the component's type. There is no need to change the family property; we inherit the javax.faces.Data family from UIData, which is what we want.

- ❸ The `var` property is the key under which each row is stored when iterating through rows in the `DataModel` object. Because each row in `ChannelDataModel` represents an RSS item (and more specifically, an instance of the `ItemIF` interface), we'll use the key `item` for each row. We'll reference this key later when we set the value-binding expressions for child components.
- ❹ This method creates all of the child components and facets. Note that we start by creating and adding a `UIColumn` instance. `UIData` requires that all components that need to be displayed for each row must be contained within a `UIColumn` instance. All of our child components are laid out within a `UIPanel` that is a child of this `UIColumn`. Once this method has completed, our component will represent a tree like the one shown in figure 18.3. All of the setters are responsible for placing the component in the proper tree position.
- ❺ Each of these components are displayed once for each item in the `ChannelDataModel`.
- ❻ Every time we create a new component, we set its component identifier. This makes it easy to find the component later, if necessary.
- ❼ If a component's default renderer doesn't provide the right functionality, we need to set a new one. The `innerPanel` component is used to lay out all of the components, so it requires a `Grid` renderer (the default renderer for `UIPanel` is `null`).
- ❽ For all of the components that display item values, we need to set a value-binding expression. Note that the key is `item`, which is the variable we defined in the constructor (❸). The second part of the expression, `link`, maps to the `link` property of the `ItemIF` interface, since an instance of `ItemIF` will be stored under the `item` key for each row.
- ❾ For components that display date values, we create and configure a new `Date-TimeConverter` instance.
- ❿ We store the `channelTitle` component as a header facet of the `UIHeadlineViewer` itself. This will be used by the `Table` renderer.
- ⓫ In order to be displayed, `innerPanel` must be a child of a `UIColumn`.
- ⓬ The `itemTitle` component is added as the header facet of `innerPanel` because it's the heading for each item iteration.
- ⓭ All of the other per-item components are simply added as children of `innerPanel`.
- ⓮ When `UIHeadlineViewer` displays itself, it starts by setting a few properties of its children and facets. Because the channel title is a property of the `ChannelIF` interface and not the `ItemIF` interface, we can't associate it with a value-binding expression that uses the `var` property (`item`). However, we can just manually set the value before display.
- ⓯ Before encoding, we also set the rendered properties of all of the components based on properties designed to control this behavior (⓯).

- ➏ The URL property is shorthand for the value property, so we create a new ChannelDataModel based on the URL. First, we check to see if there's a FeedManager instance stored under the key UIHeadlineViewerFeedManager. Remember, a FeedManager caches ChannelIF instances. (Usually, this would be a managed bean; see the next section for an example.) If one exists, we retrieve a ChannelIF instance from it; otherwise we forgo caching and retrieve a ChannelIF instance directly from the RSSParser class. Finally, we create a new ChannelDataModel instance and set it as the value of this component. Since the value is now a proper DataModel instance, we can safely reuse all of UIData's default behavior for displaying individual rows.
- ➐ All of the remaining properties are simple boolean properties. However, in order to support value-binding expressions, we store them internally as Boolean instances. This allows us to use a value-binding expression or default value if they haven't yet been set. The Util.getBooleanProperty returns the instance variable if it is non-null; otherwise it evaluates and returns its value-binding expression (if there is one):

```
public static boolean getBooleanProperty(UIComponent component,
                                         Boolean property, String key,
                                         boolean defaultValue)
{
    if (property != null)
    {
        return property.booleanValue();
    }
    else
    {
        ValueBinding binding =
            (ValueBinding)component.getValueBinding(key);
        if (binding != null)
        {
            Boolean value = (Boolean)binding.getValue(
                FacesContext.getCurrentInstance());
            if (value != null)
            {
                return value.booleanValue();
            }
        }
        return defaultValue;
    }
}
```

This is the norm for any properties that support value-binding expressions. If the property has been set, you return it. Otherwise, you evaluate its value-binding expression (if there is one).

- ⑯ For the StateHolder methods, we only need to worry about the new properties that `UIHeadlineViewer` defines. All child components and facets will be saved for us by the superclasses.

That's it for `UIHeadlineViewer`. There's no need to develop a custom renderer, since we'll be using the standard `Table` renderer. Consequently, we can skip ahead to the configuration details.

18.5 Registering the component

Configuring `UIHeadlineViewer` requires two entries in a JSF configuration file: the typical `<component>` entry, and an optional `<managed-bean>` entry for configuring the `FeedManager`:

```
<component>
    <description>
        Displays RSS feeds from a given URL.
    </description>
    <display-name>HeadlineViewer</display-name>
    <component-type>jia.HeadlineViewer</component-type>
    <component-class>
        org.jia.components.UIHeadlineViewer
    </component-class>
</component>
<managed-bean>
    <description>
        FeedManager for use with UIHeadlineViewer
    </description>
    <managed-bean-name>
        UIHeadlineViewerFeedManager
    </managed-bean-name>
    <managed-bean-class>
        de.nava.informa.utils.FeedManager
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
```

Note that we place the `FeedManager` in the application scope, and that the bean's name matches the name we used in the `UIHeadlineViewer.setURL` method (see section 18.4). In a real application, you may want to initialize `FeedManager` in more complicated ways, perhaps taking advantage of some of Informa's persistence features. Our new component, however, is blissfully unaware of such details.

Configuration is usually the simplest part of the equation; now let's look at JSP integration.

18.6 JSP integration

In order to integrate this component with JSP, all that's necessary is a new base tag handler for additional HTML properties, the main tag handler, and entry in our custom tag library.

18.6.1 Writing the JSP custom tag

We'll integrate UIHeadlineViewer with JSP through a custom tag handler called `jia.components.taglib.HeadlineViewerTableTag`. In order to support the same HTML pass-through properties that the Table renderer supports, we'll subclass the `jia.components.taglib.HtmlTableBaseTag` class. `HtmlTableBaseTag` is a subclass of `HtmlBaseTag`, which we developed in online extension chapter 17. It implements table-specific HTML pass-through attributes like `cellpadding` and `width`, in addition to the basic attributes that `HtmlBaseTag` provides.

Other than its superclass, `HeadlineViewerTableTag` is somewhat unique because some of its properties don't map directly to Table renderer attributes of the same name. It also exposes rendererer attributes that apply to child components of `UIHeadlineViewer`. This way, a front-end developer can specify a CSS style for different child components by simply specifying an attribute on the custom tag. Table 18.2 describes how the tag handler's properties map to attributes and properties of `UIHeadlineViwer`, its child components, and the Table renderer.

Table 18.2 HeadlineViewerTableTag exposes renderer attributes for UIHeadlineViewer and its child components.

Tag Property	For Component	Renderer Attribute
<code>styleClass</code>	<code>UIHeadlineViewer</code>	<code>styleClass</code>
<code>channelTitleClass</code>	<code>UIHeadlineViewer</code>	<code>headerClass</code>
<code>itemClasses</code>	<code>UIHeadlineViewer</code>	<code>rowClasses</code>
<code>itemTitleClass</code>	<code>UIHeadlineViewer.innerPanel</code>	<code>headerClass</code>
<code>itemHeaderClass</code>	<code>UIHeadlineViewer.itemCreator</code>	<code>styleClass</code>
<code>itemHeaderClass</code>	<code>UIHeadlineViewer.itemPublishedDate</code>	<code>styleClass</code>
<code>itemHeaderClass</code>	<code>UIHeadlineViewer.itemReceivedDate</code>	<code>styleClass</code>
<code>itemDescriptionClass</code>	<code>UIHeadlineViewer.itemDescription</code>	<code>styleClass</code>

As usual, all of `HeadlineViewerTableTag`'s specialized work is handled in the `setProperties` method; the grunt work is handled by the superclasses. The source

for is shown in listing 18.4; we've omitted most of the mundane getters and setters to save some trees.

Listing 18.4 HeadlineViewerTableTag.java: Tag handler for UIHeadlineViewer with the Table renderer

```
package org.jia.components.taglib;

import org.jia.components.UIHeadlineViewer;
import org.jia.util.Util;

import javax.faces.application.Application;
import javax.faces.component.UIComponent;
import javax.faces.component.UIPanel;
import javax.faces.context.FacesContext;

public class HeadlineViewerTableTag
    extends HtmlTableBaseTag
{
    private String url;
    private String styleClass;
    private String channelTitleClass;
    private String itemTitleClass;
    private String itemHeaderClass;
    private String itemDescriptionClass;
    private String itemClasses;
    private String showChannelTitle;
    private String showItemCreator;
    private String showItemDescription;
    private String showItemPublishedDate;
    private String showItemReceivedDate;
    private String showItemTitle;
    private String rows;

    public HeadlineViewerTableTag()
    {
        super();
    }

    // UIComponentTag methods

    public String getComponentType()
    {
        return UIHeadlineViewer.COMPONENT_TYPE;
    }

    public String getRendererType()
    {
        return "javax.faces.Table";
    }
}
```

1 Subclass **HtmlTableBaseTag**

2 Declare properties as **Strings**

3 Set component type for **UIHeadlineViewer**

4 Use the **Table renderer**

```
protected void setProperties(UIComponent component)
{
    super.setProperties(component);

    UIHeadlineViewer viewer = (UIHeadlineViewer)component;
    Application app = getFacesContext().getApplication();

    if (url != null)
    {
        if (isValueReference(url))
        {
            viewer.setURL((String)
                app.createValueBinding(url).
                    getValue(getFacesContext()));
        }
        else
        {
            viewer.setURL(url);
        }
    }
    if (showChannelTitle != null)
    {
        if (isValueReference(showChannelTitle))
        {
            viewer.setValueBinding(
                "showChannelTitle",
                app.createValueBinding(
                    showChannelTitle));
        }
        else
        {
            viewer.setShowChannelTitle(
                Boolean.valueOf(showChannelTitle).
                    booleanValue());
        }
    }
    if (showItemTitle != null)
    {
        if (isValueReference(showItemTitle))
        {
            viewer.setValueBinding("showItemTitle",
                app.createValueBinding(showItemTitle));
        }
        else
        {
            viewer.setShowItemTitle(
                Boolean.valueOf(showItemTitle).booleanValue());
        }
    }
    if (showItemCreator != null)
```

5 For the URL property, evaluate the expression first

6 For other properties, add bindings normally

```
{  
    if (isValueReference(showItemCreator))  
    {  
        viewer.setValueBinding("showItemCreator",  
                               app.createValueBinding(showItemCreator));  
    }  
    else  
    {  
        viewer.setShowItemCreator(  
            Boolean.valueOf(showItemCreator).booleanValue());  
    }  
}  
if (showItemPublishedDate != null)  
{  
    if (isValueReference(showItemPublishedDate))  
    {  
        viewer.setValueBinding("showItemPublishedDate",  
                               app.createValueBinding(showItemPublishedDate));  
    }  
    else  
    {  
        viewer.setShowItemPublishedDate(  
            Boolean.valueOf(showItemPublishedDate).booleanValue());  
    }  
}  
if (showItemReceivedDate != null)  
{  
    if (isValueReference(showItemReceivedDate))  
    {  
        viewer.setValueBinding("showItemReceivedDate",  
                               app.createValueBinding(showItemReceivedDate));  
    }  
    else  
    {  
        viewer.setShowItemReceivedDate(  
            Boolean.valueOf(showItemReceivedDate).booleanValue());  
    }  
}  
if (showItemDescription != null)  
{  
    if (isValueReference(showItemDescription))  
    {  
        viewer.setValueBinding("showItemDescription",  
                               app.createValueBinding(showItemDescription));  
    }  
    else  
    {  
        viewer.setShowItemDescription(  
            Boolean.valueOf(showItemDescription).booleanValue());  
    }  
}
```

```
if (rows != null)
{
    if (isValueReference(rows))
    {
        viewer.setValueBinding("rows",
                               app.createValueBinding(rows));
    }
    else
    {
        viewer.setRows(Integer.parseInt(rows));
    }
}

Util.addAttribute(app, viewer, "styleClass", styleClass);
Util.addAttribute(app, viewer, "headerClass", channelTitleClass);
Util.addAttribute(app, viewer, "rowClasses", itemClasses); 7

UIPanel panel = viewer.getInnerPanel();
panel.getAttributes().put("columns",
                          new Integer(1));
panel.getAttributes().put("cellpadding",
                          new Integer(2));
panel.getAttributes().put("cellspacing",
                          new Integer(0)); 8 Hard-code
renderer attributes
of inner panel

Util.addAttribute(app,
                 panel,
                 "headerClass",
                 itemTitleClass);
Util.addAttribute(app,
                 viewer.getItemCreator(),
                 "styleClass",
                 itemHeaderClass);
Util.addAttribute(app,
                 viewer.getItemPublishedDate(),
                 "styleClass",
                 itemHeaderClass);
Util.addAttribute(app,
                 viewer.getItemReceivedDate(),
                 "styleClass",
                 itemHeaderClass);
Util.addAttribute(app,
                 viewer.getItemDescription(),
                 "styleClass",
                 itemDescriptionClass);
}

public void release()
{
    url = null;
    styleClass = null;
```

9 Set attributes
of child
components

Set Table
renderer
attributes

```
channelTitleClass = null;
itemTitleClass = null;
itemHeaderClass = null;
itemDescriptionClass = null;
showChannelTitle = null;
showItemCreator = null;
showItemReceivedDate = null;
showItemTitle = null;
showItemDescription = null;
showItemPublishedDate = null;
rows = null;
itemClasses = null;
}

// Component properties

public String getUrl()
{
    return url;
}

public void setUrl(String url)
{
    this.url = url;
}
...
// Renderer attributes

public String getStyleClass()
{
    return styleClass;
}

public void setStyleClass(String styleClass)
{
    this.styleClass = styleClass;
}
...
}
```

-
- ➊ In order to support HTML pass-through attributes for tables, we subclass the `HtmlTableBaseTag` class, which supports HTML attributes for tables, and gives us basic component tag functionality.
 - ➋ In order to support value-binding expressions for all attributes, the instance variables for all of the properties are declared as `Strings`.
 - ➌ In order to associate this component with `UIHeadlineViewer`, we return the component type `UIHeadlineViewer.COMPONENT_TYPE`.

- ④ We want to use the standard `Table` renderer (normally used with `UIData`), so we return “`javax.faces.Table`” as the renderer type.
- ⑤ In `UIHeadlineViewer`, the `URL` property isn’t value-binding enabled. This is because the component tries to create a new `ChannelIF` based on the URL when the property is set. However, a front-end developer may want to express the `URL` via a value-binding expression, so we support it by evaluating it (if necessary) and then setting the `UIHeadlineViewer`’s `URL` property with the result.
- ⑥ All of the display properties are value-binding enabled, so we support them by creating and setting a new value-binding property if necessary. If the property isn’t value-binding enabled, we have to convert it to a boolean from a `String` before setting it. The `util.addAttribute` method sets the attribute if the tag handler’s property is non-null, creating and adding a `ValueBinding` if necessary.
- ⑦ Here, we set `Table` renderer attributes for `UIHeadlineViewer`. Note that the names don’t match exactly; the names of the tag handler’s properties make more sense for the type of component we’re displaying. (See table 18.2 for a list of how the tag handler handles renderer attributes.)
- ⑧ This code hardcodes some `Grid` renderer attributes for `UIHeadlineViewer`’s inner panel. We set these attributes here instead of in `UIHeadlineViewer` itself because this tag handler is designed to support attributes understood by the standard HTML render kit. The component itself doesn’t know which render kit it will be associated with; it only knows the renderer types, which could be valid for multiple render kits.
- ⑨ Here, we set renderer attributes of the child components. Note that we map the tag handler’s `itemHeaderClass` property to the `styleClass` attribute of several child components. See table 18.2 for a list of how the tag handler properties are mapped to attributes of the child components.

This completes our discussion of `UIHeadlineViewer`’s tag handler. Next, we’ll tell the web container about this tag by adding it to the tag library.

18.6.2 Adding the tag to the tag library

The tag handler entry for `HeadlineViewerTableTag` is straightforward. It has no required attributes, and includes all of the properties defined by the tag itself, and its superclasses (which include HTML table pass-through attributes). Portions of the code are shown in listing 18.5.

Listing 18.5 Tag handler entry for HeadlineViewerTableTag

```
<?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>
<!-- Tag Library Description Elements --&gt;
&lt;tlib-version&gt;1.0&lt;/tlib-version&gt;
&lt;jsp-version&gt;1.2&lt;/jsp-version&gt;
&lt;short-name&gt;JSF in Action Custom Tags&lt;/short-name&gt;
&lt;uri&gt;jsf-in-action-components&lt;/uri&gt;
&lt;description&gt;
Sample custom components, renderers, validators, and converters
from JSF in Action method.
&lt;/description&gt;
&lt;!-- Tag declarations --&gt;
...
&lt;tag&gt;
    &lt;name&gt;headlineViewerTable&lt;/name&gt;
    &lt;tag-class&gt;
        org.jia.components.taglib.HeadlineViewerTableTag
    &lt;/tag-class&gt;
    &lt;body-content&gt;JSP&lt;/body-content&gt;
    &lt;attribute&gt;
        &lt;name&gt;url&lt;/name&gt;
        &lt;required&gt;false&lt;/required&gt;
        &lt;rteprvalue&gt;false&lt;/rteprvalue&gt;
    &lt;/attribute&gt;
    &lt;attribute&gt;
        &lt;name&gt;id&lt;/name&gt;
        &lt;required&gt;false&lt;/required&gt;
        &lt;rteprvalue&gt;false&lt;/rteprvalue&gt;
    &lt;/attribute&gt;
    &lt;attribute&gt;
        &lt;name&gt;rendered&lt;/name&gt;
        &lt;required&gt;false&lt;/required&gt;
        &lt;rteprvalue&gt;false&lt;/rteprvalue&gt;
    &lt;/attribute&gt;
    &lt;attribute&gt;
        &lt;name&gt;showChannelTitle&lt;/name&gt;
        &lt;required&gt;false&lt;/required&gt;
        &lt;rteprvalue&gt;false&lt;/rteprvalue&gt;
    &lt;/attribute&gt;
    ...
    &lt;attribute&gt;
        &lt;name&gt;cellpadding&lt;/name&gt;
        &lt;required&gt;false&lt;/required&gt;
        &lt;rteprvalue&gt;false&lt;/rteprvalue&gt;
    &lt;/attribute&gt;
    &lt;attribute&gt;
        &lt;name&gt;width&lt;/name&gt;
        &lt;required&gt;false&lt;/required&gt;
        &lt;rteprvalue&gt;false&lt;/rteprvalue&gt;
    &lt;/attribute&gt;
&lt;/tag&gt;</pre>
```

The real tag entry includes all of the HTML pass-through attributes plus the attributes defined specifically in the tag handler class. You can download the full code from the book's web site.

The last section shows how to use this exciting new component in a JSP view.

18.7 Using the component

The purpose of developing `UIHeadlineViewer` was to provide a simpler way to display RSS feeds than using the standard `UIData` component (in addition to showing you how to build a data-aware, composite component, that is). So using the component tag should be much more convenient than the example of using `UIData` we showed in listing 18.2.

Fortunately, usage really is quite convenient. In the simplest case all that's necessary is to specify the URL of the RSS feed:

```
<jia:headlineViewerTable  
    url="http://www.jsfcentral.com/jsfcentral.rss"/>
```

This displays the default fields (channel title and header title), doesn't limit the number of rows to display, and has no styles applied. Figure 18.4 shows what it looks like in a browser. It's not exactly pretty, but using it is quite simple.



Figure 18.4 Simple use of `UIHeadlineViewer` with a single `URL` property.

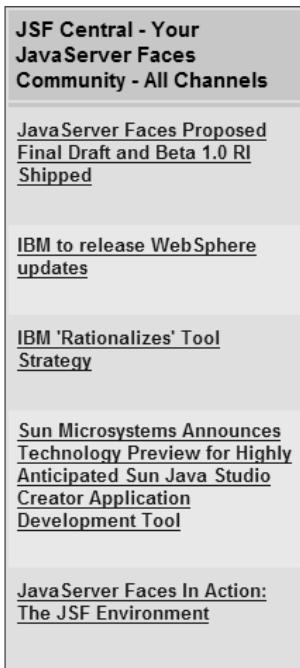


Figure 18.5
UIHeadlineViewer
with styles added.

We can spice things up by adding some styles:

```
<jia:headlineViewerTable
    url="http://www.jsfcentral.com/jsfcentral.rss"
    styleClass="hviewer"
    channelTitleClass="hviewer-channel-title"
    itemTitleClass="hviewer-item-title"
    itemClasses="hviewer-item-even, hviewer-item-odd"
    rows="5"/>
```

Here, we add a style for the overall component, the channel title, the item title, and alternating styles for each row. Also, we limit the number of rows to just 5. The browser output is a definite improvement, as figure 18.5 shows.

Displaying additional fields requires specifying the display attributes explicitly:

```
<jia:headlineViewerTable
    url="http://www.theserverside.com/rss/theserverside-1.0.rdf"
    showChannelTitle="#{testForm.trueProperty}"
    showItemTitle="#{testForm.trueProperty}"
    showItemCreator="#{testForm.trueProperty}"
    showItemPublishedDate="#{testForm.trueProperty}"
    showItemReceivedDate="#{testForm.trueProperty}"
    showItemDescription="#{testForm.trueProperty}"
    styleClass="hviewer"
```

```
channelTitleClass="hviewer-channel-title"
itemTitleClass="hviewer-item-title"
itemHeaderClass="hviewer-item-header"
itemDescriptionClass="hviewer-item-description"
itemClasses="hviewer-item-even, hviewer-item-odd"
rows="5"/>
```

All we've added here are the display properties. The example uses value-binding expressions instead of literal values, but as the name `trueProperty` implies, this property always returns true. Even though this example is a little more verbose than the previous one, it's still compact. As a matter of fact, it's equivalent to the `UIData` example in listing 18.2. All of the Informa API details, as well as the specifics of using `UIData` are gone; all we need is a URL, some styles, and some display properties. Figure 18.6 shows this example in a browser (only the first couple of items appear in the figure). Now, in addition to the default fields, all of the other fields are displayed as well.

The Server Side.Com: Your J2EE Community Forum

[XA Exposed, Part II](#)
Dion Almaer
Published on Wed Jan 14 2004 at 08:14 PM.
Received on Thu Jan 15 2004 at 03:26 AM.
Mike Spille started quite a discussion with his first article on distributed transactions. His second part goes deeper into the world of XA, and the issues that exist there. He goes into details on failure categories, recovery, and the nooks and crannies of XA.

[TSS Article: Testing Java Classes with JUnit](#)
Nitin Bharti
Published on Wed Jan 14 2004 at 07:25 PM.
Received on Thu Jan 15 2004 at 03:26 AM.
In this second part of the "TDD series" Wellie Chao looks at the advantages of JUnit as a testing framework, overviews essential testing concepts and terminology, and walks you through writing a sample JUnit test case. Upcoming articles include: Unit Testing of EJB components, Black Box and In-Container Testing of Servlets and JSPs, and In-Container Testing of Struts Web Applications.

[Quartz Enterprise Job Scheduler 1.3 Released](#)
James House

Figure 18.6
`UIHeadlineViewer` displaying all fields with styles.

For our final example, let's suppose we had the link to an RSS feed stored in the session:

```
FacesContext context = FacesContext.getCurrentInstance();
context.getApplication().
    createValueBinding("#{sessionScope.rssLink}").
    setValue(context,
        "http://news.com.com/2547-1_3-0-20.xml");
```

This link is for CNET News.com's front page headlines. We could reference it with UIHeadlineViewer's component tag like so:

```
<jia:headlineViewerTable url="#{rssLink}"
    showItemDescription="true"
    styleClass="hviewer"
    channelTitleClass="hviewer-channel-title"
    itemTitleClass="hviewer-item-title"
    itemDescriptionClass=
        "hviewer-item-description"
    itemClasses=
        "hviewer-item-even, hviewer-item-odd"/>
```

This displays the default fields plus the item description, utilizing the string stored in the session for the value property. The output is shown in figure 18.7.

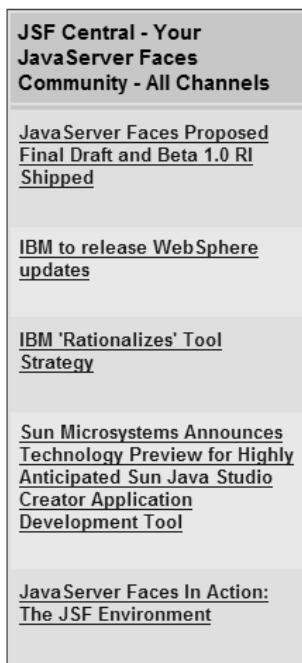


Figure 18.7
UIHeadlineViewer displaying some fields, with the value and style-related properties retrieved from the session.

Now we've consumed a feed by using a value-binding expression, but without having to worry about the Informa API. The URL could have even come from a database or some other source, like a class that chooses a feed randomly.

Now that we've shown some usage examples, it's clear that the effort was worth it: using `UIHeadlineViewer` is much simpler than using `UIData` by itself. It's now quite easy to add RSS headlines to any page with a single tag and no additional programming.

18.8 Summary

In this chapter, we developed `UIHeadlineViewer`—a composite, data-aware component that displays an RSS channel. RSS is a popular XML format used for syndicating news or web log headlines; there are tens of thousands of feeds available today.

Parsing RSS feeds isn't trivial, because several versions are available. To parse different RSS formats and normalize them into a coherent object model, we examined the open source Informa API [Informa]. It's entirely possible to use the Informa API directly with `UIData` to display an RSS channel, but doing so is fairly complicated, and not terribly easy to reuse on different pages.

Therefore, our component subclasses `UIData`. `UIData` components work with `DataModel` objects, so we wrote a simple `ChannelDataModel` class that acts as an adapter between the Informa classes and the component itself. Our customized `UIData` also creates child components and facets for displaying specific fields of an individual channel. In addition, for each field, it exposes a boolean property that controls whether the field should be displayed.

Most important, `UIHeadlineViewer` exposes a `URL` property that can point directly to an RSS feed. This makes life easier for front-end developers, because they don't have to know about the Informa API at all; they just have to know the URL of the RSS feed. Moreover, the component can optionally cache feeds by using an additional managed bean.

The resulting component is powerful—it can consume and display an RSS feed anywhere on the Internet, complete with caching capabilities.

The next chapter covers our final component example—a `UINavigator` component with a Toolbar renderer.

10

UINavigator: a model-driven toolbar component

This chapter covers

- Building a navigation component
- Building a toolbar renderer
- Developing custom model classes
- Writing a private action listener

Our sample application, ProjectTrack, has a cute little toolbar at the top. When we developed the user interface in chapter 9, we used JSP includes to reuse the toolbar code on every page. The toolbar wasn't feature-rich—it just displayed an icon and a link for each item. When you clicked a link, it would simply take you to the requested page. The link you clicked on wasn't highlighted, so there was no way to visually indicate which toolbar link mapped to the current page. Moreover, the contents of the toolbar were hardcoded—you couldn't manipulate them dynamically.

With so many limitations, the toolbar is the perfect candidate for componentization. We updated ProjectTrack to use a toolbar component, which provided all of the same features in a nice, easy-to-use, reusable package. In this chapter, we examine this sophisticated toolbar component, which is driven by a configurable list of items. Each item can have an icon and can either execute an action method or link to a URL. This list can be created in code, via JSP custom tags, or through the Managed Bean Creation facility. You can specify whether the items are displayed horizontally or vertically, and modify its appearance via CSS styles.

NOTE In order to save trees, we have omitted some of the source for this component. You can download all of it from the book's web site (<http://www.manning.com/mann>).

We've been calling this a toolbar component, but if you think about it, it represents a set of navigation options. A toolbar is only one way to represent those options. Consequently, we'll call the component `UINavigator`, and its renderer `ToolbarRenderer`. Figure 19.1 shows examples of what the two look like together.

For a model-driven component like `UINavigator`, there are quite a few different custom pieces. First, there are the model objects—the item list (`NavigatorItemList`) and the items themselves (`NavigatorItem`). These are simple objects (like `SelectItem` and `SelectItems` in the standard component set) that just store data. Next, there's the `UINavigator` class itself, a private `ActionListener` called `NavigatorActionListener`, and its renderer, `ToolbarRenderer`. Finally, there is a JSP component tag to represent individual items as well as the `UINavigator/ToolbarRenderer` combination, and also one for configuring items. All of these elements are shown in figure 19.2

Let's start our tour with the model objects.

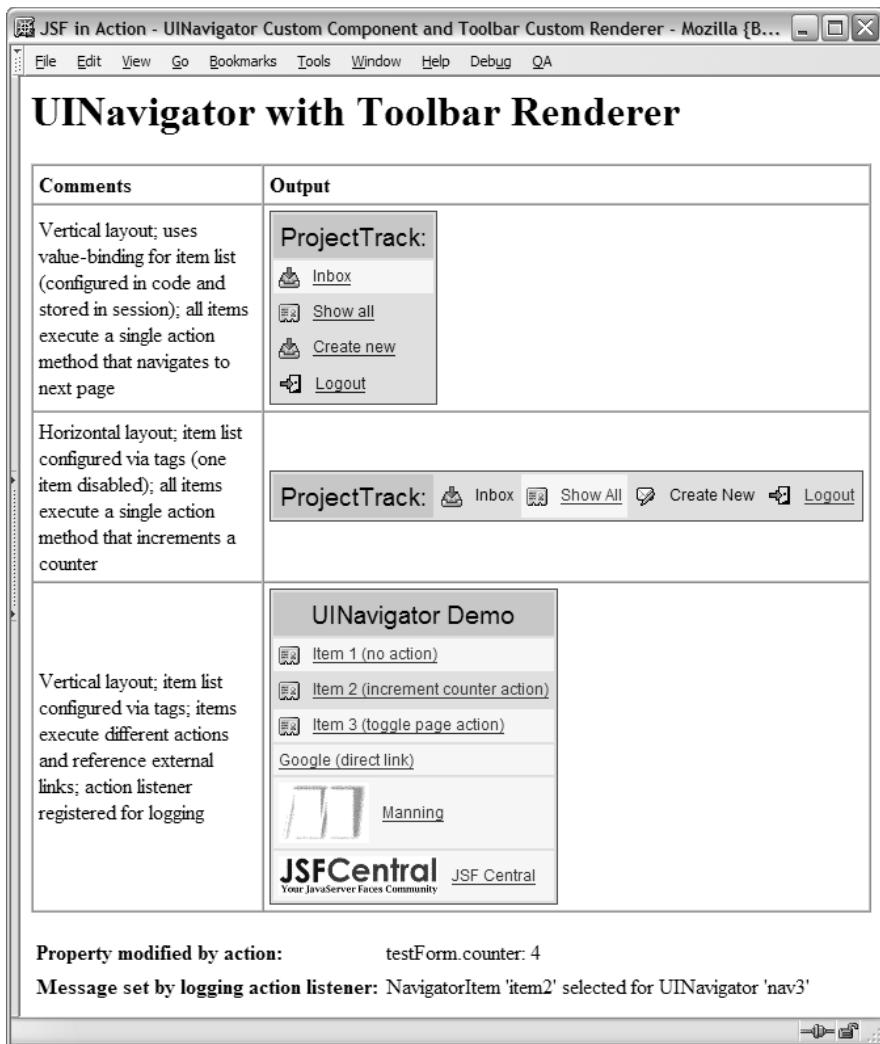


Figure 19.1 **UINavigator** represents a list of navigation options, and the **Toolbar** renderer displays them as a toolbar.

19.1 Writing the model classes

Each option displayed through a **UINavigator** control can be represented by a model class called `jia.components.model.NavigatorItem`. A `NavigatorItem` has a name, a label, and an icon, and can be disabled. It can refer to either an action or a link (URL), but not both. If it refers to a link, the `direct` property determines

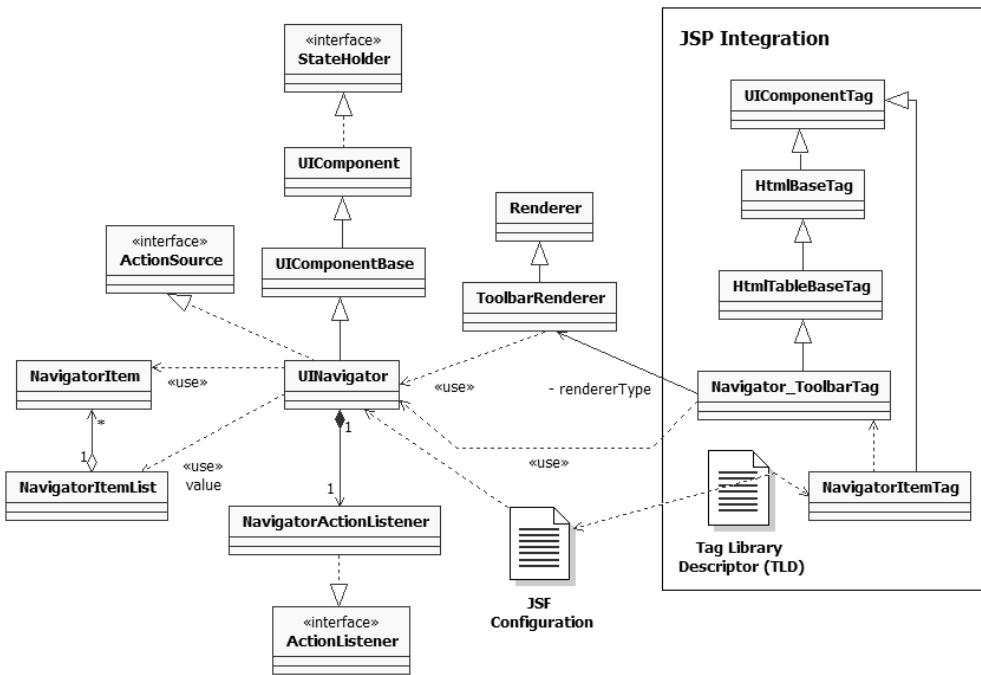


Figure 19.2 Elements involved in building UINavigator. UINavigator is the focal point; it's a component that implements ActionSource and subclasses UIComponentBase. It works with the NavigatorItemList model class, which manages NavigatorItems. UINavigator also has a private ActionListener called NavigatorActionListener. The component can be displayed by ToolbarRenderer, and there are two JSP tags: Navigator_ToolbarTag for registering the component and its renderer, and NavigatorItemTag for registering NavigatorItem instances. Technically, NavigatorItemTag depends on several other classes, but we've omitted those relationships for simplicity.

whether it outputs a direct hyperlink (as opposed to posting back to the application and then performing a redirect). These properties are summarized in table 19.1.

Table 19.1 The properties for the NavigatorItem class

Property	Type	Description
name	String	The name of the item; this is a unique identifier that is required.
label	String	The text displayed for the item.
link	String	The URL the user should see when this item is selected. Use this property or the action property.

continued on next page

Table 19.1 The properties for the `NavigatorItem` class (continued)

Property	Type	Description
<code>direct</code>	<code>Boolean</code>	True if this item should refer directly to the link (and avoid posting back to the application). Only used if the <code>link</code> property is specified.
<code>action</code>	<code>String</code>	Literal action value or method-binding expression for the action that should be executed when the user selects this item.

`NavigatorItem` is a simple JavaBean with only getters and setters (as well as a constructor that takes all of these properties), so we'll spare you the code listing.

TIP One thing you might have noticed about this class is that it doesn't reference any JavaServer Faces objects. It's usually better to leave JSF-related objects out of model classes. For example, `NavigatorItem` has a `String` for its `action` property as opposed to a `MethodBinding` instance. This makes life easier for unit testing and for storing them in the session. (`MethodBinding` isn't serializable, so it can't be persisted in a session.)

Now that we've completed `NavigatorItem`, let's take a look at the class that holds `NavigatorItem` instances. It's called `NavigatorItemList`, and it's a subclass of `java.util.ArrayList`. `UINavigator`'s `value` property will be set to an instance of this class.

`NavigatorItemList` adds a few features to Java's handy `ArrayList`: a `selectedItem` property, and a method to determine if an item with a specific name is in the list, type-safe versions of its basic methods to keep people from putting the wrong type of object in the list.¹ The class is shown in listing 19.1.

Listing 19.1 `NavigatorItemList` manages a list of `NavigatorItem` instances

```
package org.jia.components.model;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class NavigatorItemList extends ArrayList
{
    private NavigatorItem selectedItem;

    public NavigatorItemList()
    {
        super();
    }
}
```

¹ Sadly, JDK 1.5 ("Tiger") and its support for generics wasn't quite ready for prime time when I was working on this book.

```
{  
    super();  
}  
  
public boolean containsName(String name) {  
    Iterator iterator = iterator();  
    while (iterator.hasNext())  
    {  
        if (((NavigatorItem) iterator.next()).getName().equals(name))  
        {  
            return true;  
        }  
    }  
    return false;  
}  
// List methods  
  
public void add(int index, Object element)  
{  
    if (!(element instanceof NavigatorItem))  
    {  
        throw new ClassCastException(  
            "This list only accepts NavigatorItem instances.");  
    }  
    super.add(index, element);  
}  
  
public boolean add(Object element)  
{  
    if (!(element instanceof NavigatorItem))  
    {  
        throw new ClassCastException(  
            "This list only accepts NavigatorItem instances.");  
    }  
    return super.add(element);  
}  
  
public boolean addAll(Collection c)  
{  
    Iterator iterator = c.iterator();  
    while (iterator.hasNext())  
    {  
        if (!(iterator.next() instanceof NavigatorItem))  
        {  
            throw new ClassCastException(  
                "This list only accepts NavigatorItem  
                instances.");  
        }  
    }  
}
```

```
        return super.addAll(c);
    }

    public boolean addAll(int index, Collection c)
    {
        Iterator iterator = c.iterator();
        while (iterator.hasNext())
        {
            if (!(iterator.next() instanceof NavigatorItem))
            {
                throw new ClassCastException(
                    "This list only accepts NavigatorItem instances.");
            }
        }
        return super.addAll(index, c);
    }

    public Object set(int index, Object element)
    {
        if (!(element instanceof NavigatorItem))
        {
            throw new ClassCastException(
                "This list only accepts NavigatorItem instances.");
        }
        return super.set(index, element);
    }

    // Properties

    public NavigatorItem getSelectedItem() <--> Property for selected item
    {
        return selectedItem;
    }

    public void setSelectedItem(
        NavigatorItem selectedItem) <--> Property for selected item
    {
        this.selectedItem = selectedItem;
    }
}
```

That's it for the model classes. In essence, the `UINavigator` component is a wrapper around `NavigatorItemList`. Let's take a look at `UINavigator` now.

19.2 Writing the *UINavigator* class

The `jia.components.UINavigator` class subclasses `UIComponentBase` directly, and implements the `ActionSource` interface. The `ActionSource` interface is important because the component needs to generate action events when a user clicks on one of the items.

Because this component isn't subclassing a standard concrete component class, we must define both the family and the type:

```
public final static String COMPONENT_FAMILY = "jia.Navigator";
public final static String COMPONENT_TYPE = "jia.Navigator";

public String getFamily()
{
    return COMPONENT_FAMILY;
}
```

Simple enough. Now, here's the constructor:

```
public UINavigator()
{
    super();
    setRendererType("jia.Toolbar");
    navigatorListener = new NavigatorActionListener();
    addActionListener(navigatorListener);
}
```

First, we set the renderer type to “`jia.Toolbar`”. This will be the default renderer type for `UINavigator` instances, since it's the only one we will write. Next, we instantiate a `NavigatorActionListener` and add it to the list of action listeners. `NavigatorActionListener` is a special action listener for this component; we'll cover it in section 19.2.4. The method `addActionListener` is required by the `ActionSource` interface; we'll cover it later as well. We add this listener in the constructor to make sure it's always available to process action events.

Because the `value` property of this component should be of type `NavigatorItemList`, it makes sense to provide type-safe aliases for the `value` property methods:

```
public NavigatorItemList getItems()
{
    return (NavigatorItemList)getValue();
}

public void setItems(NavigatorItemList itemList)
{
    setValue(itemList);
}
```

Nothing terribly exciting here—just the joy of casting.

`NavigatorItemList` has a `selectedItem` property, which indicates the current item the user has selected. However, an item isn't really selected until its associated action event has been processed, and such processing won't occur unless validation and conversion for other controls in the same form are successful. In order to keep track of the item the user selected *before* the action event is executed, we need another property—the submitted item:

```
public NavigatorItem getSubmittedItem()
{
    return submittedItem;
}

public void setSubmittedItem(NavigatorItem submittedItem)
{
    this.submittedItem = submittedItem;
}
```

In our custom action listener, we'll set the actual `selectedItem` property of the `NavigatorItemList` to equal the `submittedItem` property. This is similar to the `submittedValue` property of the `EditableValueHolder` interface, which is used by `UIInput` controls.

If you look at figure 19.1, you'll see that each `UINavigator` has a header. Headers, footers, and such are best implemented as facets with convenience methods for accessing them:

```
public UIComponent getHeader()
{
    return (UIComponent) getFacets().get("header");
}

public void setHeader(UIComponent header)
{
    getFacets().put("header", header);
}
```

These methods just simplify the process of retrieving the `header` facet, and also make it clear that `header` is an accepted facet name.

Most of the time, the header is really just a single `UIOutput` component with a text label value. To make things simpler, `UINavigator` has an `addStandardHeader` method that takes a text label value as a parameter and creates the facet for you:

```
public void addStandardHeader(String headerLabel)
{
    UIComponent header = getHeader();
    if (header == null || !(header instanceof UIOutput))
```

```
{  
    UIOutput titleOutput = (UIOutput) getFacesContext().  
        getApplication().createComponent(UIOutput.COMPONENT_TYPE);  
    titleOutput.setValue(headerLabel);  
    setHeader(titleOutput);  
}  
else  
{  
    ((UIOutput) header).setValue(headerLabel);  
}  
}
```

First, we retrieve the current header facet. If it's null, or if it's not a `UIOutput`, we create a new `UIOutput`, set its value to equal that of the text label, and then set the `UIOutput` component as the new header. If there was already a `UIOutput` component set as the header, we just update its `value` to equal the text label. These types of convenience methods are handy when you can anticipate common use cases. The corresponding JSP custom tag (covered in section 19.6.1) takes advantage of this feature.

Surprisingly, there are no additional new `UINavigator` methods; everything else is either an overridden `ActionSource`, `UIComponentBase`, or `StateHolder` method. Let's examine the `ActionSource` methods first.

19.2.1 *Implementing ActionSource methods*

Because `UIComponentBase` doesn't implement `ActionSource`, we'll need to implement its methods from scratch. The interface defines three properties (`action`, `actionListener`, and `immediate`) and two methods (`addActionListener` and `removeActionListener`). Let's examine the `action` property first. Here's the accessor:

```
public MethodBinding getAction()  
{  
    NavigatorItem selectedItem = getItems().getSelectedItem();  
    if (selectedItem == null ||  
        selectedItem.getAction() == null)  
    {  
        return null;  
    }  
    String actionString = selectedItem.getAction();  
    if (Util.isBindingExpression(actionString))  
    {  
        return getFacesContext().getApplication().  
            createMethodBinding(actionString, null);  
    }  
    else  
    {  
        return new ConstantMethodBinding(actionString);  
    }  
}
```

The purpose of this method is to return a `MethodBinding` instance that references an action method. Rather than hold a `MethodBinding` as an instance variable, `UINavigator` creates one based on the `action` property of the currently selected `NavigatorItem`. If there isn't a currently selected `NavigatorItem`, or if the selected `NavigatorItem` instance's `action` property is `null`, we return `null`.

Otherwise, we check to see if the action is a method-binding expression. (`Util.isBindingExpression` just checks to make sure that the string starts with “# {” and ends with “}”.) If so, we create and return a new method binding using the item's `action` property. If the action isn't a method-binding expression, we return a new instance of `ConstantMethodBinding`, which is a special `MethodBinding` subclass that just returns the literal value of the action. (So, if the action was “hitMe”, the `ConstantMethodBinding` instance's `invoke` method would return “hitMe”.)

Here's the code for the corresponding mutator:

```
public void setAction(MethodBinding actionBinding)
{
    NavigatorItem selectedItem = getItems().getSelectedItem();
    if (selectedItem == null)
    {
        throw new IllegalStateException(
            "No item is currently selected.");
    }
    if (actionBinding == null)
    {
        selectedItem.setAction(null);
    }
    else
    {
        selectedItem.setAction(actionBinding.getExpressionString());
    }
}
```

Like the accessor, the mutator really delegates to the selected `NavigatorItem` instance. If there isn't a selected `NavigatorItem`, we throw an `IllegalStateException`—you can't set a property on an object that doesn't exist. Otherwise, we set the selected `NavigatorItem` instance's `action` property based on the underlying expression of the `MethodBinding` instance.

`ActionSource` also defines methods for handling `ActionListener` instances. It has a single `ActionListener` property, which is a `MethodBinding` that must refer to a method that accepts a single `ActionEvent` as a parameter. (You may remember this property from the `UICommand` component, which implements `ActionSource`.) Here are the methods for this property:

```
public void setActionListener(MethodBinding actionListener)
{
    this.actionListenerBinding = actionListenerBinding;
}

public MethodBinding getActionListener()
{
    return actionListenerBinding;
}
```

UINavigator simply holds on to a `MethodBinding` instance, so there's nothing exciting about these methods.

In addition to the single `actionListener` property, `ActionSources` must maintain a list of `ActionListener` instances. Here are the methods for handling that list:

```
public void addActionListener(ActionListener listener)
{
    addFacesListener(listener);
}

public void removeActionListener(ActionListener listener)
{
    removeFacesListener(listener);
}

public ActionListener[] getActionListeners()
{
    return (ActionListener[])getFacesListeners(ActionListener.class);
}
```

All of these methods simply delegate to `UIComponent` utility methods that handle the listeners by type.

`ActionSource` also defines an `immediate` property, which indicates whether associated `ActionListeners` should be executed during the Apply Request Value phase (if it's `true`) or the Invoke Application phase (if it's `false`) of the Request Processing Lifecycle. Like most properties, it makes life easier for developers if you make it aware of value-binding expressions. Here's the `immediate` property (which usually defaults to `false`):

```
private Boolean immediate = null;
...
public boolean isImmediate()
{
    return Util.getBooleanProperty(this, immediate, "immediate",
        false);
}

public void setImmediate(boolean immediate)
```

```
{
    this.immediate = new Boolean(immediate);
}
```

Notice that we store the property as a Boolean object instead of a primitive type. This allows us to keep track of whether a property has been set; if it is non-null, the `Util.getBooleanProperty` method will simply return its boolean value. However, if it is null, the method will evaluate its value-binding expression (if there is one). If no value-binding expression can be found, `Util.getBooleanProperty` returns a default value, which is `false` in this case. (Recall that `UIComponent` keeps a list of `ValueBinding` instances, keyed by property or attribute name.)

That's it for the `ActionSource` methods. Now, let's look at the `UIComponentBase` methods that `UINavigator` overrides.

19.2.2 Overriding `UIComponentBase` methods

Only a few of `UIComponentBase`'s methods need to be overridden. The first is `getRendersChildren`. By default, the `rendersChildren` property returns `false`, so that any child components or facets will render themselves. Because the component we're writing has a header facet and allows no additional child components, we can set the `rendersChildren` property to `true`:

```
public boolean getRendersChildren()
{
    return true;
}
```

`rendersChildren` is defined as read-only property by `UIComponent`, so there's no setter method, and we can just hardcode `true` as the return value. In general, the `rendersChildren` property should only return `true` if your encoding methods actually display child components.

By default, the `broadcast` method invokes all listeners whose `phaseId` property matches the `phaseId` of the current event. Because the `actionListener` property is a `MethodBinding` instance and not an `ActionListener`, we need to add support for executing it:

```
public void broadcast(FacesEvent event)
                      throws AbortProcessingException
{
    super.broadcast(event);

    MethodBinding binding = getActionListener();
    if (binding != null)
    {
        FacesContext context = getFacesContext();
```

```

        binding.invoke(context, new Object[] { event });
    }
}
}

```

First, we delegate to the superclass for all of the normal event processing. Next, we retrieve the `actionListener` property. If the property is non-null, we invoke the `MethodBinding`, passing in the event as the parameter.

We defined the `immediate` property earlier, but so far we haven't written any code that uses it. `ActionSource` components are supposed to process action events during the `Apply Request Values` phase of the Request Processing Lifecycle if the `immediate` property is `true`, and during the `Invoke Application` phase if `immediate` is `false`. We can enforce this rule by overriding the `queueEvent` method, which is responsible for adding new events to our component:

```

public void queueEvent(FacesEvent event)
{
    if (event instanceof ActionEvent)
    {
        if (isImmediate())
        {
            event.setPhaseId(PhaseId.APPLY_REQUEST_VALUES);
        }
        else
        {
            event.setPhaseId(PhaseId.INVOKE_APPLICATION);
        }
    }
    super.queueEvent(event);
}

```

All `FacesEvent` subclasses (such as `ActionEvent`) have a `phaseId` property that indicates when they should be executed. So all we have to do is set this property depending on the value of `immediate`. Now, whenever an action event is added to this component, we ensure that it is executed during the proper lifecycle phase.

There are no additional `UIComponentBase` methods to override. Now, let's examine state management.

19.2.3 Implementing *StateHolder* methods

State management for `UINavigator` is pretty normal, except for handling action listeners. First, there's the default `NavigatorActionListener` that we registered in the constructor. Next, there's the `actionListener` property. Take a look at how they're managed in `saveState`:

```

public Object saveState(FacesContext context)
{
}

```

```

removeActionListener(navigatorListener);

Object[] values = new Object[5];
values[0] = super.saveState(context);
values[1] = value;
values[2] = valid ? Boolean.TRUE : Boolean.FALSE;
values[3] = immediate;
values[4] = saveAttachedState(context, actionListener);

return values;
}

```

Note that we actually remove the `navigatorListener` from the list of action listeners. Because this listener has no properties, there's no need to waste time and (possibly bandwidth) saving it. Also, note that we save the `actionListener` property (which is a `MethodBinding` instance) by using `UIComponentBase`'s `saveAttachedState` utility method instead of just storing it directly. You should use `saveAttachedState` for any associated objects like this, especially if there's a chance that they might implement the `StateHolder` interface.

All of the other work in this method is standard: we create an array of `Objects`, assigning properties to each element in the array, beginning with the state of the superclass.

Here is the corresponding `restoreState` method:

```

public void restoreState(FacesContext context, Object state)
{
    Object[] values = (Object[])state;
    super.restoreState(context, values[0]);
    value = values[1];
    valid = ((Boolean)values[2]).booleanValue();
    Boolean immediateState = (Boolean)values[3];
    if (immediateState != null)
    {
        setImmediate(immediateState.booleanValue());
    }
    actionListener = (MethodBinding)restoreAttachedState(
        context, values[4]);
}

```

Retrieval of most of these properties is standard—we just retrieve them from the array of `Objects`, starting with the superclass's state. Note that we don't initialize the `navigatorListener` variable because it was already created in `UINavigator`'s constructor. Because we stored `actionListener` using the `saveAttachedState` method, we must retrieve it using `restoreAttachedState`, another `UIComponentBase` utility method that performs the opposite task.

We have now covered all of `UINavigator`'s methods. However, it would be rude to ignore the details of `NavigatorActionListener`, so we'll examine its implementation next.

19.2.4 Developing `NavigatorActionListener`: a custom `ActionListener`

When we discussed the goals of `UINavigator`, we said that for the selected `NavigatorItem` instance, the control would either fire an action method or redirect to a URL. In order to implement this functionality, `UINavigator` has its own private `ActionListener` subclass, `NavigatorActionListener`. Firing an action method is handled by the application's default `ActionListener` instance, so `NavigatorActionListener` delegates to that instance if the selected `NavigatorItem` has a null `action` property. Otherwise, it handles the redirect all by itself.

An instance of `NavigatorActionListener` is created and added to `UINavigator` in its constructor, which ensures that there's always a single up-to-date instance waiting to process any action events triggered by the user.

The `ActionListener` interface has a single method called `processAction`; here's our implementation:

```
public void processAction(ActionEvent event)
{
    UINavigator navigator = (UINavigator)event.getComponent();

    NavigatorItem selectedItem = navigator.getSubmittedItem();
    navigator.getItems().setSelectedItem(selectedItem);
    navigator.setSubmittedItem(null);

    if (navigator.getAction() != null)
    {
        getFacesContext().getApplication().getActionListener().
            processAction(event);
    }
}
```

First, we cast the source of the event as a `UINavigator` instance. This is a safe thing to do, because no other class can instantiate a `NavigatorActionListener`. Next, we retrieve the `submittedItem` property from the component, update the currently selected item, and set the `submittedItem` property to non-null. This is effectively committing the user's selection.

Now that the user's choice has been solidified, we can retrieve the `action` property from the `UINavigator` instance. Recall that this property's value is based on the selected `NavigatorItem`'s `action` property. If the property isn't null, we simply delegate processing to the Application's `actionListener`, which ensures that the default processing occurs, complete with navigation.

If the `action` property is `null`, then we can redirect to the URL specified by the selected item's `link` property:

```
else
{
    String link = selectedItem.getLink();
    if (link != null)
    {
        FacesContext context = getFacesContext();
        try
        {
            context.getExternalContext().redirect(link);
        }
        catch (IOException e)
        {
            String message = "Error redirecting to link '" + link + "'";
            context.addMessage(navigator.getClientId(context),
                new FacesMessage(FacesMessage.SEVERITY_ERROR,
                    message, null));
            context.getExternalContext().log(message, e);
        }
    }
}
```

If the `link` property is non-null, we try to redirect to it using the `sendRedirect` method of the `ExternalContext`. This method can throw an `IOException`, so we must catch it, log the error, and add a message to the current context.

This completes the tour of `UINavigator` and its inner class, `NavigatorActionListener`. Next, let's handle component registration.

19.3 Registering the component

Registration for this component is as simple or complex as any other. We'll choose the simple route, and just add the bare minimum amount of metadata; remember, however, that you can do much more (see chapter 15 for an example).

```
<component>
  <description>
    Displays a set of buttons (in a table) that execute
    action methods.
  </description>
  <display-name>Navigator</display-name>
  <component-type>jia.Navigator</component-type>
  <component-class>org.jia.components.UINavigator</component-class>
</component>
```

This declares a component of type `jia.Navigator` with the display name `Navigator` and the class name `org.jia.components.UINavigator`. Note that the type

matches the value of the `COMPONENT_TYPE` constant we declared in section 19.2. Now, on to the renderer.

19.4 Writing the ToolbarRenderer class

Now that the model and component classes are complete, it's time to address displaying the component. We'll develop a `ToolbarRenderer` that can display the items in an HTML table, either horizontally or vertically. The renderer will also fire an `ActionEvent` when a user clicks on an item.

For controlling the appearance of the associated `UINavigator`, the renderer uses attributes for specifying the layout and the styles associated with portions of the HTML table. These attributes are listed in table 19.2. The renderer also uses base HTML pass-through attributes as well as HTML pass-through attributes for tables, like `cellpadding` and `cellspacing`.

Table 19.2 The attributes for ToolbarRenderer

Property	Type	Description
<code>layout</code>	<code>String</code>	Specifies whether or not to display all items in a single row ("HORIZONTAL") or in a single column ("VERTICAL")
<code>headerClass</code>	<code>String</code>	The CSS style for the header of the table
<code>itemClass</code>	<code>String</code>	The CSS style for all unselected items
<code>selectedItemClass</code>	<code>String</code>	The CSS style for the selected item
<code>iconClass</code>	<code>String</code>	The CSS style for the icon

Now, let's look at how `ToolbarRenderer` displays the `UINavigator` component.

19.4.1 Encoding

If you look at figure 19.1, you can see that `UINavigator` is displayed as a table, complete with CSS styles, a header, and a label (and optionally an icon) for each item. The table can display either horizontally or vertically. Listing 19.2 shows what the HTML output for a horizontal table should look like (this is the HTML for the first example shown in the figure).

Listing 19.2 The expected HTML output of ToolbarRenderer for the first vertical toolbar shown in figure 19.1

```

<input type="hidden" name="mainForm:nav1" />           ←❶ Hidden field for
                                                        selected item

<table cellpadding="4" class="navigator2"> ←❷ Items laid
                                                        out in table

    <thead> ←❸ Header facet
        <tr>
            <th class="navigator-title">ProjectTrack:</th>
        </tr>
    </thead>
    <tbody>
        <tr class="navigator-selected-command">           ←❹ Each item
            <td>                                         represented
                <a href="#"                                by a row
                    onmousedown="document.forms['mainForm']
                                    ['mainForm:nav1'].value='0';
                                    document.forms['mainForm'].submit()">
                        Inbox
                    </a>
                </td>
            </tr>
        ..
        <tr class="navigator-command">
            <td>
                <a href="#"                                onmousedown="document.forms['mainForm']
                                    ['mainForm:nav1'].value='3';
                                    document.forms['mainForm'].submit()">
                    Logout
                </a>
            </td>
        </tr>
    </tbody>
</table>

```

The first thing to note is that there's a hidden form element whose name is equal to the client identifier of the component, which is `mainForm:nav1` in this case (❶). The hidden field is followed by a standard HTML table element with a `cellpadding` attribute and a CSS style (❷). It also has a header, which is derived from `UINavigator`'s header facet (❸).

Each `NavigatorItem` is represented by a single table row (❹). Each row wraps an image and the item's label with an HTML anchor that has inline JavaScript for its

onmousedown event. This JavaScript sets the enclosing hidden field's value to equal the index of the corresponding item on the server, and then submits the form.

The example shows `UINavigator` displayed with a vertical layout. It can also be displayed with a horizontal layout. The only differences are that items are displayed in columns instead of rows, and that instead of having a table header row (an HTML `<thead>` element), the header is displayed as the first column.

Let's get down to business. The class `jia.components.ToolbarRenderer` directly subclasses the `Renderer` abstract class. Because `UINavigator` has a header facet, we will implement `encodeBegin` for the beginning of the table, `encodeChildren` for the facet, and `encodeEnd` for the items and the end of the table. Let's start with `encodeBegin`:

```
public void encodeBegin(FacesContext context,
                        UIComponent component)
                        throws java.io.IOException
{
    if (component.isRendered())
    {
        UINavigator navigator = (UINavigator)component;
        Map attributes = navigator.getAttributes();
        ResponseWriter writer = context.getResponseWriter();
        writer.startElement("input", navigator);
        writer.writeAttribute("type", "hidden", null);
        writer.writeAttribute("name",
                             navigator.getClientId(context), "clientId");
        writer.endElement("input");
        writer.startElement("table", navigator);
        if (attributes.get("cellpadding") == null)
        {
            attributes.put("cellpadding", "4");
        }
        Util.writePassthroughAttributes(attributes, writer);
    }
}
```

First, we check to see if the component should be displayed. If so, we start by outputting the hidden HTML `<input>` element. Next, we begin outputting the `<table>` element. The second parameter of `ResponseWriter.startElement` is always the associated component, which is `UINavigator` in this case. (This is for use by tools, but has no effect on your application.) If no `cellpadding` attribute has been set on the component, we add one with a default value of 4. Next, we write through all of the pass-through attributes (which include the `cellpadding` attributes). The `Util.writePassthroughAttributes` method simply outputs any attribute that's used for standard HTML elements, evaluating value-binding expressions if necessary.

The encodeChildren method is where we output the header facet:

```
public void encodeChildren(FacesContext context,
                           UIComponent component)
                           throws java.io.IOException
{
    if (component.isRendered())
    {
        UINavigator navigator = (UINavigator) component;
        Map attributes = navigator.getAttributes();
        UIComponent header = navigator.getHeader();
        boolean vertical = isVertical(attributes);
        if (header != null)
        {
            ResponseWriter writer = context.getResponseWriter();
            if (vertical)
            {
                writer.startElement("thead", header);
                writer.startElement("tr", header);
                writer.startElement("th", header);
            }
            else
            {
                writer.startElement("tbody", navigator);
                writer.startElement("tr", header);
                writer.startElement("td", header);
            }
        }
    }
}
```

If the component is visible, we begin by declaring some local variables. The vertical variable tells if the table should be vertical (each item displayed on a separate row) or horizontal (all items displayed on the same row). (The isVertical method returns true if the layout attribute is non-null and equals “VERTICAL”, and false if it is non-null and equals “HORIZONTAL”.)

If the header facet itself isn’t null, we can display it. If the layout is vertical, we begin outputting markup for an HTML header row; otherwise we just start outputting a normal table body and the first column of the first row. Note that for the startElement method of the ResponseWriter, we pass the header component for all of the header-related elements instead of the UINavigator instance.

Now that we’ve started all of the elements surrounding the actual header output, it’s time to apply any attributes that may affect the header. In table 19.2, we defined the headerClass attribute, which is the style that should be applied to the header. The following code outputs that attribute:

```
if (attributes.get("headerClass") != null)
{
    writer.writeAttribute("class",
```

```
        attributes.get("headerClass"),  
        "headerClass");  
    }  
}
```

Now it's time to display the actual header component. Whenever you want to render a facet or child component, you simply call its encoding methods directly:

```
header.encodeBegin(context);  
header.encodeChildren(context);  
header.encodeEnd(context);
```

These lines perform all of the necessary encoding for the header, embedded within our table.

Finally, we finish surrounding the header's output:

```
        if (vertical)
        {
            writer.endElement("th");
            writer.endElement("tr");
            writer.endElement("thead");
        }
        else
        {
            writer.endElement("td");
        }
    }
}
```

If the layout is vertical, we end the table header row. Otherwise, we just end the column.

The `encodeBegin` and `encodeChildren` methods displayed the beginning of the table and its header. The meat, which is the markup for the individual items, is displayed in `encodeEnd`:

```
public void encodeEnd(FacesContext context, UIComponent component)
                      throws java.io.IOException
{
    if (component.isRendered())
    {
        ResponseWriter writer = context.getResponseWriter();
        UINavigator navigator = (UINavigator)component;
        Map attributes = navigator.getAttributes();
        boolean vertical = isVertical(attributes);
        UIPrimitive parentForm = getParentForm(navigator);
        if (parentForm == null)
        {
            throw new NullPointerException("No parent UIPrimitive found.");
        }

        NavigatorItemList items = navigator.getItems();
```

```

if (items != null && items.size() > 0)
{
    if (vertical)
    {
        writer.startElement("tbody", navigator);
    }
    else
    if (navigator.getHeader() == null)
    {
        writer.startElement("tbody", navigator);
        writer.startElement("tr", navigator);
    }
}

```

As usual, we first check to see if the `rendered` property is `true`. If so, we declare some variables, and then retrieve the parent `UIForm` of our `UINavigator` instance using `getParentForm`, which is a utility method that we'll cover later. Having a handle to the parent form and its index is vital for the JavaScript event handlers that are rendered for each item. Finally, if there are any items in the list, we begin outputting the table's body (the output varies depending on the layout).

Next, we output a table cell for each item in the list:

```

Iterator itemIterator = items.iterator();
while (itemIterator.hasNext())
{
    encodeItem((NavigatorItem)itemIterator.next(), items,
               context, writer, navigator, attributes,
               vertical, parentForm);
}

```

This code just calls `encodeItem` (a utility method we'll discuss next) for each item in the list. Once all of the items have been displayed, we can close the open elements that were started earlier:

```

if (!vertical)
{
    writer.endElement("tr");
}
writer.endElement("tbody");
}
writer.endElement("table");
}
}

```

That's it for `encodeEnd`. Let's look at `encodeItem`, which was executed for each item in the list. This method handles the details of displaying each individual item within a table cell. Here's the code that outputs the beginning of the cell:

```

private void encodeItem(NavigatorItem item,
                      NavigatorItemList items, FacesContext context,

```

```

        ResponseWriter writer, UINavigator navigator,
        Map attributes, boolean vertical, UIForm parentForm)
        throws IOException
    {
        if (vertical)
        {
            writer.startElement("tr", navigator);
        }
        writer.startElement("td", navigator);
        writeItemClass(writer,
                      item.equals(navigator.getItems().getSelectedItem()),
                      attributes);
    }
}

```

If the layout of the `UINavigator` is vertical, we start with a table row; otherwise, we start with a table cell element. (Remember, if the layout is vertical, there's one item per row; otherwise, each item occupies a cell in a table with only one row.) Then we output the CSS style (an HTML `class` attribute) for the item using the `writeItemClass` utility method. This simple `ToolbarRenderer` method uses the `selectedItemClass` attribute if the item is currently selected; otherwise, it uses the `itemClass` attribute.

Now, we can output the item's label and icon, enclosed in a hyperlink if necessary:

```

if (!item.isDisabled())
{
    writer.startElement("a", navigator);
    if (item.isDirect() && item.getLink() != null)
    {
        writer.writeAttribute("href",
                              context.getExternalContext().encodeResourceURL(
                                  item.getLink()),
                              null);
    }
    else
    {
        writer.writeAttribute("href", "#", null);
        String clientId = navigator.getClientId(context);
        writer.writeAttribute("onmousedown",
                              "document.forms['" +
                              parentForm.getClientId(context) +
                              "'][]['" + clientId + "'].value=''" +
                              items.indexOf(item) + "'"; document.forms['" +
                              parentForm.getClientId(context) +
                              "'][].submit()", null);
    }
}
if (item.getIcon() != null)
{

```

```

        encodeIcon(context, writer, attributes, item.getIcon());
    }
    if (item.getLabel() != null)
    {
        writer.writeText(item.getLabel(), null);
    }
    if (!item.isDisabled())
    {
        writer.endElement("a");
    }
}

```

If the item isn't disabled, we begin outputting the `<a>` element. For direct item links, we simply output the `href` attribute to avoid a roundtrip to the server. Note that we're using the `encodeResourceURL` method on the `ExternalContext` to encode the item's `link` property; using this method is required anytime you need to output a URL directly.

If the item doesn't require a direct link, then we want it to postback so that `ToolbarRenderer` can generate an `ActionEvent` based on the item the user selected (this will happen in the `decode` method, which we'll cover later). To do this, we output JavaScript that modifies the value of the hidden field (rendered in `encodeBegin`) to be equal to the index of this item. This JavaScript code uses the `clientId` property of the parent `UIForm` to make sure it's referencing the proper hidden field. (Refer back to listing 19.2 to see what this output looks like.) This way, when the renderer decodes the user's response, it can associate it with a specific item on behalf of the user.

After we output the hyperlink (if necessary), we simply display the icon and the item's label. (If the item is disabled, the icon and label are still displayed, but without the hyperlink.) Then, if necessary, we finish outputting the enclosing hyperlink element. The `encodeIcon` method outputs the actual graphic hyperlink; we'll cover that method next.

`ResponseWriter`'s `writeAttribute` and `writeText` methods take the component's property as the last parameter. In this case, we pass in `null` because this output isn't associated with a component (each `NavigatorItem` is a model object, not a component).

Once the item's label and icon have been displayed, we can end the column and row (if necessary):

```

writer.endElement("td");
if (vertical)
{
    writer.endElement("tr");
}
}

```

That's it for displaying an individual `NavigatorItem` instance. `encodeItem` did, however, call another method, `encodeIcon`:

```
protected void encodeIcon(FacesContext context,
                          ResponseWriter writer,
                          Map attributes, String iconUrl)
                          throws IOException
{
    if (iconUrl.startsWith("/"))
    {
        iconUrl = context.getExternalContext().
                    getRequestContextPath() + iconUrl;
    }
    writer.startElement("img", null);
    writer.writeAttribute("border", "0", null);
    writer.writeAttribute("src",
                          context.getExternalContext().
                          encodeResourceURL(iconUrl),
                          null);

    if (attributes.get("iconClass") != null)
    {
        writer.writeAttribute("class", attributes.get("iconClass"),
                              null);
    }
    else
    {
        writer.writeAttribute("style",
                              "margin-left: 0px; margin-right: 10px; " +
                              "border: 0px; vertical-align: middle;", null);
    }
    writer.endElement("img");
}
```

There are two important features of this method. First, if the icon starts with a forward slash (/), we prepend it with the context path. Prepending the URL ensures that all absolute links are made relative to the current servlet or portlet; this is the same behavior the standard components use when displaying image URLs. Second, for each `` element, we output a CSS style that equals the `iconClass` renderer attribute, if it has been set. Otherwise, we hardcode the style to equal something that works well in many cases.

TIP

It's usually a good idea to provide default style values so that the component user gets a feel for how the component should look before they tweak its appearance.

Finding the parent form

Instead of making any assumptions about the form enclosing the `UINavigator` component, `ToolbarRenderer`'s `encode` method finds the `UINavigator` instance's parent `UIForm`. The `encodeItem` method uses the form's `clientId` property to make sure that the JavaScript it outputs references the proper form on the client. Here's the code for `getParentForm`:

```
protected UIForm getParentForm(UIComponent component)
{
    UIComponent parent = component.getParent();
    while (parent != null)
    {
        if (parent instanceof UIForm)
        {
            break;
        }
        parent = parent.getParent();
    }
    return (UIForm)parent;
}
```

This method moves from the current component (our `UINavigator` instance) recursively up the component tree until it finds the first `UIForm` object, which is the parent form. Finding the parent form is useful for cases where you need to reference the form explicitly on the client side (as we did with JavaScript in the `encodeItem` method).

Now that we've covered the encoding logic of `ToolbarRenderer`, let's see how it handles decoding.

19.4.2 Decoding

As usual, decoding is a much simpler process than encoding. All we have to do is check the request parameters for one whose name equals the `clientId` of the `UINavigator` instance. If there's a match, we just set the selected item and fire an `ActionEvent`. Here's the `decode` method:

```
public void decode(FacesContext context, UIComponent component)
{
    if (Util.canModifyValue(component))
    {
        UINavigator navigator = (UINavigator)component;
        Map parameterMap = context.getExternalContext() .
                           getRequestParameterMap();
        String selectedItem =
            (String)parameterMap.get(
                (String)navigator.getClientId(context));
```

```

        if (selectedItem != null && selectedItem.length() > 0)
        {
            NavigatorItemList items = navigator.getItems();
            if (items != null)
            {
                navigator.setSubmittedItem((NavigatorItem)items.get(
                    Integer.parseInt(selectedItem)));
                navigator.queueEvent(new ActionEvent(navigator));
            }
        }
    }
}

```

First, we check to see if this component is accepting input. The `Util.canModifyValue` returns `true` if the component's `rendered` property is `true`, and it's not disabled or read-only (as specified by render-dependent HTML attributes). If we can modify the value, we start by retrieving the request parameter map from the external context. We then check to see if the map includes a parameter whose name equals the client identifier of this component. If so, the user has clicked on this `UINavigator` instance, so we select the proper item in the `UINavigator`'s list and fire a new `ActionEvent`. Note that the item is selected based on its index, which the `encodeItem` method rendered in its JavaScript event handler. (Another option would have been to select an item based on its `name` property.)

Once the `decode` method has executed, the normal JSF processing takes place. If a new `ActionEvent` was fired, the processing results in execution of the `NavigationActionListener` instance, which either delegates to the application's default `ActionListener` or redirects to the selected `NavigatorItem`'s `link` property.

This completes our tour of `ToolbarRenderer`. Now, let's register our new renderer with JSF.

19.5 Registering the renderer

Here's the renderer entry for `ToolbarRenderer` in an application configuration file:

```

<renderer>
    <description>
    <display-name>Toolbar</display-name>
    <component-family>jia.Navigator</component-family>
    <renderer-type>jia.Toolbar</renderer-type>
    <renderer-class>
        org.jia.components.ToolbarRenderer
    </renderer-class>
</renderer>

```

Note that the same renderer type was set in `UINavigator`'s constructor, effectively making `ToolbarRenderer` its default navigator. On to the world of JSP.

19.6 JSP integration

Sometimes, integrating a component with JSP involves developing more than one tag handler. This is usually the case when you've developed child components that work with your primary component. In this case, there are no child components, but we'll use a custom tag handler to create new `NavigatorItem` instances for the parent `UINavigator`.

TIP Remember, custom tags don't always have to create JSF components. They can perform other operations that initialize or manipulate your component in some way.

So, this means that in addition to developing the component tag for the `UINavigator-ToolbarRenderer` pair, we'll develop one that represents a single `NavigatorItem` instance and that is designed to be nested within the component tag.

19.6.1 Writing the `Navigator_ToolbarTag` component tag

The primary component tag for our new component and renderer is called `org.jia.components.Navigator_ToolbarTag`. It subclasses a class called `HtmlTableBaseTag`, which provides support for properties that map to standard HTML tables. The implementation of this class is similar to `HtmlBaseTag` (covered in online extension chapter 17), so we'll skip the details. Just remember that it implements additional table-specific HTML pass-through attributes such as `cellpadding` and `width`, in addition to the basic attributes the `HtmlBaseTag` provides.

`HtmlBaseTag` supports pass-through attributes, but we still need to support both component properties and renderer attributes in `Navigator_ToolbarTag`. We'll expose them all as properties of the tag handler; table 19.3 lists them all.

Table 19.3 The properties for `Navigator_ToolbarTag`. This includes all of the properties from the `UINavigator` component and the attributes from `ToolbarRenderer`.

Property	Type	Description	UINavigator	Toolbar
<code>value</code>	<code>String</code>	The value of this component. Must be a value-binding expression that evaluates to a <code>NavigatorItemList</code> .	X	

continued on next page

Table 19.3 The properties for Navigator_ToolbarTag. This includes all of the properties from the UINavigator component and the attributes from ToolbarRenderer. (continued)

Property	Type	Description	UINavigator	Toolbar
immediate	String	True if the default NavigatorAction-Listener should be executed immediately (during the Apply Request Values phase).	X	
action-Listener	String	Method-binding expression pointing to an action listener method.	X	
layout	String	Specifies whether or not to display all items in a single row ("HORIZONTAL") or in a single column ("VERTICAL").		X
header-Class	String	The CSS style for the header of the table.		X
itemClass	String	The CSS style for all unselected items.		X
selected-ItemClass	String	The CSS style for the selected item.		X
iconClass	String	The CSS style for the icon.		X
headerLabel	String	The value used for the default header facet.		

You might have noticed a couple things about this table. First, UINavigator's action property isn't exposed. This is intentional—the action is set based on the currently selected item. Also, the headerLabel property doesn't map to a specific UINavigator property or Toolbar attribute. This is because a non-empty headerLabel property initiates a call to UINavigator's addStandardHeader method, which creates a default UIOutput instance.

TIP Think of the tag handler as an interface to your component–renderer pair. As such, you may expose properties whose relationship with the component or renderer counterpart is more complicated than just setting a property.

As the name Navigator_ToolbarTag implies, this component tag is specifically designed for the UINavigator component type and the Toolbar renderer type. The first implementation step is to set those properties:

```
public String getComponentType()
{
    return UINavigator.COMPONENT_TYPE;
}
```

```
public String getRendererType()
{
    return "jia.Toolbar";
}
```

Now that it's clear which types of objects this handler supports, the next step is to override the `setProperties` method. The first thing it does is set the `UINavigator`'s `immediate` property:

```
protected void setProperties(UIComponent component)
{
    super.setProperties(component);

    UINavigator navigator = (UINavigator)component;
    FacesContext context = FacesContext.getCurrentInstance();
    Application app = context.getApplication();

    if (immediate != null)
    {
        if (isValueReference(immediate))
        {
            navigator.setValueBinding("immediate",app.
                createValueBinding(immediate));
        }
        else
        {
            navigator.setImmediate(new Boolean(immediate).booleanValue());
        }
    }
}
```

First, we call the superclass's `setProperties` method. Then, if the `immediate` property is non-null and it's a value-binding expression, we set its value binding. Otherwise, we set the property directly. (The `isValueReference` method is a utility method implemented by the `UIComponentTag` base class. Remember, you must go through the extra step of checking for a value-binding if you want to support value-binding expressions in every property, like the standard component tags.)

Next, we handle the `headerLabel` property:

```
if (headerLabel != null)
{
    if (isValueReference(headerLabel))
    {
        ValueBinding binding = app.createValueBinding(headerLabel);
        navigator.addStandardHeader(
            (String)binding.getValue(context));
    }
    else
```

```
{  
    navigator.addStandardHeader(headerLabel);  
}  
}  
}
```

The code here is almost exactly the same, except that we call `addStandardHeader` instead of a setter property. This method will create a new header `UIOutput` facet that has `headerLabel` as its value. This is an example of a tag handler property that doesn't map directly to a component property or renderer attribute.

The `actionListener` property must be a method-binding expression, so we handle things slightly differently:

```
if (actionListener != null)
{
    if (isValueReference(actionListener))
    {
        MethodBinding mBinding =
            app.createMethodBinding(actionListener,
                new Class[] { ActionEvent.class });
        navigator.setActionListener(mBinding);
    }
    else
    {
        throw new IllegalArgumentException(
            "The actionListener property " +
            "must be a method-binding expression.");
    }
}
```

We start by once again calling `isValueReference` (this works for both value and method-binding expressions), and then create a new `MethodBinding` instance if necessary. Note that the method binding specifies a single `ActionEvent` parameter for the method signature, which is the requirement for action listener methods. If the `actionListener` property isn't a method-binding expression, we throw an `IllegalArgumentException`, since that is a requirement. (It is also possible to handle such details using a `TagExtraInfo` class, which validates individual JSP tags.)

Next, we set the value of the component:

```
if (value != null)
{
    if (isValueReference(value))
    {
        navigator.setValueBinding("value",
                                  app.createValueBinding(value));
    }
    else
```

```

    {
        throw new IllegalArgumentException(
            "The value property must be a value" +
            "binding expression that points to a " +
            "NavigatorItemList object.");
    }
}
}

```

First, if the `value` property of the tag handler isn't null, we either create a new `ValueBinding` instance and set it on the component, or throw an exception if the `value` isn't a value-binding expression.

The final step is to add the normal attributes:

```

Util.addAttribute(app, navigator, "headerClass", headerClass);
Util.addAttribute(app, navigator, "itemClass", itemClass);
Util.addAttribute(app, navigator, "selectedItemClass",
    selectedItemClass);
Util.addAttribute(app, navigator, "iconClass", iconClass);
Util.addAttribute(app, navigator, "layout", layout);
}

```

This adds each of the additional attributes specified in table 19.3. `Util.addAttribute` simply adds the specified attribute if it's non-null, creating and setting a `ValueBinding` instance if necessary.

The final task is the exciting `release` method:

```

public void release()
{
    super.release();

    headerClass = null;
    itemClass = null;
    selectedItemClass = null;
    headerLabel = null;
    layout = null;
    value = null;
    immediate = null;
    iconClass = null;
    actionListener = null;
}

```

Here, we simply clear all of the instance variables used for this tag.

Now that we've seen `Navigator_ToolbarTag`, let's examine the tag handler for individual `NavigatorItem` instances.

19.6.2 Writing the `NavigatorItemTag` tag handler

When front-end developers use the `Navigator_ToolbarTag`, they have the option of either referencing a value-binding expression for the component's `value` or

configuring it via additional JSP tags. Those additional tags reference the `NavigatorItemTag` custom tag handler, which creates a new `NavigatorItem` instance and adds it to the list of items maintained by the parent `UINavigator` component. This way, developers can either reference an object already stored somewhere in the application (either in code or through the Managed Bean Creation facility), or use the tags to initialize the list.

`NavigatorItemTag` subclasses `UIComponentTag` directly, since it doesn't have to process any HTML attributes. Even though this class isn't associated with a component directly, subclassing `UIComponentTag` makes it much easier to integrate with component tags and the JSF components on the current page.

Rather than exposing properties that are also used by a component or a renderer, `NavigatorItemTag` exposes properties that map to an individual `NavigatorItem` instance. This makes sense, since the primary purpose of this tag is to create a new `NavigatorItem` instance. These are the same properties that are listed in table 19.1.

Because this tag handler isn't associated with a component or a renderer, we return `null` for the `componentType` and `rendererType` properties:

```
public String getComponentType()
{
    return null;
}

public String getRendererType()
{
    return null;
}
```

In all of the examples so far, most of the work in a tag handler has been performed in the `setProperties` method. This is because usually a tag handler is just responsible for mapping its properties to those of the associated component and renderer. Since `NavigatorItemTag`'s primary purpose is to create a new `NavigatorItem` instance based on its properties, we can perform its work in the `doStartTag` method. `UIComponentTag` usually sets up the associated component and starts the encoding process in this method, but we will focus on creating a new `NavigatorItem` instance and updating the parent `UINavigator`'s item list.

```
public int doStartTag() throws JspException
{
    FacesContext context = FacesContext.getCurrentInstance();
    Application app = context.getApplication();
    UIComponentTag parentTag = getParentUIComponentTag(pageContext);
```

```

if (parentTag == null ||
    !(parentTag instanceof Navigator_ToolbarTag))
{
    throw new JspException(
        "This tag must be nested inside a " +
        "parent Navigator_ToolbarTag.");
}

Navigator_ToolbarTag parentToolbarTag =
    (Navigator_ToolbarTag)parentTag;

```

First, we check to make sure this tag is nested inside a `Navigator_ToolbarTag` using `getParentUIComponentTag`, which is a `UIComponentTag` utility method. If not, we throw an exception. (This check can also be handled by a tag library validator.)

Once we have a handle to the parent `Navigator_ToolbarTag`, we retrieve the associated `UINavigator` component using `getComponentInstance`, which is a `UIComponentTag` method. Then, we retrieve the list of items from the `UINavigator`. If the list is null, we create it.

```

UINavigator navigator =
    (UINavigator)parentToolbarTag.getComponentInstance();
NavigatorItemList items = navigator.getItems();
if (items == null)
{
    items = new NavigatorItemList();
    navigator.setValue(items);
}

```

Creating a new `NavigatorItemList` instance ensures that a list is available for this item.

One of the key differences between working directly with a model object as opposed to a `UIComponent` is support for value-binding expressions. Whereas a `UIComponent` instance maintains a collection of `ValueBinding` instances, `NavigatorItem` knows nothing about them. So, it's up to `Navigator_ToolbarTag` to evaluate the expressions by retrieving a `ValueBinding` instance and calling the `getValue` method. We begin with the `name` property:

```

if (name != null && isValueReference(name))
{
    name = (String)app.createValueBinding(name).
        getValue(context);
}

```

Once we've evaluated the `name` property (if necessary), we proceed only if the item isn't already in the list (so that we don't overwrite any existing items):

```

if (!items.containsName(name))
{
    if (label != null && isValueReference(label))

```

```

{
    label = (String)app.createValueBinding(label).
                getValue(context);
}
if (icon != null && isValueReference(icon))
{
    icon = (String)app.createValueBinding(icon).
                getValue(context);
}
if (link != null && isValueReference(link))
{
    link = (String)app.createValueBinding(link).
                getValue(context);
}
boolean bDirect = false;
if (direct != null)
{
    if (isValueReference(direct))
    {
        bDirect = ((Boolean)app.createValueBinding(direct).
                    getValue(context)).booleanValue();
    }
    else
    {
        bDirect = Boolean.valueOf(direct).booleanValue();
    }
}
boolean bDisabled = false;
if (disabled != null)
{
    if (isValueReference(disabled))
    {
        bDisabled = ((Boolean)app.createValueBinding(disabled).
                    getValue(context)).booleanValue();
    }
    else
    {
        bDisabled = Boolean.valueOf(disabled).booleanValue();
    }
}

items.add(new NavigatorItem(name, label, icon, action, link,
                            bDirect, bDisabled));
}
return getDoStartValue();
}

```

Most of the work here is evaluating value-binding expressions. The real work is the last two lines (in bold) that create a new `NavigatorItem` instance and add it to the list.

Afterwards, we return the result of the `getDoStartValue` method. `getDoStartValue` is a protected method of `UIComponentTag` that makes it easy for subclasses to change the value returned from the `doStartTag` method. For us, the default value is fine.

There's a corresponding `doEndValue` method that performs the same task for the `doEndTag` method. Our `doEndTag` implementation simply calls that method:

```
public int doEndTag() throws JspException
{
    return getDoEndValue();
}
```

It's important to override `doEndTag` in this case because we want to avoid `UIComponentTag`'s default processing, which assumes that there's an associated component and tries to decode it.

TIP Don't forget to call the `getDoStartValue` or `getDoEndValue` method when you're overriding the `doStartTag` or `doEndTag` method of `UIComponentTag`. If you return a specific value, you're breaking `UIComponentTag`'s guarantee that the values from those properties will always be returned. If you want to return a specific value, override `getDoStartValue` or `getDoEndValue` explicitly. (For more information about `UIComponentTag`, see chapter 15.)

The last step is the obligatory `release` method:

```
public void release()
{
    super.release();

    name = null;
    label = null;
    icon = null;
    action = null;
    link = null;
    direct = null;
    disabled = null;
}
```

Nothing exciting here; we just reset all of the variables to `null`.

That's all there is for `NavigatorItemTag`. Now that we've covered the Java code for both tags, we can examine the XML tag library descriptors for them.

19.6.3 Adding the tags to the tag library

`Navigator_ToolbarTag` doesn't require anything fancy in its tag library descriptor. The descriptor does, however need to expose all of the HTML pass-through attributes defined in the tag handler's superclasses (`HtmlBaseTag` and `HtmlTableBaseTag`). The relevant portions of the source are shown in listing 19.3.

Listing 19.3 The tag library descriptor for `Navigator_ToolbarTag`

```
<?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>
  <!-- Tag Library Description Elements -->
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>JSF in Action Custom Tags</short-name>
  <uri>jsf-in-action-components</uri>
  <description>
    Sample custom components, renderers, validators, and converters
    from JSF in Action method.
  </description>
  <!-- Tag declarations -->
  ...
  <tag>
    <name>navigatorToolbar</name>
    <tag-class>
      org.jia.components.taglib.Navigator_ToolbarTag
    </tag-class>
    <body-content>JSP</body-content>
    <attribute>
      <name>headerClass</name>
      <required>false</required>
      <rtpexprvalue>false</rtpexprvalue>
    </attribute>
    <attribute>
      <name>headerLabel</name>
      <required>false</required>
      <rtpexprvalue>false</rtpexprvalue>
    </attribute>
    <attribute>
      <name>itemClass</name>
      <required>false</required>
      <rtpexprvalue>false</rtpexprvalue>
    </attribute>
    <attribute>
      <name>selectedItemClass</name>
      <required>false</required>
      <rtpexprvalue>false</rtpexprvalue>
    </attribute>
  </tag>
</taglib>
```

```
</attribute>
<attribute>
    <name>styleClass</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>iconClass</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>immediate</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>layout</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>value</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>id</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>rendered</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>binding</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
...
<attribute>
    <name>cellpadding</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
</attribute>
<attribute>
    <name>width</name>
    <required>false</required>
    <rteprvalue>false</rteprvalue>
```

```
</attribute>
</tag>
</taglib>
```

The descriptor for the `NavigatorItemTag` is quite simple, since there are no extra HTML attributes to expose. It's shown in listing 19.4.

Listing 19.4 The tag library descriptor for the `NavigatorItemTag` (the top-level `taglib` element has been excluded)

```
<tag>
  <name>navigatorItem</name>
  <tag-class>
    org.jia.components.taglib.NavigatorItemTag
  </tag-class>
  <body-content>JSP</body-content>
  <attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>label</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>icon</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>link</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>action</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>direct</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>disabled</name>
    <required>false</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
```

```
</attribute>
</tag>
```

The only interesting thing to note here is that the `name` attribute is required because it's the item's identifier.

Now that we've fully integrated this component and renderer with JSP, it's time to actually use it.

19.7 Using the component

One of the primary reasons for developing this component was to show how to support a custom model class, which is the `NavigatorItemList`, in this case. The list can be initialized either in code, via the Managed Bean Creation facility, or via the `NavigatorItemTag`. Let's look at how it can be initialized in code:

```
FacesContext context = FacesContext.getCurrentInstance();
ValueBinding binding = (ValueBinding)context.getApplication().
    createValueBinding("#{sessionScope.items}");
NavigatorItemList items = (
    NavigatorItemList)binding.getValue(context);
if (items == null)
{
    String action = "#{testForm.toggleNextOrPrevious}";
    items = new NavigatorItemList();
    items.add(new NavigatorItem("inbox", "Inbox",
        "images/inbox.gif", action,
        null, false, false));
    items.add(new NavigatorItem("showAll", "Show all",
        "images/show_all.gif", action,
        null, false, false));
    items.add(new NavigatorItem("createNew", "Create new",
        "images/inbox.gif", action,
        null, false, false));
    items.add(new NavigatorItem("logout", "Logout",
        "images/logout.gif", action,
        null, false, false));
    binding.setValue(context, items);
}
```

This code creates a session-scoped variable with the key `items`. If no object is available for the binding, it creates a new `NavigatorItemList` and adds four `NavigatorItem` instances to it. Note that all of them reference the same action method—`testForm.toggleNextPrevious`. This is a simple method that toggles between an outcome of "next" and "previous". Once the new list has been created and populated, we update the value binding with it.

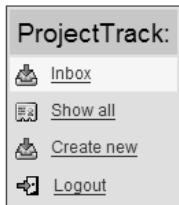


Figure 19.3
UINavigator referencing a scoped variable with vertical layout and CSS styles.

Now that we've created a `NavigatorItemList` and made it available, we can reference it using the component tag:

```
<jia:navigatorToolbar id="nav1" layout="VERTICAL"
    styleClass="navigator2"
    itemClass="navigator-command"
    selectedItemClass="navigator-selected-command"
    headerClass="navigator-title"
    iconClass="navigator-command-icon"
    value="#{items}" headerLabel="ProjectTrack:" />
```

This displays a new toolbar with a vertical layout and a header that says “ProjectTrack:”, and that references the list we stored under the key `items`. This is the tag that generated the HTML shown in listing 19.2. Figure 19.3 shows what it looks like in a browser.

Note that the first item in figure 19.3 is highlighted. This is because it's the currently selected item, and consequently the `ToolbarRenderer` has displayed it with the CSS style `navigator-selected-command` instead of `navigator-command`. If the user clicks on any of these links, it will execute the `toggleNextOrPrevious` action of `testForm`.

Let's look at an example with a horizontal layout, an explicitly declared header facet, and items configured via custom tags instead of code:

```
<jia:navigatorToolbar id="nav2"
    layout="HORIZONTAL"
    styleClass="navigator2"
    itemClass="navigator-command"
    selectedItemClass="navigator-selected-command"
    headerClass="navigator-title"
    iconClass="navigator-command-icon">
<f:facet name="header">
    <h:outputText value="ProjectTrack:" />
</f:facet>
<jia:navigatorItem name="inbox" label="Inbox"
    icon="images/inbox.gif"
    action="#{testForm.incrementCounter}"
    disabled="true"/>
<jia:navigatorItem name="showAll" label="Show All"
    icon="images/show_all.gif"
    action="#{testForm.incrementCounter}" />
```

```
<jia:navigatorItem name="createNew" label="Create New"
    icon="images/create.gif"
    action="#{testForm.incrementCounter}"
    disabled="true"/>
<jia:navigatorItem name="logout" label="Logout"
    icon="images/logout.gif"
    action="#{testForm.incrementCounter}"/>
</jia:navigatorToolbar>
```

In this example, we've added an explicit header facet. In the previous example, this wasn't necessary, because using the `headerLabel` attribute told the tag handler to create the facet for us. The two methods are semantically equivalent: they both create a facet that contains a single `UIOutput` component with the value "ProjectTrack:".

After the facet, this example includes several `<jia:navigatorItem>` tags to add individual `NavigatorItem` instances to the list. Each item references the `action` method `testForm.incrementCounter`, which simply increments the value of the `testForm.counter` property by 1. Also, note that the first and third items have the `disabled` property set to `true`. Figure 19.4 shows how this declaration looks in a browser.

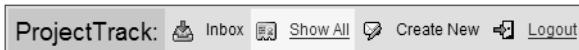


Figure 19.4 UINavigator with hardcoded items, CSS styles, and a horizontal layout (the first and third items are disabled and the second item is selected).

The header has the same text, and all of the items have the same name, but the first and third items have no hyperlink (because they are disabled), the second item is selected, and the table is a single row instead of five rows (since the layout is horizontal). Also, when the user clicks on one of the items, it will execute the `testForm.incrementCounter` action method instead of `testForm.nextOrPrevious`.

All of the items don't have to execute the same action method, though. The following example has some items that behave quite differently from each other:

```
<jia:navigatorToolbar id="nav3" layout="vertical"
    styleClass="navigator2"
    itemClass="navigator-selected-command"
    selectedItemClass="navigator-command"
    headerClass="navigator-title"
    iconClass="navigator-command-icon">

    <f:facet name="header">
        <h:outputText value="UINavigator Demo"/>
    </f:facet>

    <jia:navigatorItem name="item1" label="Item 1 (no action)">
        <f:facet name="content">
            <h:outputText value="This item does nothing." />
        </f:facet>
    </jia:navigatorItem>

    <jia:navigatorItem name="item2" label="Item 2 (increments counter)">
        <f:facet name="content">
            <h:commandButton value="Click Me!" action="#{testForm.incrementCounter}" />
        </f:facet>
    </jia:navigatorItem>

    <jia:navigatorItem name="item3" label="Item 3 (increments counter)">
        <f:facet name="content">
            <h:commandButton value="Click Me!" action="#{testForm.incrementCounter}" />
        </f:facet>
    </jia:navigatorItem>
</jia:navigatorToolbar>
```

```

        icon="images/show_all.gif"/>
<jia:navigatorItem name="item2"
    label="Item 2 (increment counter action)"
    icon="images/show_all.gif"
    action="#{testForm.incrementCounter}"/>
<jia:navigatorItem name="item3"
    label="Item 3 (toggle page action)"
    icon="images/show_all.gif"
    action="#{testForm.toggleNextOrPrevious}"/>
<jia:navigatorItem name="google" label="Google (direct link)"
    direct="true" link="http://www.google.com"/>
<jia:navigatorItem name="manning" label="Manning"
    icon="/images/manning_logo.gif"
    link="http://www.manning.com"/>
<jia:navigatorItem name="jsfcentral" label="JSF Central"
    icon="/images/jsf_logo_small.gif"
    link="http://www.jsfcentral.com"/>
<f:action_listener
    type="org.jia.components.NavigatorLoggingActionListener"/>
</jia:navigatorToolbar>

```

The first item in this example doesn't reference an action method at all. The next two items reference different actions—`testForm.incrementCounter` and `testForm.toggleNextOrPrevious`, respectively. The fourth item doesn't have an icon, it has the `direct` property set to `true`, and it references an external URL. The final two items reference external URLs as well. Also, note that we added an `ActionListener` called `NavigatorLoggingActionListener`. Figure 19.5 shows what this toolbar looks like in a browser.

We reversed the styles for the selected and unselected items to spice things up; Item 2 is the one that's selected. Clicking on Item 1 won't fire an action, but it will cause the item to be selected. Clicking on Item 2 will execute `testForm.incrementCounter`, and Item 3 will cause `testForm.nextPrevious` to be executed.

Clicking on the Google link has an entirely different effect; it's a direct link to `www.google.com` that doesn't cause a postback to the component itself. The last two links do, however, post back to the server, and result in executing

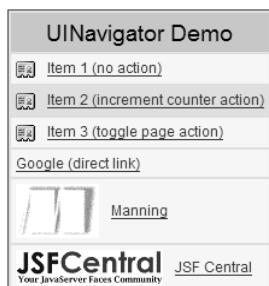


Figure 19.5

UINavigator with vertical layout, CSS styles, and a combination of items that reference direct links and action methods (the second item is selected).

`NavigationActionHandler` to redirect the user to www.manning.com (this book's publisher), and www.jsfcentral.com (the author's JSF community site), respectively.

Allowing the component to perform a postback is useful if you want to log the user's selected item. This is where the `NavigatorLoggingActionListener` comes in. It simply outputs the selected item to the log file. A real application might register an `ActionListener` that logs such actions to a database. This would allow it to keep track of its user's activity (if you're working for Big Brother) or even adapt the system's behavior based on usage patterns.

19.8 Summary

The goal of this chapter was to build the toolbar used in ProjectTrack. We developed a model-driven component, `UINavigator`, that allows a user to select one of many possible items. We wrote the `NavigatorItem` class, which represents an individual item. We also wrote an `ArrayList` subclass, `NavigatorItemList`, which maintains a list of `NavigatorItem` instances and keeps track of the currently selected one. `UINavigator` is essentially a component wrapper around a `NavigatorItemList` instance.

`UINavigator` implements the `ActionSource` interface, so it's a source of action events. There's also a private `ActionListener` implementation, `NavigatorActionListener`, that's automatically registered for each `UINavigator` instance. This listener is responsible for either executing the `action` property or redirecting the user to the `link` property of the selected `NavigatorItem`.

To display a `UINavigator`, we wrote a `ToolbarRenderer` that displays all of the items in an HTML table, complete with CSS styles and layout options. In order to properly understand which item the user selects, `ToolbarRenderer` uses JavaScript and hidden fields, and also references the parent form on the page.

For JSP integration, we developed two custom tag handlers: `Navigator_ToolbarTag` for the component, and `NavigatorItemTag` for individual items. `NavigatorItemTag` is unique because it doesn't map to a specific component tag; it's only used for creating and configuring new `NavigatorItem` instances. Front-end developers will typically nest several `NavigatorItemTags` inside a single `Navigator_ToolbarTag`.

The final result is a highly flexible, reusable navigation component with a familiar toolbar representation. A developer can configure the list of items in code, via the Managed Bean Creation facility, or through custom tags. Each item can point to an action or an external link, and additional action listeners to react to a user's selection.

This completes our study of components and renderers. Next, we'll take a closer look at building validators and converters.

20

Validator and converter examples

This chapter covers

- When to write a validator
- Developing a validator
- When to write a converter
- Developing a converter

Now that we've completed our whirlwind tour of the component development world, it's time to take a look at the JSF user interface helpers: validators and converters. Developing either one is pretty similar to developing user interface components, but much easier. The process involves implementing a simple interface and registering the new class with the application.

JSP integration is the same: subclass a JSF custom tag handler, override a couple of methods, and then add it to the tag library definition (TLD). Because neither components nor validators are associated with JSF's rendering mechanism, there's no need to worry about developing renderers. This chapter assumes that you're familiar with the validator and converter classes and interfaces; refer back to chapter 15 for an overview.

20.1 Validator methods vs. validator classes

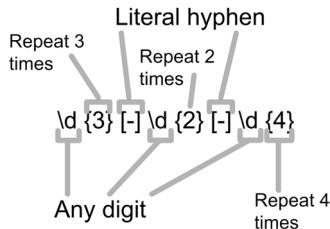
In chapter 13, we showed you how to write methods in your backing beans that perform validation. For most application logic, this is perfectly acceptable and even encouraged, especially if there's no need for reuse across dissimilar pages. (You can reuse a validator method across similar pages if you create a hierarchy of backing beans that are specialized for different pages, or reuse the same backing bean in multiple pages.) However, for cases where you want validation logic to be separate from application logic (for example, a credit card validator), and cases where you are developing a library for resale or distribution (externally or internally), stand-alone validators are the way to go.

20.2 Developing a validator

Out of the box, JSF has several standard validators for basic things like numbers and the length of a string. That doesn't help with such things as a social security or a phone number. Desktop RAD tools usually handle this with a MaskEdit control of some sort, but in the JSF world, this can be done with a validator. Instead of defining a specialized edit mask format, however, we might as well use regular expressions.

NOTE

Remember, JSF does not have explicit support for client-side validation. If you want to write a validator that emits JavaScript as well, you have two choices: use the Struts validator (see chapter 14), or write a `UIComponent` that outputs JavaScript based on the validators on the page. (Validators can't output anything, so you must write a Component instead.)

**Figure 20.1**

Regular expression for a U.S. social security number. Each “ \d ” represents group of possible characters, or *character class*, that can be any digit. The curly braces match the previous class a specific number of times. So, the string “ $\d{3}$ ” matches any three digits, such as 123, 234, or 456. The brackets define specific character class, which in this case, has only one valid character: a hyphen. So, literally the expression means any string that starts with three digits, then has a hyphen, two digits, another hyphen, and ends with four digits. This would match any string in the format ###-# #-###, where # is any digit. Specific examples are 123-45-6789 or 555-55-5555.

Regular expressions are special strings that can be used to match patterns in text. They've been around for quite a while, and are the cornerstone of the Unix command line and languages like Perl. From our point of view, they're a great way to create an extremely versatile validator that can be used for not only a social security or phone number but just about anything else.

Each character in a regular expression has a special meaning about the characters it will match in an input string. For example, the character “.” matches any character. The character “*” will match the preceding set of characters any number of times. So the regular expression “*” matches any string. Figure 20.1 shows a regular expression for a U.S. social security number. For a thorough explanation, see *Mastering Regular Expressions* [Friedl].

Figure 20.2 shows our validator in action, validating a U.S. social security number, a U.S. phone number, and an email address.

The screenshot shows a Mozilla Firefox browser window with the title "JSF in Action - RegularExpressionValidator Custom Validator - Mozilla {Build ID: 2004031...}". The main content is a page titled "RegularExpressionValidator". It features three text input fields:

- "Enter a social security number:" with the value "555-55-5555".
- "Enter a phone number:" with the value "555 555 555" and the error message "Invalid phone number. Format is (###)###-####."
- "Enter an e-mail address:" with the value "nobody" and the error message "Invalid e-mail address."

At the bottom left is a "Validate it!" button, and at the bottom center is a "Done" link.

Figure 20.2 The `RegularExpressionValidator` in action for validating a U.S. social security number, a U.S.-style phone number (with optional area code), and an email address. Error messages can be customized, and displayed via an `HtmlMessage` or `HtmlMessages` component.

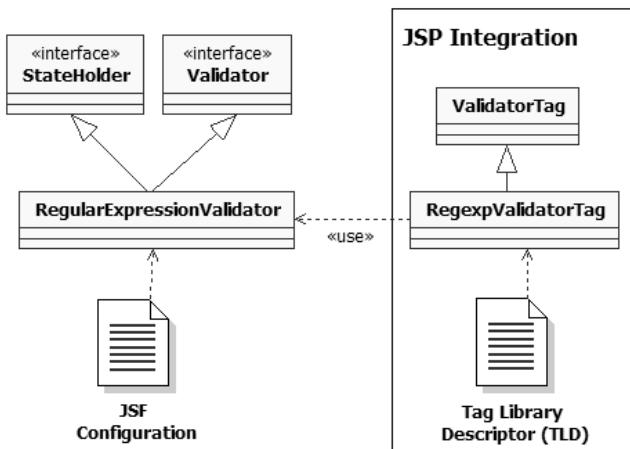


Figure 20.3 Elements for the RegularExpressionValidator.
 The primary class is `RegularExpressionValidator`, which implements both the `Validator` and `StateHolder` interfaces. A single tag handler, `RegexpValidatorTag`, provides JSP integration for the validator.

To build a regular expression validator, we'll need two classes: `RegularExpressionValidator` which provides the core functionality, and the `RegexpValidatorTag` class, which is the JSP tag handler. Also, since `RegularExpressionValidator` has properties that must be remembered between requests, it implements the `StateHolder` interface. We'll also need to register the validator in a JSF configuration file, and the tag handler in a tag library definition (TLD) file. Figure 20.3 shows all of the necessary elements.

Let's get started with the validator class itself.

20.2.1 Writing the `RegularExpressionValidator` class

The idea behind this validator is simple: the front-end developer associates it with an input control and specifies a regular expression and an error message. If the control's `value` property (which must convert to a `String`) matches the regular expression, we do nothing. Otherwise, we throw a `ValidatorException` with a new `FacesMessage` containing the error message specified by the developer. The message can then be displayed with a `Htmlmessage` or `Htmlmessages` component.

This boils down to the two properties listed in table 20.1. The validator has an `errorMessage` property for situations where the developer wants to customize the validation error message that the user sees. This is important, because `RegularExpressionValidator` doesn't know what the regular expression represents—it just

knows how to match input against it. With the `errorMessage` property, the developer can replace the default string “Validation Error: This field is not formatted properly.” with something a little more user friendly, like “Your phone number is not in the correct format.”

Table 20.1 RegularExpressionValidator properties

Property	Type	Description	Required?
<code>expression</code>	<code>String</code>	Regular expression to match the input against.	Yes
<code>errorMessage</code>	<code>String</code>	Message to be displayed if there is a validation error.	No

Like all validators, `org.jia.validators.RegularExpressionValidator` implements the simple `Validator` interface. Also, to maintain state, it implements the `StateHolder` interface. The source is shown in listing 20.1.

Listing 20.1 RegularExpressionValidator.java: Validates input against a regular expression

```
package org.jia.validators;

import javax.faces.application.FacesMessage;
import javax.faces.component.StateHolder;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;

import java.util.regex.Pattern;    ←① Import Pattern for
                                regular expressions
public class RegularExpressionValidator
    implements Validator, StateHolder
{
    public static final String VALIDATOR_ID =
        "jia.RegularExpressionValidator";    ↗② Implement
                                             StateHolder to
                                             save properties
    private String expression;
    private String errorMessage;

    private boolean transientFlag;
    protected Pattern pattern;

    public RegularExpressionValidator()
    {
        this(null, null);
    }

    public RegularExpressionValidator(String expression,
                                     String errorMessage)
                                ↗③ Declare the
                                identifier
```

```

{
    super();
    transientFlag = false;           ←④ Set transient flag to
    setExpression(expression);      false to save state
    setErrorMessage(errorMessage);
}

// Validator methods

public void validate(FacesContext context,
                      UIComponent component, Object inputValue)
    throws ValidatorException
{
    EditableValueHolder input = (EditableValueHolder)component;
    String value = null;
    try
    {
        value = (String)inputValue;
    }
    catch (ClassCastException e)
    {
        throw new ValidatorException(
            new FacesMessage(
                "Validation Error: Value " +
                "cannot be converted to String.",
                null));
    }

    if (!isValid(value))
    {
        String messageText = errorMessage;
        if (messageText == null)
        {
            messageText = "Validation Error: " +
                "This field is not formatted " +
                "properly.";
        }
        throw new ValidatorException(
            new FacesMessage(messageText,
                null));
    }
}

// Protected methods

protected boolean isValid(String value)
{
    if (pattern == null)
    {
        throw new NullPointerException(
            "The expression property hasn't " +

```

④ Set transient flag to false to save state

⑤ If value isn't a String, validation fails

⑥ Validate and throw exception if it fails

⑦ Place validation logic in separate method

```
        " been set.");
    }

    return pattern.matcher(value).matches();
}

// Properties

public String getErrorMessage()
{
    return errorMessage;
}

public void setErrorMessage(String errorMessage)
{
    this.errorMessage = errorMessage;
}

public String getExpression()
{
    return expression;
}

public void setExpression(String expression)
{
    this.expression = expression;
    if (expression != null)
    {
        pattern = Pattern.compile(expression);
    }
}

// StateHolder methods

public void setTransient(boolean transientValue)
{
    this.transientFlag = transientValue;
}

public boolean isTransient()
{
    return transientFlag;
}

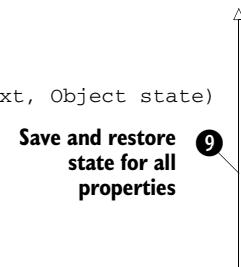
public Object saveState(FacesContext context)
{
    Object[] values = new Object[3];
    values[0] = pattern;
    values[1] = expression;
    values[2] = errorMessage;
}
```

The diagram consists of three numbered callouts with arrows pointing to specific code snippets:

- 7 Place validation logic in separate method**: Points to the line `return pattern.matcher(value).matches();`.
- 8 Create Pattern instance from expression**: Points to the line `pattern = Pattern.compile(expression);`.
- 9 Save and restore state for all properties**: Points to the line `Object[] values = new Object[3];`.

```
        return values;
    }

    public void restoreState(FacesContext context, Object state)
    {
        Object[] values = (Object[])state;
        pattern = (Pattern)values[0];
        expression = (String)values[1];
        errorMessage = (String)values[2];
    }
}
```



The diagram shows a callout arrow originating from the text "Save and restore state for all properties" and pointing towards the line of code where the "values" array is assigned.

- ➊ We'll use the standard Java regular expression library to do all of the real work of this validator. The `Pattern` class is the primary class in the `java.util.regex` package, so we must import it.
- ➋ All validators must implement the `Validator` interface, and since this class has state that we want to maintain, we must also implement the `StateHolder` interface.
- ➌ Like every other JSF user interface extension, Validators have identifiers. This constant declares the standard identifier for this class. We'll use this constant in the corresponding JSP tag handler, and we'll use its value in the application configuration file.
- ➍ The `StateHolder` interface defines a `transient` property, which should be `false` if we want to save state.
- ➎ If for some reason the value of the associated `EditValueHolder` control can't be converted to a `String`, we can't treat it as a regular expression. So, this counts as a validation error, and we throw a `ValidatorException` with a new `FacesMessage` instance. The message can be displayed back to the user by using an `HtmlMessage` or `HtmlMessages` component. Note that if there was a converter associated with this control, it would have been called before this validator.
- ➏ If the `EditValueHolder` control's `value` property doesn't match our validator's regular expression, we throw a new `ValidatorException`. The text of the `FacesMessage` instance for this exception is based on the value of the `errorMessage` property. If the property has been set, we create a new `FacesMessage` based on that property; otherwise, we create one based on a default error message. These messages can be displayed back to the user by using an `HtmlMessage` or `HtmlMessages` component. Note that if the value is valid, we do nothing.
- ➐ The `isValid` method performs the real validation check. All it does is delegate to the `Pattern` instance, which was created based on the validator's `expression` property. Note that this method's signature doesn't include any JSF classes.

TIP

It's generally a good idea to put the core converter logic in a separate method that doesn't use any runtime JSF classes. (`FacesMessage` is okay, because it's a simple JavaBean with no runtime dependencies.) This makes unit testing easier, because the tests aren't based on the validator method, and consequently don't require a web container or mock objects to create the JSF environmental objects. The sample code for this book (available at www.manning.com/mann) includes JUnit test cases that take advantage of this fact.

- ❸ When the `expression` property is set, we create a new `Pattern` object from it for use in the `isValid` method (❷). Compiling regular expressions is an expensive thing to do, so it's worthwhile to do it up-front. A new `Pattern` instance will be created only if the `expression` property is changed again.
- ❹ In the `StateHolder` state management methods, we save and restore the state for all of the properties, as well as the cached `Pattern` instance.

That's all there is to it for `RegularExpressionValidator`. Now, let's tell JSF about it.

20.2.2 Registering the validator

Registering this validator with JSF requires a `<validator>` element with child `<validator-id>` and `<validator-class>` elements. Here's a sample entry:

```
<validator>
    <description>
        Validates an input control based on a regular expression.
    </description>
    <validator-id>jia.RegularExpressionValidator</validator-id>
    <validator-class>
        org.jia.validators.RegularExpressionValidator
    </validator-class>
</validator>
```

This registers the class `org.jia.validators.RegularExpressionValidator` with the validator identifier `jia.RegularExpressionValidator` and provides a simple description for use with tools (or for those perusing a configuration file). This entry can be placed in any JSF configuration file.

That's it for JSF registration. Now, let's look at integrating our new validator with JSP.

20.2.3 Integrating with JSP

Just as with components, integrating validators with JSP requires developing a custom tag handler. However, the process is simpler because there are no renderers. This means we don't need to worry about renderer attributes, including HTML

pass-through attributes. After we've developed the tag handler, we'll register it with the tag library.

Writing the RegexpValidatorTag class

For `RegularExpressionValidator`, all we need is a single tag handler that supports its `expression` and `errorMessage` properties (described in table 20.1). The handler will be responsible for setting these properties on a new `RegularExpressionValidator` instance based on its own properties (which are exposed as attributes in JSP). Just as we did with component tag handlers, we'll also support value-binding expressions for these properties. The source for `org.jia.validators.taglib.RegexpValidatorTag` is shown in listing 20.2.

Listing 20.2 `RegexpValidatorTag.java`: The tag handler for `RegularExpressionValidator`

```
package org.jia.validators.taglib;

import org.jia.util.Util;
import org.jia.validators.RegularExpressionValidator;

import javax.faces.application.Application;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.webapp.ValidatorTag;

import javax.servlet.jsp.JspException;

public class RegexpValidatorTag extends ValidatorTag
{
    private String expression;
    private String errorMessage;

    public RegexpValidatorTag()
    {
        super();
        setValidatorId(
            RegularExpressionValidator.
            VALIDATOR_ID);
    }

    protected Validator createValidator() ←③ Override
        throws JspException
    {
        RegularExpressionValidator validator =
            (RegularExpressionValidator)
        super.createValidator();
    }
}
```

The code is annotated with four numbered callouts:

- ① **Extend ValidatorTag**: Points to the line `extends ValidatorTag`.
- ② **Set the validatorId in constructor**: Points to the line `setValidatorId(RegularExpressionValidator.VALIDATOR_ID);`.
- ③ **Override createValidator method**: Points to the line `protected Validator createValidator()`.
- ④ **Retrieve validator**: Points to the line `super.createValidator();`.

```

FacesContext context = FacesContext.getCurrentInstance();
Application app = context.getApplication();

if (expression != null)
{
if (Util.isBindingExpression(expression))

{
    validator.setExpression(
        (String)app.createValueBinding(
            expression).getValue(context));
}
else
{
    validator.setExpression(expression);
}
}
if (errorMessage != null)
{
    if (Util.isBindingExpression(errorMessage))
    {
        validator.setErrorMessag
        (String)app.createValueBinding(
            errorMessage).getValue(context));
    }
else
{
    validator.setErrorMessag(errorMessage);
}
}

return validator;
}

public void release()
{
super.release();
expression = null;
errorMessage = null;
}

// Properties

public String getExpression()
{
    return expression;
}

public void setExpression(String expression)
{
    this.expression = expression;
}

```

5 Set validator properties based on tag properties

6 Override release method

```

    }

    public String getErrorMessage()
    {
        return errorMessage;
    }

    public void setErrorMessage(String errorMessage)
    {
        this.errorMessage = errorMessage;
    }
}

```

- ➊ All validator tag handlers must extend the `ValidatorTag` class.
- ➋ Instead of overriding a method like `getComponentType` (as is the case with subclassing `UIComponentTag`), you call the `setValidatorId` method in the tag handler's constructor. This associates the tag handler with a specific validator class. In this case, we use the `VALIDATOR_ID` constant defined by `RegularExpressionValidator`.
- ➌ When you subclass `UIComponentTag`, you override `setProperties`, which is where all of the action takes place. With `ValidatorTag` subclasses, you override `createValidator` instead. This method is responsible for creating and configuring the proper `Validator` instance.
- ➍ `ValidatorTag`'s `createValidator` method is responsible for creating the initial `Validator` instance based on the `validatorId` property. Before you begin configuring a new validator instance inside `createValidator`, you should retrieve it from the superclass's implementation of the method.
- ➎ Here we set the `expression` and `errorMessage` properties on the `Validator` instance based on the corresponding properties of the tag handler itself. We support value-binding expressions by simply evaluating them directly and setting the property based on the result. (`Util.isValueBinding` simply checks to make sure the string starts with “#{” and ends with “}”.)

NOTE

The `Validator` class doesn't maintain a collection of `ValueBinding` instances like `UIComponentBase` does. This is because unlike components, whose value bindings may be executed at a later time, validator properties represent configuration parameters that are unlikely to change after they're initially set. Consequently, it's okay to evaluate value-binding expressions for validators in the tag handler.

- ➏ As is the case with any tag handler subclass, we override the `release` method and reset any instance variables.

As you can see, implementing a validator tag handler is a simple process. Now, let's add it to the tag library.

Adding the tag to the tag library

The tag library entry for `RegexpValidatorTag` is pretty straightforward. It's shown in listing 20.3.

Listing 20.3 Tag library entry for `RegexpValidatorTag`

```
<?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>
    <!-- Tag Library Description Elements -->
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>JSF in Action Custom Tags</short-name>
    <uri>jsf-in-action-components</uri>
    <description>
        Sample custom components, renderers, validators, and
        converters from JSF in Action.
    </description>
    <!-- Tag declarations -->
    ...
    <tag>
        <name>validateRegexp</name>
        <tag-class>
            org.jia.validators.taglib.RegexpValidatorTag
        </tag-class>
        <body-content>JSP</body-content>
        <attribute>
            <name>expression</name>
            <required>true</required>
            <rteprvalue>false</rteprvalue>
        </attribute>
        <attribute>
            <name>errorMessage</name>
            <required>false</required>
            <rteprvalue>false</rteprvalue>
        </attribute>
    </tag>
    ...
</taglib>
```

The code in listing 20.3 registers a tag called `<validateRegexp>` for the class `org.jia.validators.taglib.RegexpValidatorTag`. We've exposed the two properties,

expression and errorMessage, and expression is required (you can't match against an expression that isn't set).

Now that we've registered the tag, let's use our new validator.

20.2.4 Using the validator

Using our new RegularExpressionValidator is simple. We just have to register it with an input component:

```
<h:inputText id="SSNInput">
    <jia:validateRegexp expression="\d{3}[-]\d{2}[-]\d{4}" />
</h:inputText>
```

This registers a new RegularExpressionValidator with the `HtmlInputText` control, configured to match against the regular expression for a U.S. social security number. (This is the same regular expression shown in figure 20.1.) Note that we have escaped the backslashes, which is a requirement for Java regular expressions. Because the errorMessage attribute wasn't specified, the default error message “Validation Error: This field is not formatted properly” will be displayed if the `HtmlInputText` component's value doesn't match the expression. The error message can be displayed with the following tag:

```
<h:message for="SSNInput" styleClass="error" />
```

Note that the `for` attribute matches the identifier of the `UIInput` control. We've also added a CSS style to spice things up a bit.

For cases when the standard message is a little too unfriendly, we can specify the `errorMessage` attribute:

```
<h:inputText id="phoneInput" value="#{testForm.string}">
    <jia:validateRegexp
        expression="((\d{3})?)|(\d{3}-))?\d{3}-\d{4}"
        errorMessage=
            "Invalid phone number. Format is (###)###-####." />
</h:inputText>
```

Here, we use a regular expression for matching a U.S.-style phone number. The first three digits are optional, so “(555)555-5555” is valid, “555-555-555” is valid, and “555-5555” is valid. However, the dash is required in between digits, so “555 5555” is invalid. The error message is customized so that something a little more informative will be displayed if the user types in an incorrect value. Instead of using a `HtmlMessage` component to display errors, you can also use a `HtmlMessages` component:

```
<h:messages title="You had validation errors." layout="table"
    styleClass="error" />
```

Because the `for` attribute wasn't specified, this would display all validation errors for the page in a table format with a header and CSS class.

`RegexpValidatorTag` contains extra code to deal with value-binding references. So, let's say you had a regular expression available in the session:

```
FacesContext context = FacesContext.getCurrentInstance();
context.getApplication().createValueBinding(
    "#{sessionScope.emailExpression}").setValue(context,
    "\\\w+([-.]\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*");
```

This places `String` in the session under the key `emailExpression`. The `String` is the regular expression for an email address of the usual format “`name@domain.extension`”.

Moreover, let's say you had a resource bundle called `CustomComponents-Resources` in your classpath with this line:

```
emailErrorMessage=Invalid e-mail address.
```

and that you added the following line to your JSP to import the bundle:

```
<f:loadBundle basename="CustomComponentResources" var="bundle" />
```

Both can be referenced via the validator tag:

```
<h:inputText id="emailInput">
    <jia:validateRegexp expression="#{emailExpression}"
        errorMessage="#{bundle.emailErrorMessage}" />
</h:inputText>
```

This will register a `RegularExpressionValidator` that uses the expression we stored in the session, and the localized error message that was stored in the request by the `<f:loadBundle>` tag. Anytime users input an invalid email address, they will be shown a customized, localized, error message (assuming that the front-end developer uses a `HtmlMessage` or `HtmlMessages` component).

That's all there is to using our new validator class. Thanks to Java's built-in support for regular expressions, we've been able to easily develop an extremely versatile validator that can validate almost any type of string. A more sophisticated version of this component would provide default regular expressions that work for typical use cases like the ones presented here.

Fortunately, developing converters is as simple as developing validators. Let's turn our attention to that topic now.

20.3 When custom converters are necessary

In chapter 6, we surveyed the different standard converters that ship with JSF. In many cases, the standard converters will be sufficient; they handle all of the standard Java data types, and also provide sophisticated formatting functionality for dates and numbers. The main goal is to allow front-end developers to associate component data types without having to parse them or call special methods for display. You should develop custom converters any time you want to make it easy for a front-end developer to display data type, or accept input for that data type.

Let's suppose, for example, that you had written a class that represented an algebraic equation, called `Equation`. A front-end developer would have a hard time displaying your `Equation` object with a `UIOutput` component unless your `toString` method displayed something that made sense. Even if you did implement `toString`, that wouldn't help translate a user's input back into an `Equation`. Instead of developing application code to handle this chore, you could write a converter that translated a user's input string into an `Equation`. For conversion from an `Equation` to a `String`, you could also add formatting options. Writing a converter would allow you to easily use `Equation` objects with many types of JSF components in different applications, without having to rewrite the translation code each time.

20.4 Developing a converter

For our example, we'll write a converter for the `User` object developed for Project-Track. This isn't quite as exciting as a converter for an `Equation` object, but it will do for our purposes.

Our converter will make it easy to display a `User` object from a JSF component. It will also have some simple options for formatting—things like displaying the first name (“John”, for example) by itself or the last name, a comma, and then the first name (like “Mann, John”). This way a front-end developer doesn't have to worry about pulling individual properties out of the `User` object. Instead, the developer can just use the `UserConverter` and specify a style of “`lastName(firstName)`”.

Because converters work in two ways, it's also possible to create a new `User` object based on user input. So an end user could type in “John Doe” and `UserConverter` will automatically create a `User` object whose `firstName` property is “John” and whose `lastName` property is “Doe”. Figure 20.4 shows what the converter looks like in a browser.

Developing a converter is similar to developing a validator. We start with the converter class, `UserConverter`, which must be registered with JSF via a configuration file.

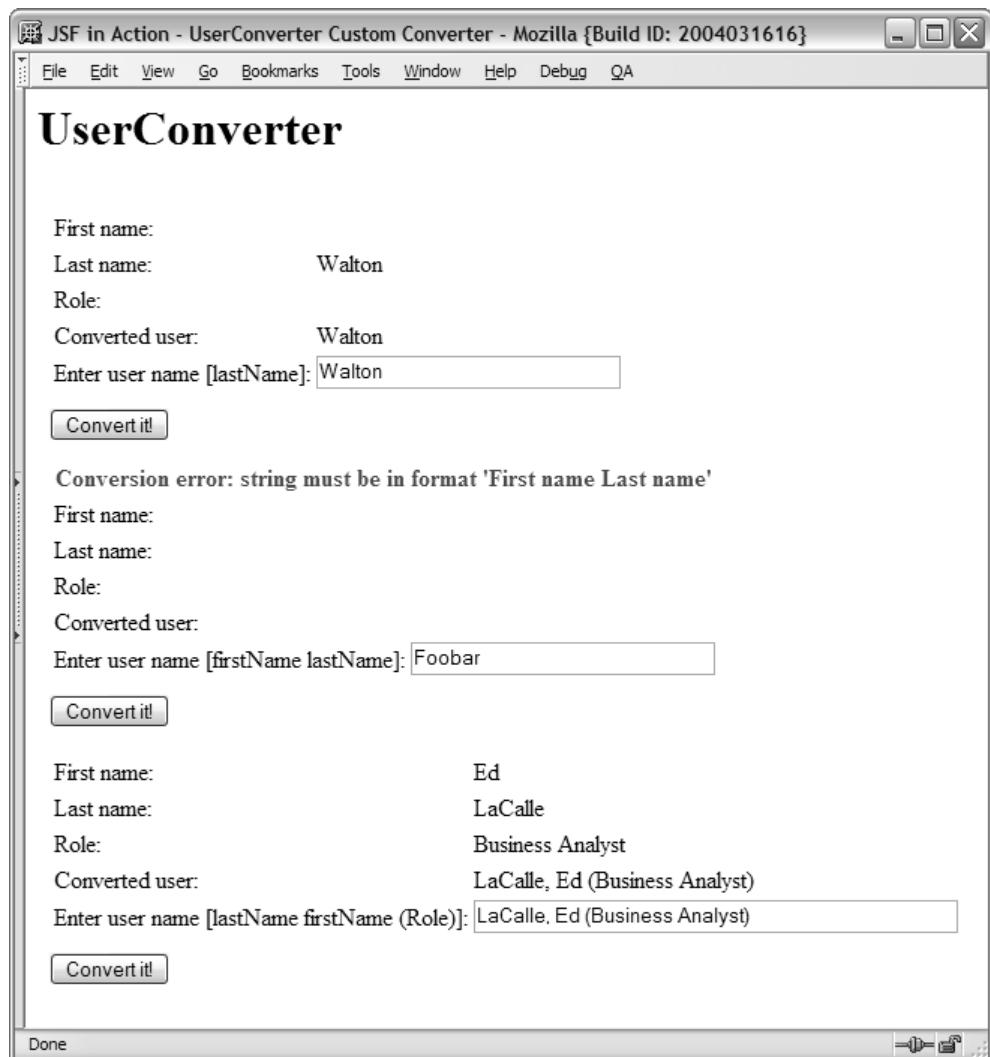


Figure 20.4 The UserConverter converts between a User and a String. This screenshot shows three different styles: last name only ("Walton"), first name and last name ("John Mann"), and last name and first name with the role displayed ("LaCalle, Ed (Business Analyst)"). The second example shows a conversion error because the user entered a value in the wrong format. Note that the value of the user object hasn't changed, though, because JSF will not update an object with an incorrect value.

Because this converter maintains state, it must implement the StateHolder interface. It can be integrated with JSP through a custom tag handler, UserConverterTag,

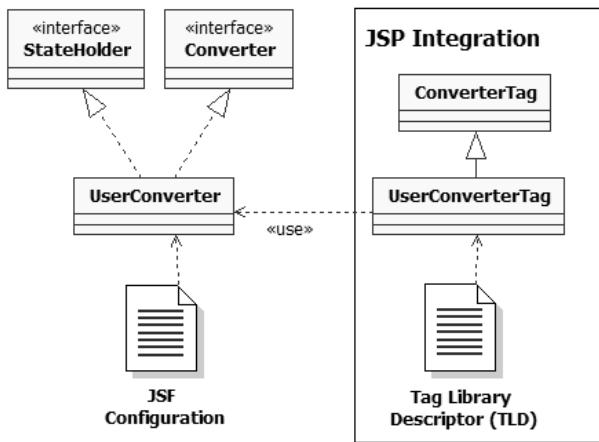


Figure 20.5
Classes and files for `UserConverter`. The `UserConverter` class implements both the `Converter` and `StateHolder` interfaces and is integrated with JSP through the `UserConverter` tag.

which must be registered in a tag library's tag library descriptor file. The necessary classes and configuration files are shown in figure 20.5.

Now, let's examine the `UserConverter` class itself.

20.4.1 Writing the `UserConverter` class

The job of `org.jia.converters.UserConverter` is to display a `User` object as a string to a user, and translate a string of user input into a `User` object. In practice, this means that we either create a new `String` object based on properties of a `User`, or parse a `String` and set properties on a newly created `User` object.

In order to make things a little interesting, `UserConverter` has two properties: `style` and `showRole`. `style` has several values that determine which combination of first and last names will be converted. The `showRole` property indicates whether or not the `User`'s role will be converted as well. These properties are described in table 20.2.

Table 20.2 `UserConverter` properties

Property	Type	Description	Required?
<code>style</code>	<code>StyleType</code>	Specifies the format for converting the first name and last names. Possible values are <code>StyleType.FIRSTNAME</code> , <code>StyleType.LASTNAME</code> , <code>StyleType.FIRSTNAME_LASTNAME</code> , and <code>StyleType.LASTNAME_FIRSTNAME</code> .	Yes
<code>showRole</code>	<code>boolean</code>	True if the role should be converted.	No

As the table shows, `style` is a `StyleType` object, which is a type-safe enumeration that's defined as a nested class. This way, when someone sets the `style`, it's impossible to set an incorrect value.

All converters must implement the `Converter` interface, and `UserConverter` is no different. It also implements the `StateHolder` interface so that it can properly save the state of its properties. The source is shown in listing 20.4.

Listing 20.4 UserConverter.java: A converter for ProjectTrack's User object

```
package org.jia.converters;

import org.jia.ptrack.domain.EnumeratedType;
import org.jia.ptrack.domain.EnumeratedType.EnumManager;
import org.jia.ptrack.domain.RoleType;
import org.jia.ptrack.domain.User;

import javax.faces.application.FacesMessage;
import javax.faces.component.StateHolder;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;

public class UserConverter implements
    Converter, StateHolder
{
    public final static String CONVERTER_ID =
        "jia.User";
    private boolean transientFlag;
    private boolean showRole;
    private StyleType style;

    public UserConverter()
    {
        this(StyleType.FIRSTNAME_LASTNAME, false);
    }

    public UserConverter(StyleType style, boolean showRole)
    {
        transientFlag = false; ←③ Set transient
        style = style;          flag to false to
        showRole = showRole;    save state
    }

    // Converter methods

    public Object getAsObject(
        FacesContext context,
        UIComponent component,
        String displayString)
        throws ConverterException
    {
        if (displayString == null)
            return null;
        else
            return User.getFromNameAndRole(
                displayString,
                showRole);
    }
}
```

1 Implement `StateHolder` to save properties

2 Declare identifier

3 Set transient flag to false to save state

4 Implement to return a `User` from a `String`

```

{
    User user = new User();
    FacesMessage message =
        getStringAsUser(user, displayString);
    if (message != null)
    {
        throw new ConverterException(message);
    }
    return user;
}

public String getAsString(
    FacesContext context,
    UIComponent component,
    Object object)
throws ConverterException
{
    return getUserAsString((User)object);
}

// Protected methods

protected FacesMessage getStringAsUser(
    User user,
    String displayString)
{
    FacesMessage errorMessage = null;

    if (style == StyleType.FIRSTNAME)
    {
        errorMessage = getUserFromFirstName(user, displayString);
    }
    else
    if (style == StyleType.LASTNAME)
    {
        errorMessage = getUserFromLastName(user, displayString);
    }
    else
    if (style == StyleType.FIRSTNAME_LASTNAME)
    {
        errorMessage = getUserFromFN_LN(user, displayString);
    }
    else
    if (style == StyleType.LASTNAME_FIRSTNAME)
    {
        errorMessage = getUserFromLN_FN(user, displayString);
    }

    if (errorMessage == null && showRole == true)
    {
        errorMessage = setUserRole(user, displayString);
    }
}

```

5 If there is a conversion error, throw exception

6 Implement to return a String from a User

7 Perform actual conversion from String to User

```

        }
        return errorMessage;
    }

    // Utility methods

    protected FacesMessage getUserFromFirstName( ←⑧
                                                User user, String displayString)
    {
        String[] names = displayString.split("\\s", 2);
        if (showRole != true && names.length > 1)
        {
            return new FacesMessage(
                "Conversion error: string must be in format " +
                "'First name' (one word)",
                null);
        }
        else
        {
            user.setFirstName(displayString);
        }
        return null;
    }

    protected FacesMessage getUserFromLastName(
                                                User user, String displayString)
    {
        String[] names = displayString.split("\\s", 2);
        if (showRole != true && names.length > 1)
        {
            return new FacesMessage(
                "Conversion error: string must be in format " +
                "'Last name' (one word)",
                null);
        }
        else
        {
            user.setLastName(displayString);
        }
        return null;
    }

    protected FacesMessage getUserFromFN_LN(
                                                User user, String displayString)
    {
        String[] names = displayString.split("\\s", 3);
        if (names.length < 2 || (showRole == false && names.length > 2))
        {
            return new FacesMessage("Conversion error: string must be " +
                "in format 'First name Last name'",
                null);
        }
    }
}

```

Either create an error message or set User properties

```
        }

        user.setFirstName(names[0]);
        user.setLastName(names[1]);
        return null;
    }

protected FacesMessage getUserFromLN_FN(
        User user, String displayString)
{
    String[] names = displayString.split(",\\s", 2);
    String firstNames[] = null;
    if (names.length > 1)
    {
        firstNames = names[1].split("\\s", 2);
    }
    if (names.length != 2 ||
        (showRole == false && firstNames.length > 1))
    {
        return new FacesMessage("Conversion error: string must be " +
            "in format 'Last name, First name'",
            null);
    }
    user.setLastName(names[0]);
    user.setFirstName(firstNames[0]);
    return null;
}

protected FacesMessage setUserRole(
        User user, String displayString)
{
    int startIndex = displayString.indexOf(" (");
    int endIndex = displayString.lastIndexOf(')');
    if (startIndex == -1 || endIndex == -1)
    {
        return new FacesMessage("Conversion error: no role found; " +
            "string must end in format '(Role)'",
            null);
    }
    String roleString = displayString.substring(startIndex + 2,
                                                endIndex);
    try
    {
        user.setRole((RoleType)RoleType.getEnumManager().
            getInstance(roleString));
    }
    catch (IllegalArgumentException e)
    {
        return new FacesMessage("Conversion error: invalid role. " +
            "Try capitalizing each word.",
            null);
    }
}
```

```

        return null;
    }
    protected String getUserAsString(User user) ←⑨ Perform actual conversion
    {
        StringBuffer buffer = new StringBuffer();
        if (style == StyleType.FIRSTNAME && user.getFirstName() != null)
        {
            buffer.append(user.getFirstName());
        }
        else
        if (style == StyleType.LASTNAME && user.getLastName() != null)
        {
            buffer.append(user.getLastName());
        }
        else
        if (style == StyleType.FIRSTNAME_LASTNAME)
        {
            if (user.getFirstName() != null)
            {
                buffer.append(user.getFirstName());
                buffer.append(" ");
            }
            if (user.getLastName() != null)
            {
                buffer.append(user.getLastName());
            }
        }
        else
        if (style == StyleType.LASTNAME_FIRSTNAME)
        {
            if (user.getLastName() != null)
            {
                buffer.append(user.getLastName());
                buffer.append(", ");
            }
            if (user.getFirstName() != null)
            {
                buffer.append(user.getFirstName());
            }
        }
    }

    if (showRole == true && user.getRole() != null)
    {
        buffer.append("(");
        buffer.append(user.getRole().getDescription());
        buffer.append(")");
    }
    return buffer.toString();
}

// Properties

```

```

public StyleType getStyle()
{
    return style;
}

public void setStyle(StyleType style)
{
    this.style = style;
}

public boolean isShowRole()
{
    return showRole;
}

public void setShowRole(boolean showRole)
{
    this.showRole = showRole;
}

// StateHolder

public boolean isTransient()
{
    return transientFlag;
}

public void setTransient(boolean transientFlag)
{
    this.transientFlag = transientFlag;
}

public Object saveState(FacesContext context)
{
    Object[] values = new Object[2];
    values[0] =
        new Integer(style.getValue());
    values[1] =
        (showRole == true ?
            Boolean.TRUE : Boolean.FALSE);
    return values;
}

public void restoreState(
        FacesContext context,
        Object state)
{
    Object[] values = (Object[])state;
    int styleValue =
        ((Integer)values[0]).intValue();
    style =

```

⑩ **Save and
restore
properties**

```
(StyleType)StyleType.getEnumManager().  
    getInstance(styleValue);  
showRole = ((Boolean)values[1]).booleanValue();  
}  
  
// Nested top-level class  
  
public static class StyleType extends  
    EnumeratedType  
{  
    public final static StyleType FIRSTNAME =  
        new StyleType(0, "firstname");  
    public final static StyleType LASTNAME =  
        new StyleType(10, "lastname");  
    public final static StyleType FIRSTNAME_LASTNAME =  
        new StyleType(20, "firstname_lastname");  
    public final static StyleType LASTNAME_FIRSTNAME =  
        new StyleType(30, "lastname_firstname");  
  
    private static EnumManager enumManager;  
  
    static  
{  
        enumManager = new EnumManager();  
        enumManager.addInstance(FIRSTNAME);  
        enumManager.addInstance(LASTNAME);  
        enumManager.addInstance(FIRSTNAME_LASTNAME);  
        enumManager.addInstance(LASTNAME_FIRSTNAME);  
    }  
  
    public static EnumManager getEnumManager()  
{  
        return enumManager;  
    }  
  
    private UserDisplayType(int value, String description)  
{  
        super(value, description);  
    }  
}
```

10

Save and
restore
properties11 Use for style
property

- ① All converters must implement the `Converter` interface, but we also implement the `StateHolder` interface so that JSF can maintain the value of the converter's properties between requests.
- ② Like all of the other user interface extensions, converters have identifiers, which are configured in a JSF configuration file. Here, we declare the default identifier for this converter.

- ❸ We set the `transientFlag` (used by the `StateHolder` interface's `transient` property) to `false`, to ensure that the state will be saved.
- ❹ All converters must implement the `getAsObject` method. In this case, we translate the user's input into a new `User` instance.
- ❺ If we receive a `FacesMessage` instance from the `getAsUser` method (described in ❻), it means that there's a conversion error. We can report this error simply by throwing a `ConversionException` with this `FacesMessage` instance. These messages will automatically be associated with the proper control, and can be displayed back to the user by using a `HtmlMessage` or `HtmlMessages` component. If no `FacesMessage` was returned, then the conversion went successfully, and we can return a new `User` object.
- ❻ All converters must implement `getAsString`. In this case, it converts a `User` object into a `String` based on the properties of the converter. The actual work is performed by the `getUserAsString` method (❾). Note that we don't throw any exceptions because there's no chance for error—we convert as little or as much information as the `User` object has. There's also no need to worry about the `User` object being `null`, because JSF won't even call the converter in that case.
- ❼ The `getStringAsUser` method performs the core work of converting a `String` into a `User` object. This method is implemented without any dependencies on the JSF environment so that it can be easily unit-tested. For each possible `style` value, it delegates to another utility method. It also delegates processing to another method if the `showRole` property is `true`. All of these methods work similarly (see ❻).

TIP

If possible, you should put the core converter logic in separate methods that don't use any runtime JSF classes. (`FacesMessage` is okay, because it's a simple JavaBean with no runtime dependencies.) This simplifies unit testing because there's no requirement for a web container or mock objects. The sample code for this book includes JUnit test cases that take advantage of this fact.

- ❻ This class has one utility method for each display style. All of these methods have the same basic algorithm. For the given `style` type, split up the input string and check to make sure it has the proper number of elements. If not, create a new `FacesMessage` and return it. If so, update the properties of the newly created `User` object based on the different pieces of the string.
- ❼ The `getUserAsString` method is responsible for the real work of translating a `User` object into a `String`. Like `getStringAsUser`, it's not dependent on JSF runtime classes so that it can be easily unit-tested. Unlike `getStringAsUser`, it performs all of the necessary work rather than delegating to other methods. All it does is build

a new String based on the current style and showRole properties. The format of the String will always match the format that `getUserAsString` expects.

- ⑩ We store and retrieve all properties with the `saveState` and `restoreState` methods of the `StateHolder` interface. Because `style` is a `StyleType` (⑪) instance that has a fixed number possible of static values, there's no reason to serialize the whole object. Instead, we just save its value, which is used to retrieve the associated object during the restoration process.
- ⑪ The `style` property is represented using an instance of `StyleType`, which is a nested top-level class. (It can be instantiated outside our `UserConverter` class, using the notation `UserConverter.StyleType`.) `StyleType` is a type-safe enumeration that represents all of the possible values for the `style` property.

That's it for the `UserConverter` class. The next step is to register it with JSF.

20.4.2 Registering the converter

Registering a converter requires, at a minimum, a `<converter>` entry that has `<converter-id>` and `<converter-class>`. This entry must be added to a JSF configuration file that can be found by your application. Here's the entry for `UserConverter`:

```
<converter>
    <description>
        Converts a User object to and from a String.
    </description>
    <converter-id>jia.User</converter-id>
    <converter-class>
        org.jia.converters.UserConverter
    </converter-class>
</converter>
```

This registers the class `org.jia.converters.UserConverter` under the identifier `jia.User`, and provides a simple description. Note that the identifier has the same value as the constant `UserConverter.CONVERTER_ID`.

`UserConverter` is registered by identifier, but recall that you can also register a converter by the type of class it is converting. So, `UserConverter` would be registered like so:

```
<converter>
    <description>
        Converts a User object to and from a String.
    </description>
    <converter-for-class>
        org.jia.ptrack.domain.User
    </converter-for-class>
</converter>
```

```
org.jia.converters.UserConverter
</converter-class>
</converter>
```

When a converter is registered by type, JSF automatically uses it anytime it encounters that type (as long as no other converter is registered for the component). This means that the developer has no control over when the converter is applied, and cannot set any properties.

We'd like to give developers control over when and how `UserConverter` is used—this is why it is registered by identifier as opposed to type. Remember, however, that it's possible for the same converter to be registered both ways.

Now, let's examine the JSP side of the coin.

20.4.3 JSP integration

Integrating a converter with JSP is not always necessary. Converters that are registered by type don't require JSP tags because they're not registered explicitly. Converters that are registered by identifier can always be registered with the `converter` attribute of many component tags:

```
<h:outputText value="#{myUser}" converter="jia.User"/>
```

This applies our new `UserConverter` to a `HtmlOutputText` component without creating any additional tags. This works fine for converters that have no properties, as is often the case. (The converter we developed for ProjectTrack in chapter 12 was used this way.) You can also use the `<f:converter>` tag to achieve the same effect. However, if your converter has properties, or you want to provide a more concrete interface for front-end developers, you will need to develop a custom JSP tag handler.

This process is as simple as it is with validators: develop a tag handler class and then register it with a tag library. Because there are no renderers, there's no need to handle extra attributes—only the properties of the corresponding converter must be supported.

Writing the `UserConverterTag` class

`org.jia.converters.taglib.UserConvertTag` subclasses `ConverterTag`, which is designed almost exactly like `ValidatorTag`. Our new tag handler will support the same two properties as `UserConverter` (see table 20.2); all we need to do is pass through the tag handler's property values to those of the newly created `UserConverter` instance. To make life easier for the tag's user, we'll support value-binding expressions as well. The source is shown in listing 20.5.

Listing 20.5 UserConverterTag.java: JSP tag handler for UserConverter

```

package org.jia.converters.taglib;

import org.jia.converters.UserConverter;
import org.jia.util.Util;

import javax.faces.application.Application;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.webapp.ConverterTag;

public class UserConverterTag extends ConverterTag      ↗① Subclass
{                                                 ConverterTag

    private String style;
    private String showRole;

    public UserConverterTag()
    {
        super();
        setConverterId(          ↗② Set the
            UserConverter.CONVERTER_ID);   converterId in
    }                                         constructor

    // ConverterTag methods

    protected Converter createConverter()      ↗③ Override the
                                                createConverter
                                                method
    throws JspException
    {
        UserConverter converter =
            (UserConverter)super.createConverter();   ↗④ Get converter
                                                       instance from
                                                       superclass

        FacesContext context = FacesContext.getCurrentInstance();
        Application app = context.getApplication();

        if (style != null)
        {
            if (Util.isBindingExpression(style))
            {
                converter.setStyle(
                    (UserConverter.StyleType)app.
                    createValueBinding(style).
                    getValue(context));
            }
            else
            {
                UserConverter.StyleType styleType =
                    (UserConverter.StyleType)
                    UserConverter.StyleType.
                    getEnumManager().getInstance(

```

① Subclass ConverterTag

② Set the converterId in constructor

③ Override the createConverter method

④ Get converter instance from superclass

⑤ Set converter properties based on tag handler properties

```

        style.toLowerCase());
        converter.setStyle(styleType);
    }
}

if (showRole != null)
{
    if (Util.isBindingExpression(
        showRole))
    {
        converter.setShowRole((Boolean)
            app.createValueBinding(showRole).
                getValue(context)).booleanValue());
    }
    else
    {
        converter.setShowRole(
            Boolean.valueOf(showRole).
                booleanValue());
    }
}

return converter;
}

public void release()
{
    super.release();
    style = null;
    showRole = null;
}

// Properties

public String getStyle()
{
    return style;
}

public void setStyle(String style)
{
    this.style = style;
}

public String getShowRole()
{
    return showRole;
}

public void setShowRole(String showRole)
{
}

```

5 Set converter properties based on tag handler properties

6 Override the release method

```
        this.showRole = showRole;
    }
}
```

- ➊ All converter tag handlers should subclass ConverterTag.
- ➋ In the constructor, we call `setConverterId` to associate this tag with a particular converter. Note that we use `UserConverter`'s constant, which matches the value under which we registered the class in the JSF configuration file.
- ➌ All of the work happens in the `createConverter` method, so we must override it.
- ➍ Before we begin processing, we retrieve a `Converter` instance from the superclass's implementation of `createConverter`, which creates a new instance based on the `converterId` property, which we set in ➋.
- ➎ Here, we set the properties of our new `UserConverter` instance based on the properties of the tag handler itself. Note that for the `style` property, the tag handler does the work of retrieving a `UserConverter.StyleType` instance. In essence, it's translating between its own property (which is a `String`) and the converter's property (which is a `UserConverter.StyleType` instance).

For both properties, we support value-binding expressions. (The `Util.isBinding` method just checks to see if a string starts with “#{” and ends with “}”.) Like validators, converters don't keep a list of `ValueBinding` instances to be evaluated at a later date, because their properties are generally onetime configuration values. So, instead of adding a new `ValueBinding` instance, as we would for a `UIComponent`, we simply evaluate the expression and set the property to equal its result.

- ➏ Finally, we override the `release` method to reset all of our values. (As always, we call the superclass's version of the method first.)

Fortunately, implementing `UserConverterTag` was a fairly trivial exercise. Now, let's add it to the tag library.

Adding the tags to the tag library

The tag handler entry for `UserConverterTag` is pretty simple because there are only two attributes. It's shown in listing 20.6.

Listing 20.6 Tag library entry for UserConverterTag

```
<?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>
  <!-- Tag Library Description Elements -->
```

```

<tlib-version>1.0</tlib-version>
<jsp-version>1.2</jsp-version>
<short-name>JSF in Action Custom Tags</short-name>
<uri>jsf-in-action-components</uri>
<description>
    Sample custom components, renderers, validatos, and
    converters from JSF in Action.
</description>
<!-- Tag declarations -->
...
<tag>
    <name>userConverter</name>
    <tag-class>
        org.jia.converters.taglib.UserConverterTag
    </tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>style</name>
        <required>true</required>
        <rteprvalue>false</rteprvalue>
    </attribute>
    <attribute>
        <name>showRole</name>
        <required>false</required>
        <rteprvalue>false</rteprvalue>
    </attribute>
</tag>
...
</taglib>

```

The code in listing 20.6 registers a tag called `<userConverter>` for the class `org.jia.converters.taglib.UserConverterTag` with `style` and `showRole` attributes. The `style` attribute is required.

This completes the fairly mundane task of integrating `UserConverter` with JSP. Let's move on to actually using the converter.

20.4.4 Using the converter

`UserConverter` can be used to either display a `User` object with an output component like `HtmlOutputText`, or to create a `User` object from an input component like `HtmlInputText`. Let's examine the former scenario first.

Let's say that we have a `User` object stored in the session under the key `user`. We can use our converter to display this object with an `HtmlOutputText` component like so:

```

<h:outputText value="#{user}">
    <jia:userConverter style="lastName"/>
</h:outputText>

```

This displays only the user's last name. This happens to be equivalent to the following tag:

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

Here, we reference the `lastName` property of the `User` object directly. So far, our converter isn't offering us a lot of value.

Things get more interesting when you use `UserConverter` with a `HtmlInputText` control:

```
<h:inputText id="firstNameLastNameInput" value="#{user}" size="30">
    <jia:userConverter style="firstName_LastName" />
</h:inputText>
```

Now, if anyone enters a two-word name into this `HtmlInputText`, a new `User` object will be created and stored in the session under the key `user`. So, if the input was "Eduardo LaCalle", the `User` object's `firstName` property would be "Eduardo" and the `lastName` property would be "LaCalle". If the value isn't two words, `UserConverter` will create an error message, which can be displayed with a `HtmlMessage` component:

```
<h:message for="firstNameLastNameInput" styleClass="error" />
```

This tag displays all the messages for the `HtmlInputText` above, with a CSS style of "error".

Displaying a `User` with this converter makes more sense if you use `style` and set the `showRole` property to `true`:

```
<h:outputText value="#{user}">
    <jia:userConverter style="lastName_FirstName" showRole="true" />
</h:outputText >
```

So, if Eduardo LaCalle's role was `Business Analyst`, this would display "LaCalle, Eduardo (Business Analyst)". Displaying such output without the converter is a little more involved. Moreover, using the converter results in consistent formatting—it's a canonical way to display the `User`.

That's all there is to using `UserConverter`. A more sophisticated version of this class might support displaying the `login` property and check to make sure that the words contained all letters. While this particular converter is more suited for displaying an object in a specific, repeatable way, others can be quite useful for input as well.

20.5 Summary

In this chapter, we examined specific examples of building a custom validator and converter. We began with a custom `RegularExpressionValidator`, which allows the front-end developer to specify a regular expression that can match against user input. Because regular expressions can describe a variety of possible string formats, we were able to validate a phone number, a social security number, and an email address all with the same validator. Developing the validator required a single validator class that had to be registered with JSF, and a tag handler class that needed to be registered with a JSP tag library.

Our converter example, `UserConverter`, was responsible for converting a `User` object to and from a `String`. The converter had properties for converting a `User` in different ways, like using the last name, first name, and role. Using this converter provides a specific way of displaying a `User` object that eliminates the need to access specific `User` properties. Developing the converter required a single converter class and a tag handler class. The converter class had to be registered with JSF through a configuration file, and the tag handler had to be registered with a tag library.

A survey of JSF IDEs and implementations

One of the key goals of JavaServer Faces is to support integrated development environments (IDEs). JSF is a standard, so many developers will interact with it through IDEs. Throughout this book, we have shown how these tools expose different parts of the standard. In this appendix, we take a closer look at Oracle's JDeveloper [Oracle, JDeveloper], IBM's WebSphere Studio [IBM, WSAD], and Sun's Java Studio Creator [Sun, Creator]. Using each of one of these tools, we'll build ProjectTrack's Login page step by step. This should give you a feel for each tool's feature set, and an understanding of how tools can differ from vanilla Faces development.

NOTE The code used in these examples is not *exactly* the same as the code used earlier in this book. Specifically, we've left out the servlet context listener, and some other design choices, such as a base backing bean class, have been avoided for the sake of simplicity. Our goal here is to give you a flavor of how the tools work with a familiar example, as opposed to building the application in exactly the same manner as we have earlier in this book.

In addition, the UI component names used within IDEs and in this appendix aren't necessarily the same as the names we've been using throughout this book, like `HtmlPanelGrid`. In chapter 4 (table 4.1), we list common IDE names for all of the standard components.

B.1 Using JSF with Oracle JDeveloper

Contributed by Jonas Jacobi, Oracle Application Server Technologies

In the spring of 2004, Oracle introduced a new version of its Java development tool: Oracle JDeveloper 10g [Oracle, JDeveloper]. JDeveloper provides an extensive range of tools for developing, debugging, testing, tuning, deploying, and versioning J2EE applications and web services. Furthermore, JDeveloper is a complete SQL and PL/SQL development environment, enabling developers to work on all tiers of their applications.

With JDeveloper, you can build J2EE applications and web services either from scratch or by using a J2EE framework such as the Oracle Application Development Framework (ADF) [Oracle, ADF]. Whichever implementation you choose, JDeveloper offers powerful tools needed to get the job done, including UML modelers, visual editors, wizards, dialogs, and code editors.

Oracle ADF is a flexible and extensible framework, based on industry standards. It offers pluggable technologies for the model, view, and controller, allowing developers to make implementation choices at various layers of the architecture.

You can already use JSF with the current production version of Oracle JDeveloper 10g,¹ but Oracle will fully support JSF in the next major release of JDeveloper. This version will include a sophisticated and rich set of user interface components, called ADF Faces Components, in addition to fully supporting development with the standard JSF components.

B.1.1 Oracle's view on JSF

Although JavaServer Faces is a young technology, Oracle is putting a lot of effort behind this new component-based view framework. Oracle believes that JSF as a component-based technology will provide developers with a productive way of building J2EE applications—close to what established tools like Oracle Forms or Visual Basic can provide. Oracle also believes that with a component-based architecture, developers can provide end users with a richer user experience. For example, Oracle's ADF Faces Components support partial page rendering (PPR), which makes applications appear more responsive.

As a technology, JSF is not a new thing. Several component frameworks are already available, such as Tapestry [ASF, Tapestry] and Oracle's ADF UIX [Oracle, ADF UIX]. As one of the leading members of the JSF expert group, Oracle has devoted time and resources toward developing JSF, and consequently, JSF is architecturally similar to ADF UIX.

B.1.2 What are ADF Faces Components?

The ADF Faces Components are already available in the current production release of Oracle JDeveloper 10g in the form of ADF UIX components. ADF UIX is a user interface framework for building J2EE-compliant web applications that are component based and XML metadata driven; it is one of the view technologies available in Oracle ADF. ADF Faces Components are a new version of these components based on JSF.

As a technology, ADF UIX has been extensively used within Oracle for several years to produce products like Oracle iLearning, Oracle Enterprise Manager, and

¹ JDeveloper 10g has full support for working with JSP tag libraries visually, and it's quite easy to integrate a JSF implementation into your project. See Chris Schalk's article [Schalk] for details on using JDeveloper 10g with JSF.

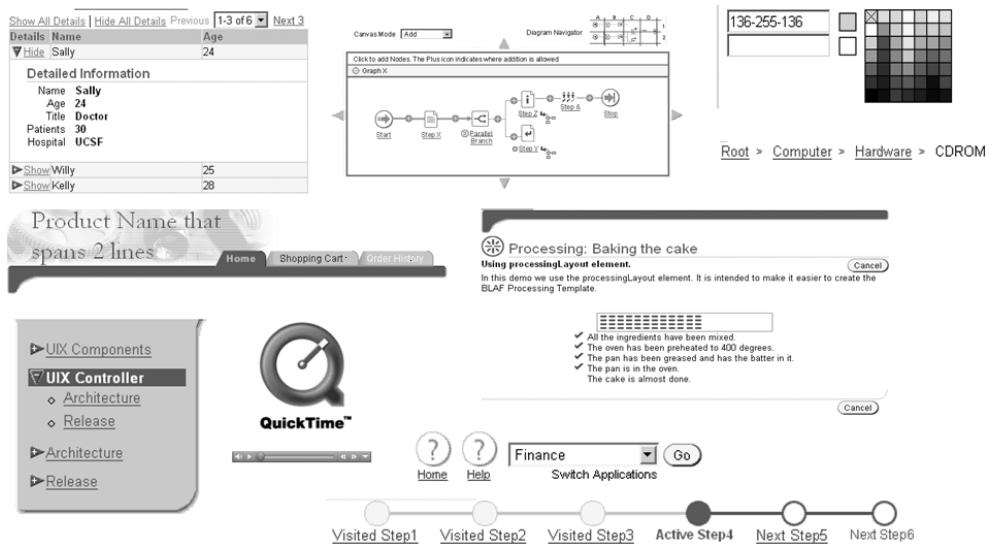


Figure B.1 A set of ADF Faces Components.

the Oracle's eBusiness Suite. In figure B.1, you can see a set of the currently available ADF UIX Components that are all being migrated over to the new JSF standard.

Although the initial release of JavaServer Faces provides developers with a rich architecture, there are limitations in the functionality and only a small number of standard UI components are available. In order for developers to create an enterprise application, they may need additional libraries of reusable user interface components and render kits. Oracle ADF Faces Components have a wide range of benefits that can be broken into two main categories: design-time and runtime.

Design-time benefits of Oracle ADF Components and JDeveloper include the following:

- *A WYSIWYG development environment*—With the next release of Oracle JDeveloper 10g, developers will be able to build JSF applications with a WYSIWYG approach within the Visual Editor and Page Flow Editor. For third-party component providers, no extra layers or metadata need to be applied to a custom component in order to have a nice rendering experience in the Visual Editor. Oracle JDeveloper's Visual Editor will use the same renderers as are used during runtime.
- *A rich set of UI components*—ADF Faces provide the developer with a rich set of UI components such as Table, hGrid, Color Picker, and Calendar, that can be customized and reused.

- *A declarative development environment*—Because ADF Faces is based on XML metadata, the JDeveloper IDE provides a declarative environment where the developer can, for example, use a Property Inspector to modify the UI components instead of writing code.
- *ADF Face features*—You can choose to use all, some, or no ADF Faces features, depending on your application development needs.
- *Full support*—Full support for the JSF EL.
- *Drag-and-drop*—Drag-and-drop access to different business services, which allows you to apply them without knowing the details of the underlying technology.
- *Customization*—ADF Faces applications are easily customized for different end-user environments. Different layouts and styles can also be implemented.
- *A consistent look and feel*—Using ADF Faces to create user interfaces ensures a consistent look and feel, allowing you to focus more on user interface interaction than look-and-feel compliance.
- *Data binding*—Access to the ADF Data Binding framework allows developers to drag and drop data-bound JSF components directly onto the page without worrying about the underlying business services. The ADF Data Binding framework is a standard, declarative way to bind data from a business service, such as web services, Enterprise JavaBeans (EJB), Java, J2EE Connector Architecture (JCA), and JDBC, to other entities, such as UI components. For more information, see JSR 227, “A Standard Data Binding and Data Access Facility for J2EE” [JSR 227].

ADF Faces’ runtime benefits include:

- *Standards*—ADF Faces is based solely on JSF. No other Oracle products are required at runtime to use ADF Faces, so you can use them with a standard web container (like Tomcat) and any JSF implementation.
- *High interactivity*—The ADF Faces Components provide a high level of interactivity during runtime. ADF Faces Components’ support for PPR allows the page to just render a piece of the page instead of the entire page, which is the default in most cases. Perhaps, for example, you would like scroll through records in a table; this would normally require the entire page to be refreshed, but with PPR only the table will be refreshed.
- *Internationalization and accessibility support*—ADF Faces UI components have built-in support for internationalization and accessibility.

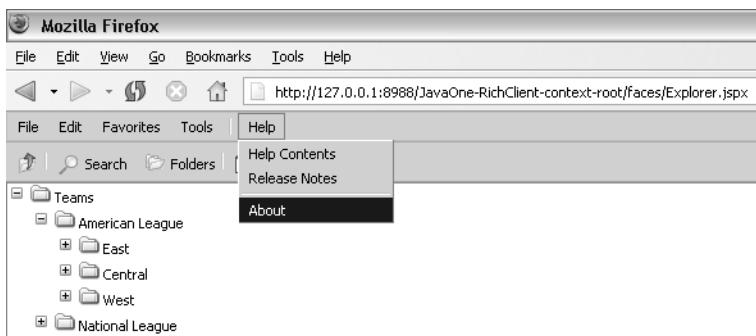


Figure B.2 Next-generation ADF Faces Menu and Tree components.

While the first-generation ADF Faces Components are still in transition from their former shell (ADF UIX), Oracle is already working on its next generation of Faces components. These new components will extend the current web interface into a richer interface that will have the same functionality as a traditional thick client (such as a Swing application). Examples of such components are menus, trees, and splitters. In figure B.2 you can see an early release of Menu and Tree components. The difference between these components and a menu implemented in JSP is that these components will behave like a desktop component, with live updates in the browser.

B.1.3 Exploring JDeveloper

Before we start building ProjectTrack's Login page, let's explore JDeveloper's workspace. If you look at figure B.3, you can see that there are five windows, each with specific functionality.

In the far upper left-hand corner, you can see the Application Navigator, which contains a tree view of work area and files. Underneath it is the Structure window, which displays the structure of the selected object, which may be a JSP page or an EJB. The center of the screen is the actual content of the page or navigation rule you are currently editing, with tabs for the design and source views.

In the upper right-hand corner, you can see the Component Palette, which contains UI components and tags that you can drag and drop onto your page or navigation rule. Underneath the palette is the Property Inspector, which displays any relevant information available for the current selected object (such as a UI component).

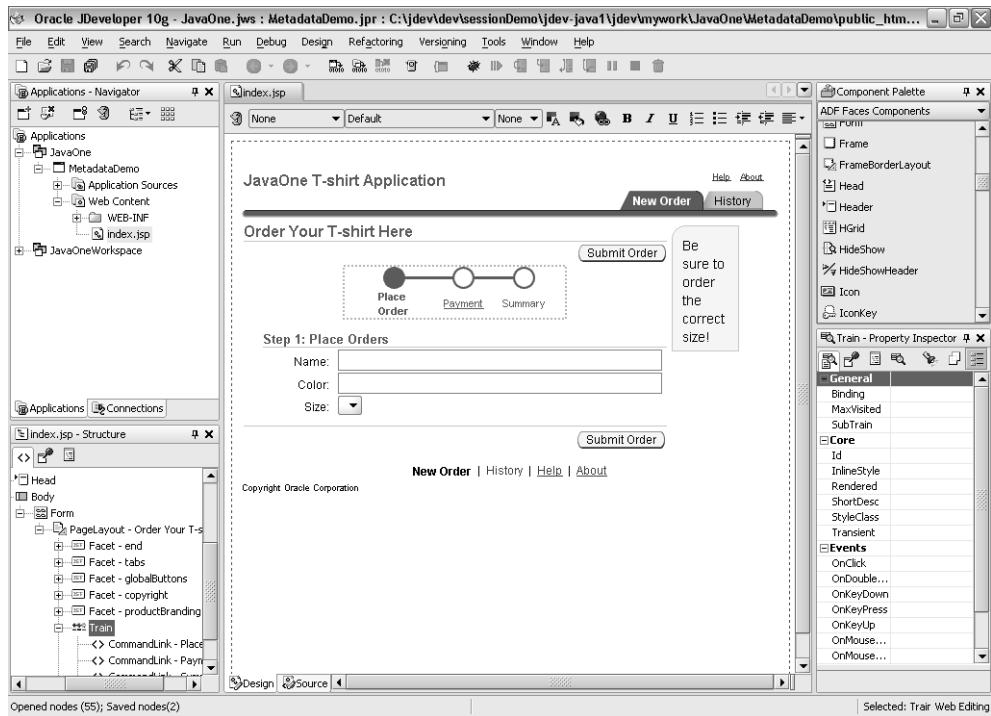


Figure B.3 Next generation of Oracle JDeveloper with a set of ADF Faces Components.

B.1.4 Building ProjectTrack's Login page

In this section, we will look at how you would build the ProjectTrack Login page using Oracle's JDeveloper. The release used in this appendix is an early build of the next generation of JDeveloper, so some of the features covered in this appendix may change before the actual production release.

Creating a new project

Because we already have a project containing all the logic we need, it makes sense to reuse this for our Login page. To import a WAR file, we need to first create a workspace. This can be done by right-clicking on the Application node in the Application Navigator. You can create a new workspace through the Create Application Workspace dialog box, shown in figure B.4.

In this dialog box you can enter the name of the application and select the application template. In this sample we will be importing a project created in another tool, so just keep the default selection and click OK.

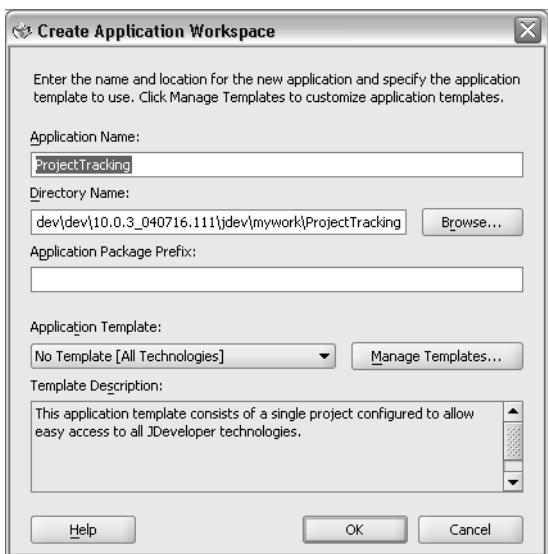


Figure B.4
The Create Application Workspace dialog box.

Right-click on the newly created workspace and select New Project. You should now see the New Gallery dialog box, shown in figure B.5. Select Project from WAR file and click OK.

Step through the wizard that launches and enter a proper name for the project, and then select the WAR file to import to the workspace. Your workspace should look something like figure B.6.

Creating a new page

The page that we are going to build using JDeveloper is already available in the project, so let's see what the page should look like when we have finished. By double-clicking on the page in the Application Navigator, you can open the file in the Visual Editor, as shown in figure B.7.

As you can see, the page renders, and by clicking on the different components in either the Structure window or the Visual Editor, you can see its properties in the Property Inspector. Now, let's build this page in JDeveloper.

First we need to create a new JSF page. Do that by right-clicking on the project node and selecting New, which displays the New Gallery dialog box shown in figure B.8.

In the dialog box, expand the WebTier node and select JavaServer Pages (JSP). In the right pane select the Faces JSP node to create a Faces page and click OK. JDeveloper displays the Create Faces JSP dialog box, shown in figure B.9. Enter a name for the page and select the JSP type that you would like use.

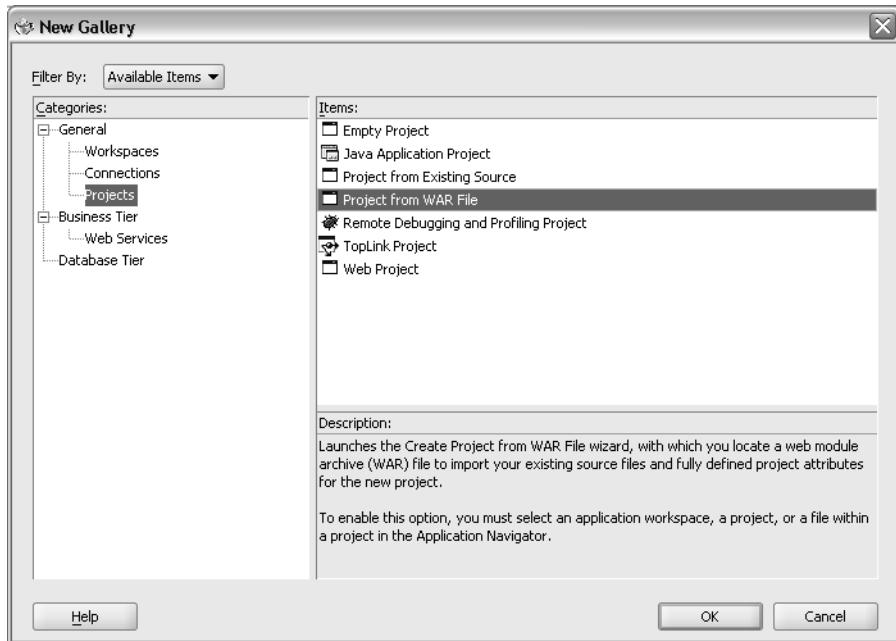


Figure B.5 Creating a new workspace with the New Gallery dialog box.

Selecting a Type of Document in this dialog box will create a proper XML-based document, which allows better support for XML tools. However, for this sample we will be using the regular Page type, so click Page. Now click Finish. You should now see an empty page in the IDE (figure B.10).

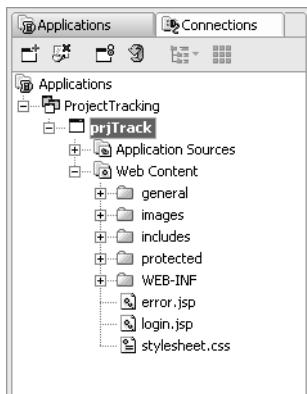


Figure B.6
Application Navigator with
imported ProjectTrack files.

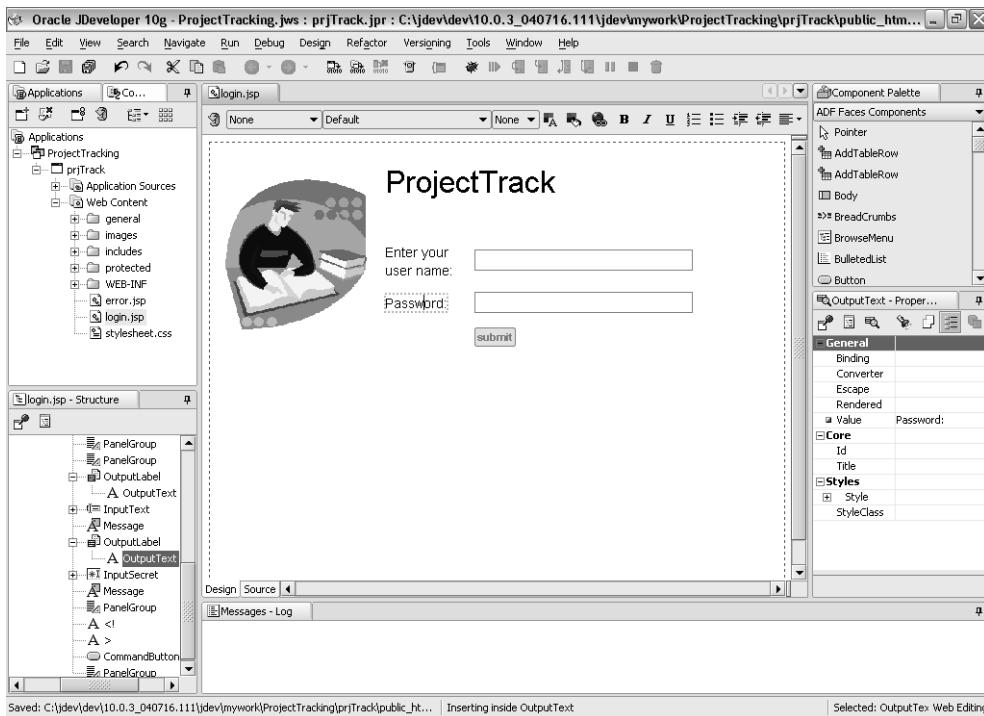


Figure B.7 JDeveloper with the original Login page displayed.

This is an empty page containing only the bare minimum that a developer needs in order to start building a JSF application. The red dotted border indicates the area covered by the `HtmlForm` component by default. In this area we will add the username and password input fields, as well as the Submit button.

Designing the page

By using the Component Palette (figure B.11), you can drag and drop components onto the page as needed. For this page we are going to use the following JSF HTML components: `PanelGrid`, `InputText`, `InputSecret`, `OutputLabel`, `Image`, `Message`, and `CommandButton`.

In order for us to lay out our components as shown in figure B.7, we can use a `PanelGrid` component. Drag a `PanelGrid` component onto the page from the Component Palette. When you have the `PanelGrid` in your page, you can set its properties through the Property Inspector. Set the `Cellpadding` property to 3, `Cellspacing` to 3, and `Columns` to 2, as shown in figure B.12.

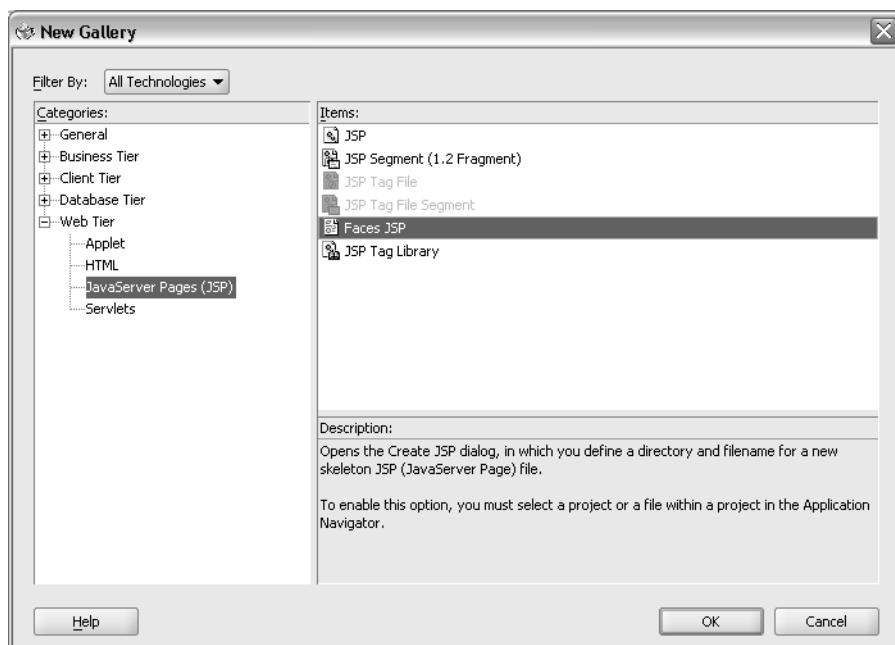


Figure B.8 Creating a new page with the New Gallery dialog box.

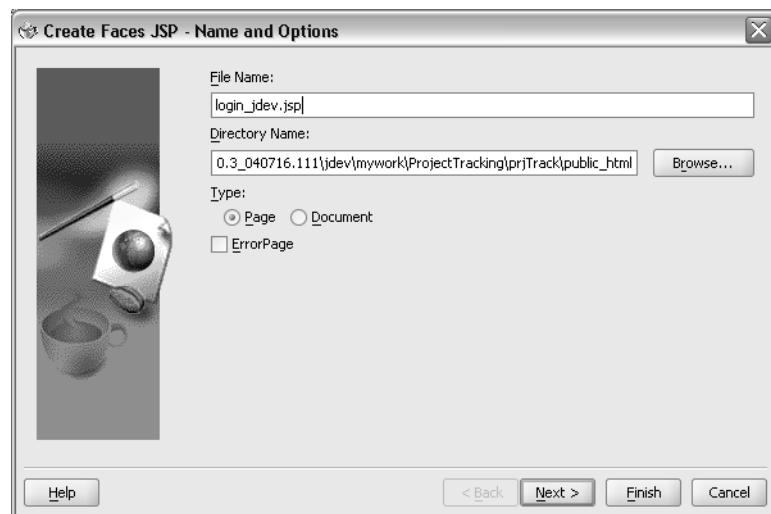


Figure B.9 The Create Faces JSP dialog box.

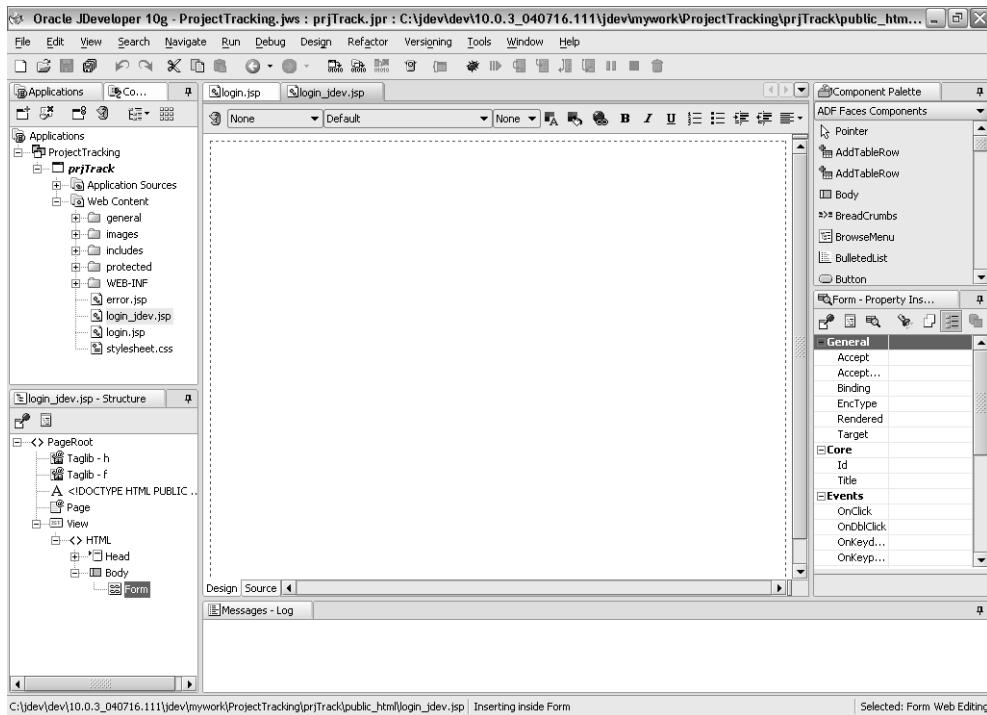


Figure B.10 New empty Faces page.

Next, we must place an image into the left column of the `PanelGrid`. You can drag the image from your desktop, from the Application Navigator, or directly from a browser. The image you want is `logo.gif` (which has already been imported); figure B.13 shows it selected in the Application Navigator.

Drag the image to the page and release the mouse button. This will create a `GraphicImage` component with a reference to the image you selected. Your page should look something like figure B.14.

Before you continue, you should change the size of the image to be less prominent. You can do so either by using the Property Inspector or by using the drag points on the image to visually change the size of the image.

In JDeveloper, you can use the Structure window to customize your page, as well as view its structure. To illustrate this, right-click on the `PanelGrid` component in the Structure window. Select `Insert Inside PanelGrid` and then choose `JSF HTML`, as shown in figure B.15.

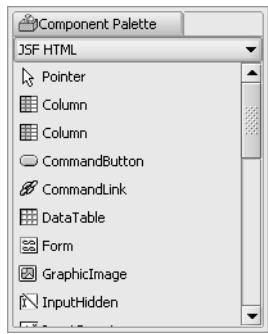


Figure B.11 The Component Palette.

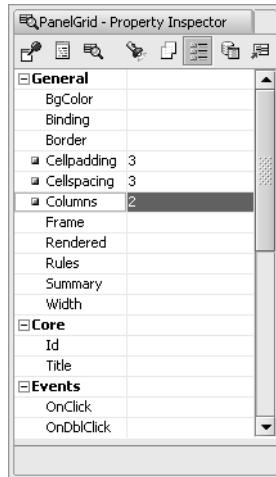


Figure B.12 The Property Inspector for PanelGrid.

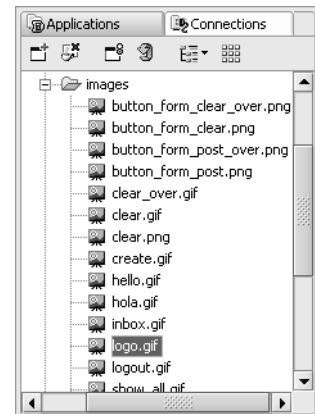


Figure B.13 Selecting the logo.gif image in the Application Navigator.

Set the following properties on the new PanelGrid: Cellpadding to 5, Cellspacing to 3, and Columns to 2, as shown in figure B.16.

Continue by adding the InputText and InputSecret components to the newly added PanelGrid. Set the following properties on the InputText component: MaxLength to 30, Required to true, Size to 20, and Id to “userNameInput”. Figure B.17



Figure B.14 Login page with image. The PanelGrid is represented by the outer dotted lines.

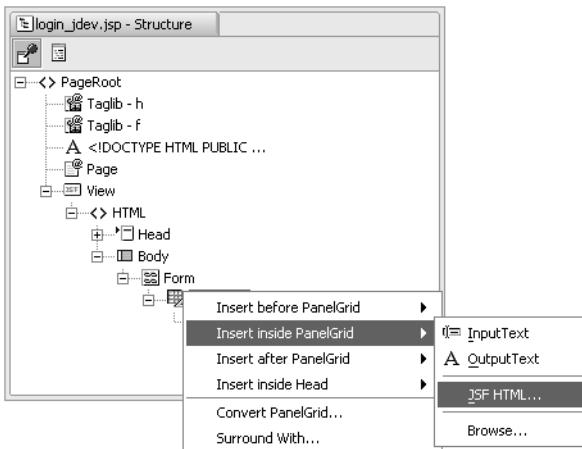


Figure B.15 Inserting another `PanelGrid` in the Structure window.

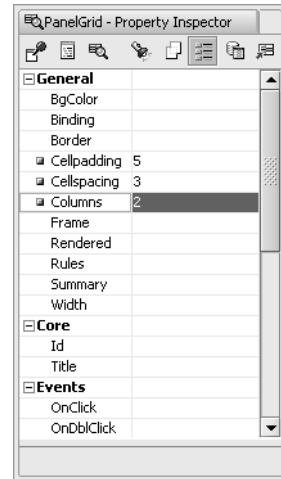


Figure B.16 Property Inspector for the inner `PanelGrid`.

shows these properties in the Property Inspector. For the `InputSecret` component, set the following properties: `MaxLength` to 20, `Required` to true, `Size` to 20, and `Id` to “`passwordInput`”.

We also need to add some labels to our text fields. From the Component Palette, drag and drop two `OutputLabel` components to the page. Set the `for` property on the two components to “`userNameInput`” and “`passwordInput`” to associate them with their respective text fields. In each of these `OutputLabel` components we need to add an `OutputText` component in order to render a string that will be

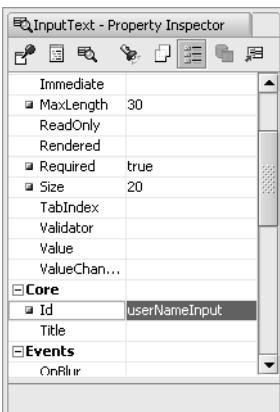


Figure B.17 Property Inspector for the `InputText` component.

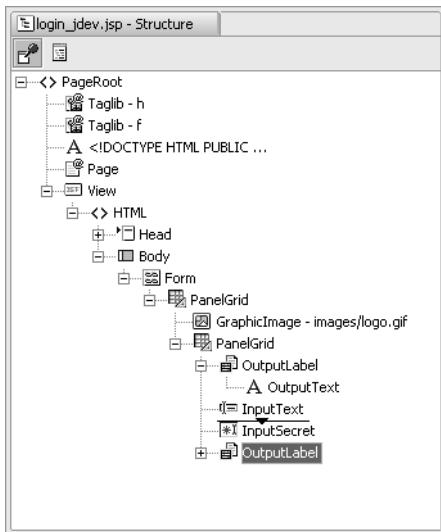


Figure B.18
Structure window for the `OutputLabel` and `OutputText` components.

our text field's prompt. Drag and drop an `OutputText` component to each `OutputLabel` component and set the `value` property to "Enter your user name:" and "Password:", respectively. The location of the newly added `OutputLabel` components are not what we wanted, so in order to adjust this we can drag and drop them to the right location either in the Visual Editor or in the Structure window (see figure B.18).

For the user to be able to submit his or her username and password, we need a `CommandButton`. Drag and drop a `CommandButton` component from the Component Palette onto the page below the Password text field. If you want the `CommandButton` to align with the text fields you can add an empty `PanelGroup` component before it. Your page should now look something like figure B.19.

For the button we also want to add an image through the Property Inspector. Make sure you have selected the button, and in the Property Inspector select the `image` property and enter the needed URI: "/images/submit.gif". Your Login page should now look like figure B.20.

Next, we will use the header facet of the `PanelGrid` component to add the title "ProjectTrack" to our page.

Drag and drop a facet from the JSF Core panel in the Component Palette to the nested `PanelGrid` component and set the name to "header". Inside this facet, add an `OutputText` component. You can edit text for components like `OutputText` directly in the Visual Editor by placing the cursor where you want to modify or

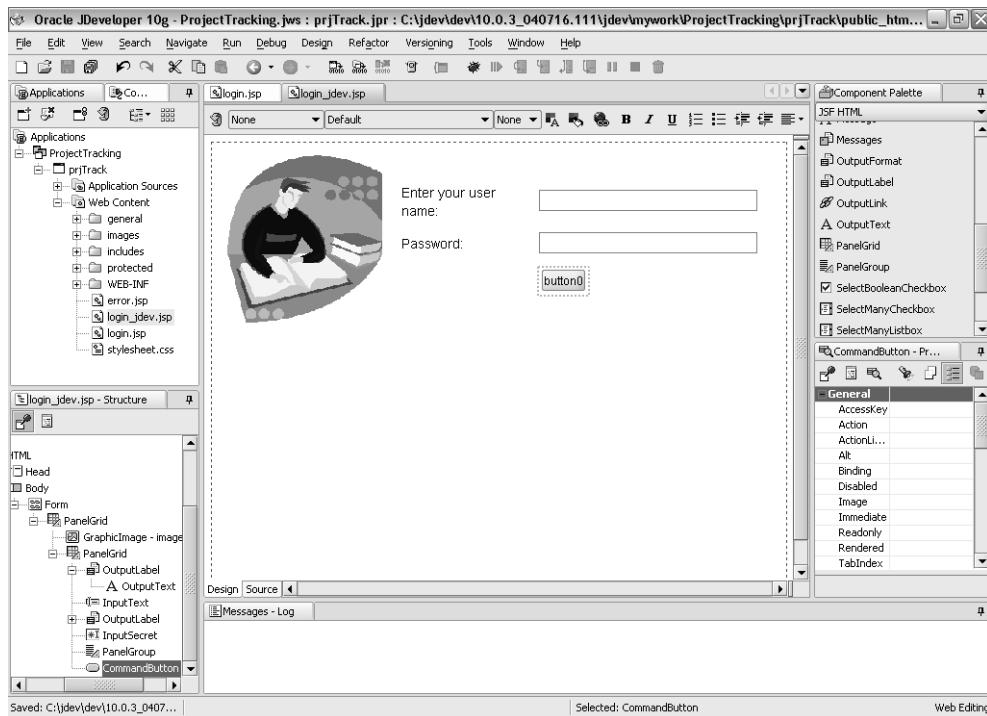


Figure B.19 The Login page, partially designed.

add text. Use this feature to change the text of the `OutputText` component in the header facet to “ProjectTrack”. The result should look like figure B.21.

Adding validation

Next, we must add validation to make sure that the password and username are in the right format. For this we will use the standard JSF validators. First, let’s add a validator to each of the text fields to ensure the length of the string entered is within our requirements.

In the JDeveloper IDE, change the panel grouping for the Component Palette to display the JSF Core components. In this section, select the `ValidateLength` validator and drag it on to the `InputText` component either in the Visual Editor or in the Structure window. Validators are nonvisual, so they can only be viewed in the Structure window, underneath the `InputText` component, as shown in figure B.22.

Set the `maximum` and `minimum` properties on the validator to 30 and 5, respectively. Add another `ValidateLength` validator to the `InputSecret` component in the same way and set the `maximum` and `minimum` properties to 20 and 5, respectively.



Figure B.20 Login page with an added image for the button.

Validation errors can be displayed by adding Message components after the `InputText` and `InputSecret` components.



Figure B.21 The completely designed Login page.

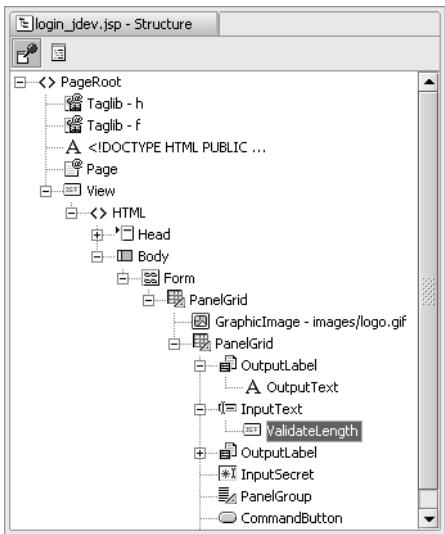


Figure B.22
The Structure window displaying the ValidateLength validator.

Editing the configuration file

JDeveloper has several options for editing your faces-config.xml file. You can launch the Faces Config Editor by double-clicking on the file in the Application Navigator. In this editor you can declaratively edit the faces-config.xml file, or edit the XML source directly by clicking on the Source tab at the bottom of the window. You can also edit the file through the Structure window or the Property Inspector, and even add elements to it through the Component Palette.

To finish this page, we need to wire it up to our authentication implementation and configure the page's navigation rules. Checking authorization for a user in this sample application is done with the AuthenticationBean. To get access to this bean in our application, we need to reference it from a managed bean in the faces-config.xml file. In the Visual Editor we will then select our components and point to the bean using the JSF EL.

Because we imported ProjectTrack's existing faces-config.xml file, the managed beans have already been configured. JDeveloper's Faces Config Editor displays them visually, as shown in figure B.23.

Binding components to backing beans

Because AuthenticationBean has already been configured, let's continue by binding this bean to our CommandButton, InputText, and InputSecret components. Return to the login_jdev.jsp page and select the Submit button. In the Property Inspector, select the Action property and click the Data binding button at the top of the

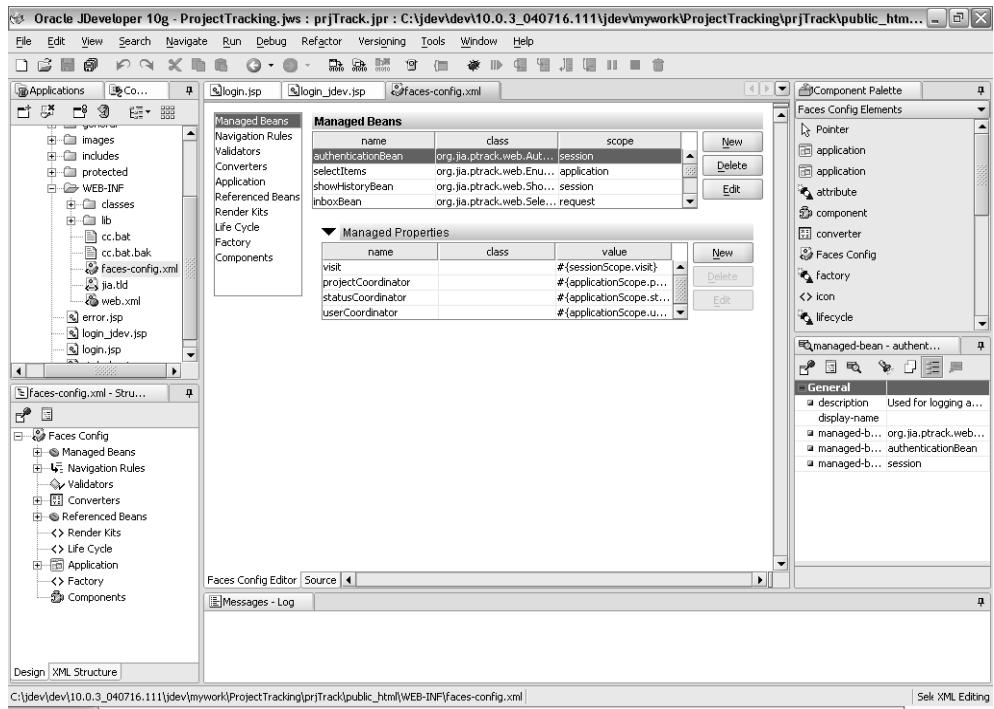


Figure B.23 The Faces Config Editor, Structure window, Property Inspector, and Component Palette.

Property Inspector. This will enable the property to use the EL Binding Editor. Optionally you could type the expression directly in the Property Inspector without clicking this button. Your Property Inspector should look like figure B.24.

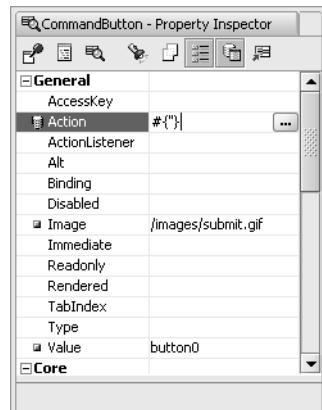


Figure B.24
Data-enabled properties.

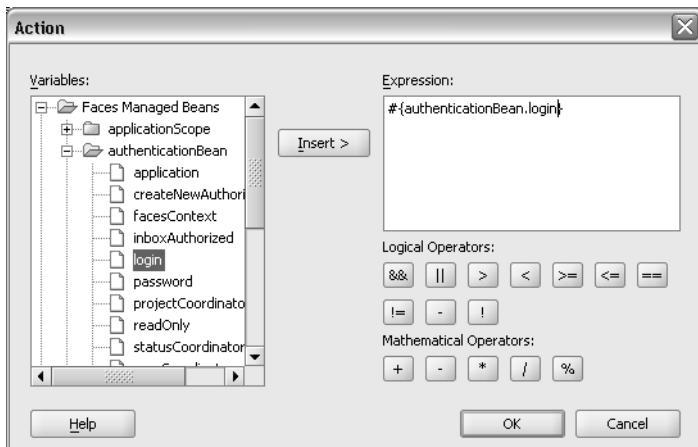


Figure B.25
EL Binding Editor.

Click on the “...” button next to the Action property in the Property Inspector to launch the EL Binding Editor. From here you can select the managed bean to bind to your action. In the EL Binding Editor, expand the Faces Managed Beans node and select the login method underneath the authenticationBean node. Click the Insert button to insert the expression to bind the CommandButton to the login method, as shown in figure B.25. Click OK to close the editor.

If you don't have an existing backing bean to bind to your page, JDeveloper can automatically create a backing bean for you. For example, double-clicking on a CommandButton generates a new backing bean and creates an action method for that button. When the backing bean is generated, the tool will also create component bindings for all UI components available in the page. When new components are added to the page, new component bindings will automatically be added to the backing bean.

Instead of using the EL Binding Editor, you can use the Property Inspector to enter the expression directly. To illustrate this, select the `InputText` component on the page and change its `value` property to “`#{authenticationBean.loginName}`” in the Property Inspector. Next, do the same for `InputSecret`, but set its `value` property to “`#{authenticationBean.password}`”.

Adding navigation

Although a full-fledged editor for navigation rules (called Page Flows in JDeveloper) is planned for the production release of Oracle's next generation of JDeveloper, it is not supported in the early build used in this book. It is simple, however, to edit navigation rules inside the Faces Config Editor by selecting the Navigation Rules

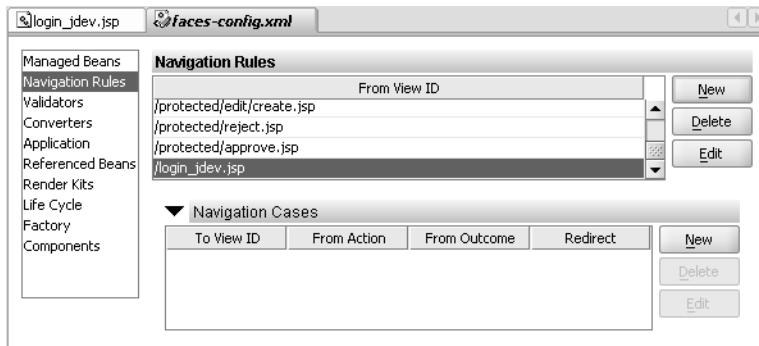


Figure B.26 Faces Config Editor—navigation rules.

node in the left pane. First we need to add the From View ID, which in this case is the login_jdev.jsp file. Click the New button at the right side of Navigation Rules pane in the editor and enter the name of the file we are navigating from—/login_jdev.jsp, as shown in figure B.26.

After we have added the login_jdev.jsp file to our navigation rules, we must add two navigation cases: one for a failure to log in and one for a successful login. Click the New button to the right of the Navigation Cases pane and set the first navigation case as follows: set the To View ID property to “/protected/inbox.jsp” and the From Outcome property to “success_readwrite”. The second navigation case should have the To View ID property set to “/login_jdev.jsp” and the From Outcome property set to “failure”, as shown in figure B.27.

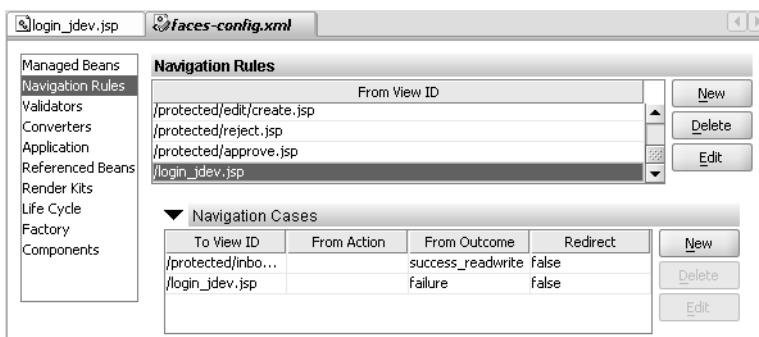


Figure B.27 Faces Config Editor—navigation cases.

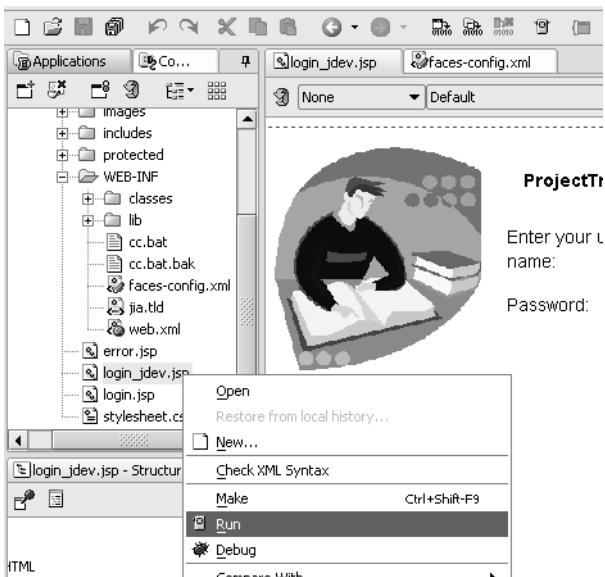


Figure B.28
Running the page.

Testing

Oracle JDeveloper provides both local and remote debugging, so developers can examine code as it is executed on remote J2EE containers. For J2EE applications, developers can use the built-in J2EE container that comes with Oracle JDeveloper to test their JSP, Servlets, and EJBs without having to install a stand-alone application server. You can also deploy directly from the IDE into a J2EE container of your choice, such as Oracle Application Server, BEA's WebLogic, JBoss, and Tomcat.

We have now finished our ProjectTrack's Login page. You can test it by right-clicking on the page in the Application Navigator and selecting Run (as shown in figure B.28), or by clicking the Run button on the toolbar.

The first time you run the page, the embedded J2EE container will start up, and subsequent runs will only deploy the project and launch a browser. After you run the page, it should look like figure B.29.

B.1.5 Wrapping up

Oracle's JDeveloper will supply developers with an IDE supporting JSF visually and declaratively. Developers will also have access to the infrastructure—ADF Faces and ADF Data Binding—needed to successfully build rich web applications as well as nearly a hundred ADF Faces Components. Third-party component providers will also enjoy the rendering facilities built into their visual editor without worrying about having to write proprietary solutions for quality rendering.

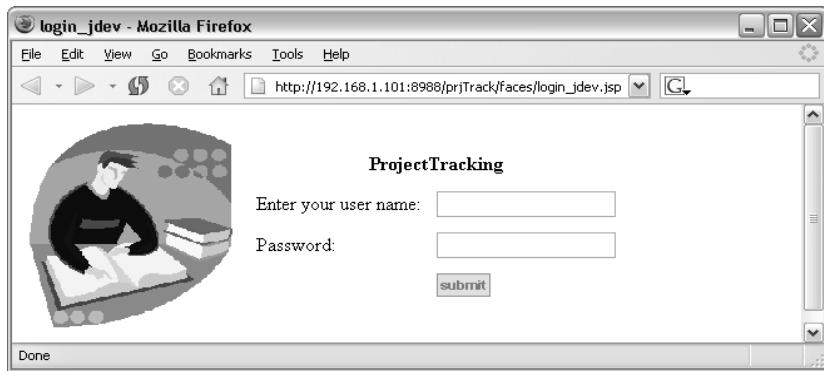


Figure B.29 Running the Login page.

support in the IDE. As one of the active members of the JavaServer Faces Expert Group, Oracle has committed itself to evolve and support the standard. This early preview of Oracle JDeveloper is evidence of the company's commitment.

B.2 Using JSF with WebSphere Studio

Contributed by Roland Barcia, IBM Software Group

The IBM WebSphere Studio [IBM, WSAD] fulfills the vision of JSF by providing full RAD capabilities for developing enterprise-class web applications. Using drag-and-drop technology, developers can quickly assemble their web pages and link visual controls using various data components, such as Service Data Objects (SDO),² Enterprise JavaBeans, or web services. In addition, you can use the Portal Toolkit to develop JSF-enabled portlets that run inside WebSphere Portal Server.

Version 5.1.1 of WebSphere Studio provided a technical preview based on an early JSF draft specification. Version 5.1.2 is the official production-ready release with full support for JSF 1.0. It lets you develop standard J2EE applications as well as Java Specification Request (JSR) 168 portlets using JSF.

The following sections provide an overview of WebSphere Studio's JSF capabilities, and an example using ProjectTrack. For a more complete tutorial, see Roland

² SDO is a specification developed by IBM and BEA that provides a simple API to access heterogeneous data sources (web services, relational databases, XML files, EJBs, and so on). It has been submitted as JSR 235.

Barcia's Developing JSF Applications Using WebSphere Studio V5.1.1 series [Barcia Series 2004].

B.2.1 Exploring WebSphere Studio

WebSphere Studio is based on the open source Eclipse project, so it allows developers to write plug-ins to extend the IDE and make use of the many plug-ins currently on the market. Eclipse works on the notion of perspectives and views. JSF developers will work in the web perspective, which provides all the views needed to assemble JSF pages. Figure B.30 shows the overall workbench.

The web perspective provides access to the standard Project Navigator view and a Visual Design editor. It also has a few views specifically for web development:

- *Page Data*—The Page Data view enables you to define JSP and/or JavaScript variables on the page. You can then use the Attributes view to bind a visual component to that data. Data can be as simple as a String or as complex as a web service.

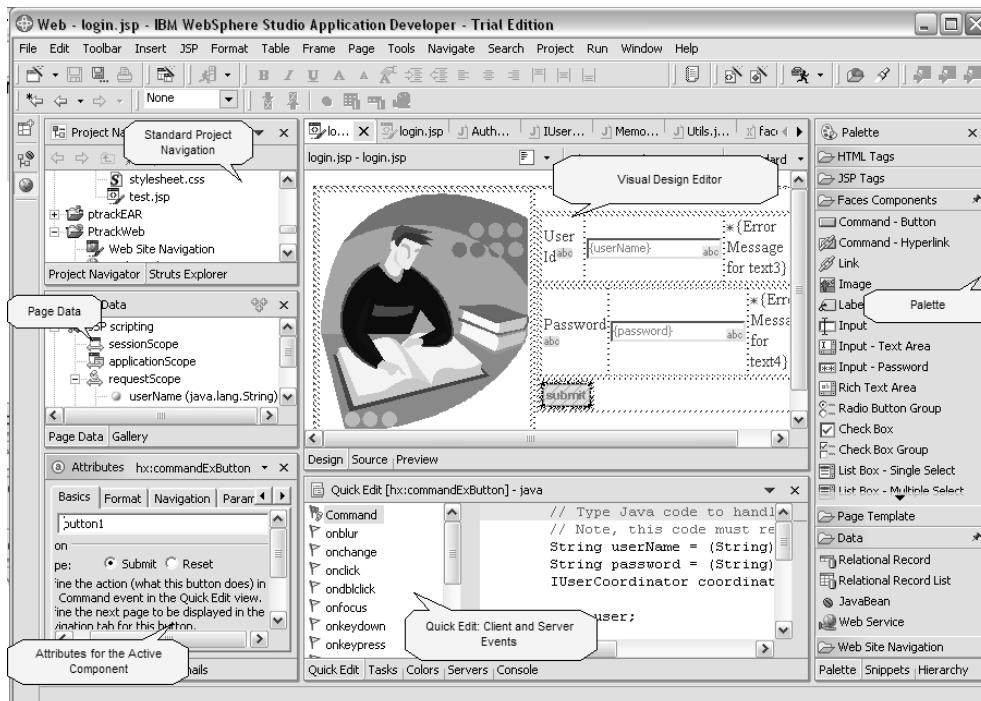


Figure B.30 WebSphere Studio web perspective.

- *Palette*—The Palette view contains various palettes with many components that you can drag on a page, ranging from standard JSF components to WebSphere data components.
- *Quick Edit*—The Quick Edit view lets you write code snippets that respond to various server-side JSF events or client-side events (with JavaScript).
- *Attributes*—Every component has attributes (also called properties) that you must configure. The Attributes view enables you to configure the current highlighted component.

In addition to the views, these features are available to JSF developers:

- Special palettes
 - ❑ *Faces palette*—Contains the standard JSF components mandated by the specification. In addition, you will find some IBM components that extend the standard components.
 - ❑ *Faces Client components*—These are special components with embedded JavaScript that aid in caching data and enhance performance.
 - ❑ *Data components*—Allow developers to drag special components that are automatically bound to data. These components include JavaBeans, SDOs, and web services.
 - ❑ *Struts palette*—For Struts applications, there is a palette of standard Struts constructs, such as the HTML tags that you can drag on a page.
 - ❑ *Standard palettes (JSP, HTML)*—You can also drag standard JSP or HTML constructs onto a page.
 - ❑ *Web site navigation*—Allows you to drag navigation components and build site maps for a page.
 - ❑ *Page template*—Allows developers to build static page templates.
 - ❑ *Portlet palette*—Contains special components for portlet developers who wish to deploy to WebSphere Portal (part of the WebSphere Portal Toolkit).
- Wizards
 - ❑ *Create Projects*—This wizard lets you create many different project types, such as web services, EJB projects, and database projects. JSF developers will create dynamic web pages most of the time.
 - ❑ *Create Pages*—Allows you to quickly create pages and apply templates and models.

B.2.2 Building ProjectTrack's Login page

To demonstrate some of the capabilities of WebSphere Studio, we will examine the steps necessary to build ProjectTrack's Login page.

Creating a new project

The first step in JSF development is setting up the right type of project. In WebSphere Studio, you can create what is called a *dynamic web project*, which maps to a J2EE WAR file. A wizard that steps you through the project creation process is accessible from the main menu (choose File, then New) or via certain context menus, depending on the view you are in. The New Dynamic Web Project wizard is shown in figure B.31.

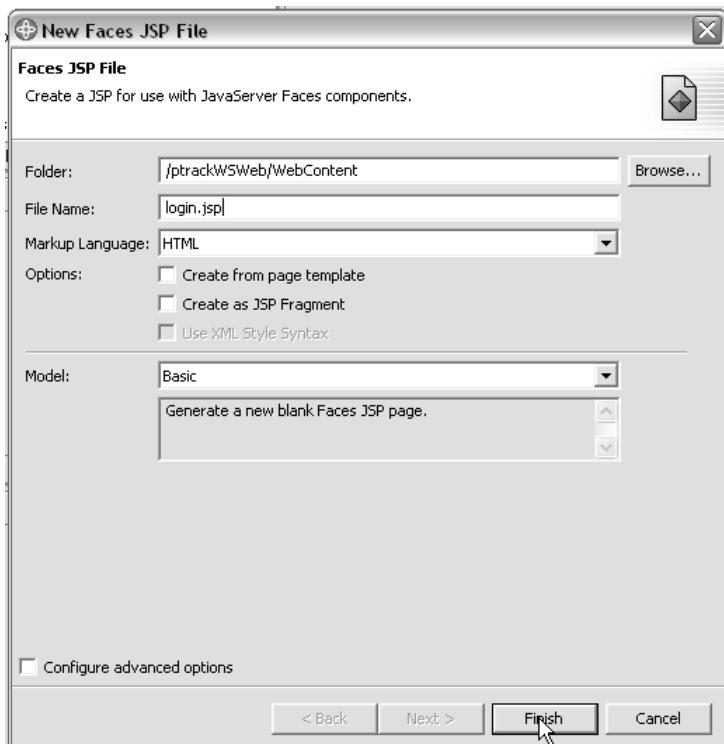


Figure B.31 The New Dynamic Web Project wizard.

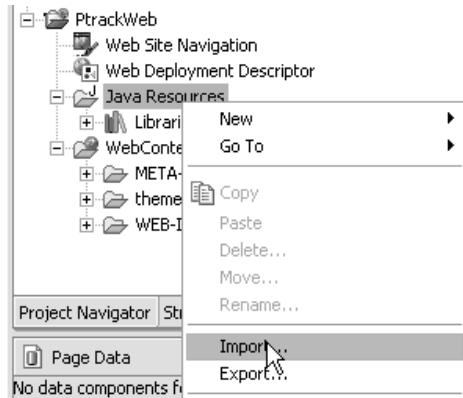


Figure B.32 Importing Java resources.

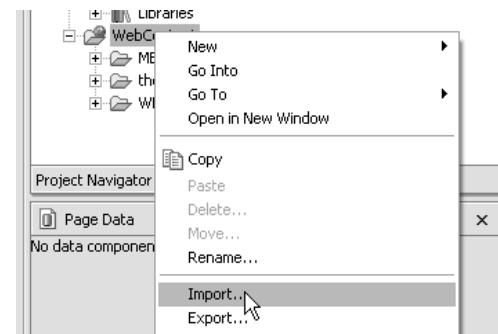


Figure B.33 Importing web artifacts.

After creating the web project, import the ProjectTrack domain classes into the Java Resources folder in the Project Navigator. Classes placed here are automatically compiled into the classes directory of the WAR file, as shown in figure B.32.

Similarly, you import web artifacts such as images into the WebContent folder, as shown in figure B.33.

Creating a new page

Once your project is set up with the necessary artifacts, you can create a Faces JSP page just by right-clicking the desired folder and selecting New, then Faces JSP File, as shown in figure B.34.

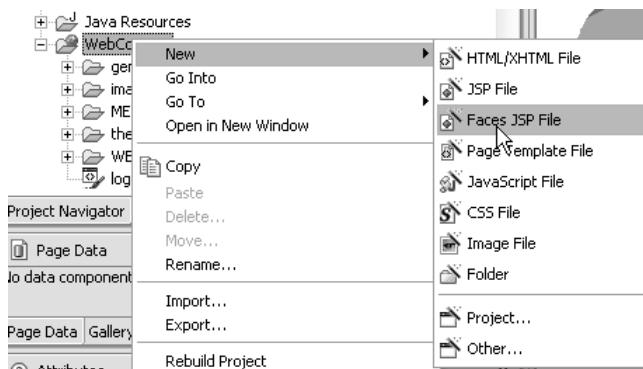


Figure B.34
Creating a JSF page.

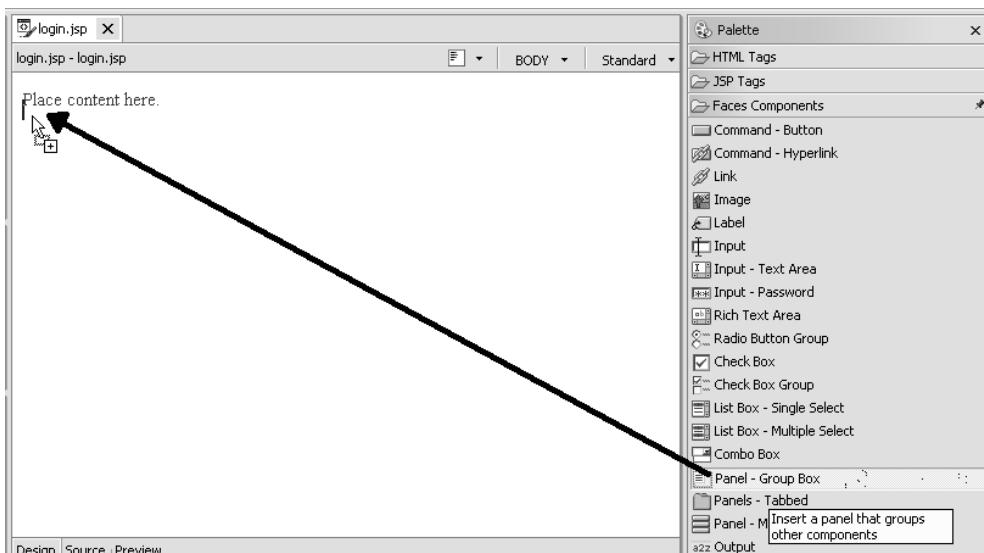


Figure B.35 Dragging a panel into the design page.

Designing the page

Once the JSF page is created, open it using the WebSphere Studio web page editor. The editor contains various tabs. The Design tab serves as the canvas for JSF components that you drag and drop from the Palette view. As you'll recall, many of the ProjectTrack pages made use of nested panels, and WebSphere Studio has strong support for panels.

The first step is to drag and drop the panel layout desired, as shown in figure B.35. You can drag a panel inside another panel to get the effect you want.

As you drag each panel, a dialog box appears in which you specify the type of panel you want (see figure B.36).

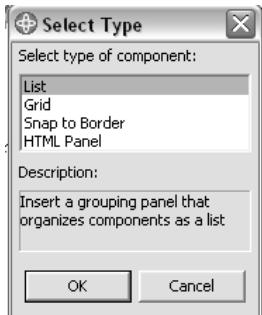


Figure B.36
Selecting the panel type.



Figure B.37 Dragging an image into a panel.

After you finish setting up your panels, you can drag the desired component into the panel. In figure B.37, we've dragged the `Image` component into the outer panel.

Once you have placed the component on the page, you can modify its attributes for the page. Select the component and use the Attributes view to specify the attributes. In figure B.38, we've assigned the actual image file to the JSF `Image` component.

Figure B.39 shows the `Image` component after we've assigned the correct attribute. The set of attributes displayed in the Attributes view changes depending on the currently selected UI component.

Let's continue building the page by dragging the desired components and modifying their attributes. Figure B.40 shows an overview of the components we're using for the Login page.

Adding validation

You can also use the Attributes view to add validation to the component. Simply select the Validation tab of the Attributes view, as shown in figure B.41.

You can add standard validation to each component and even generate an `HtmlMessage` component to display error messages. Figure B.42 shows a closer view of the Validation tab. As you can see, you can add behavior based on the validation results by selecting the Add Display Error checkbox.

This tab resembles a wizard for configuring the most common set of validation options. It generates the standard validation tags nested in the component, like so:

```
<f:validateLength minimum="6" maximum="20"></f:validateLength>
```

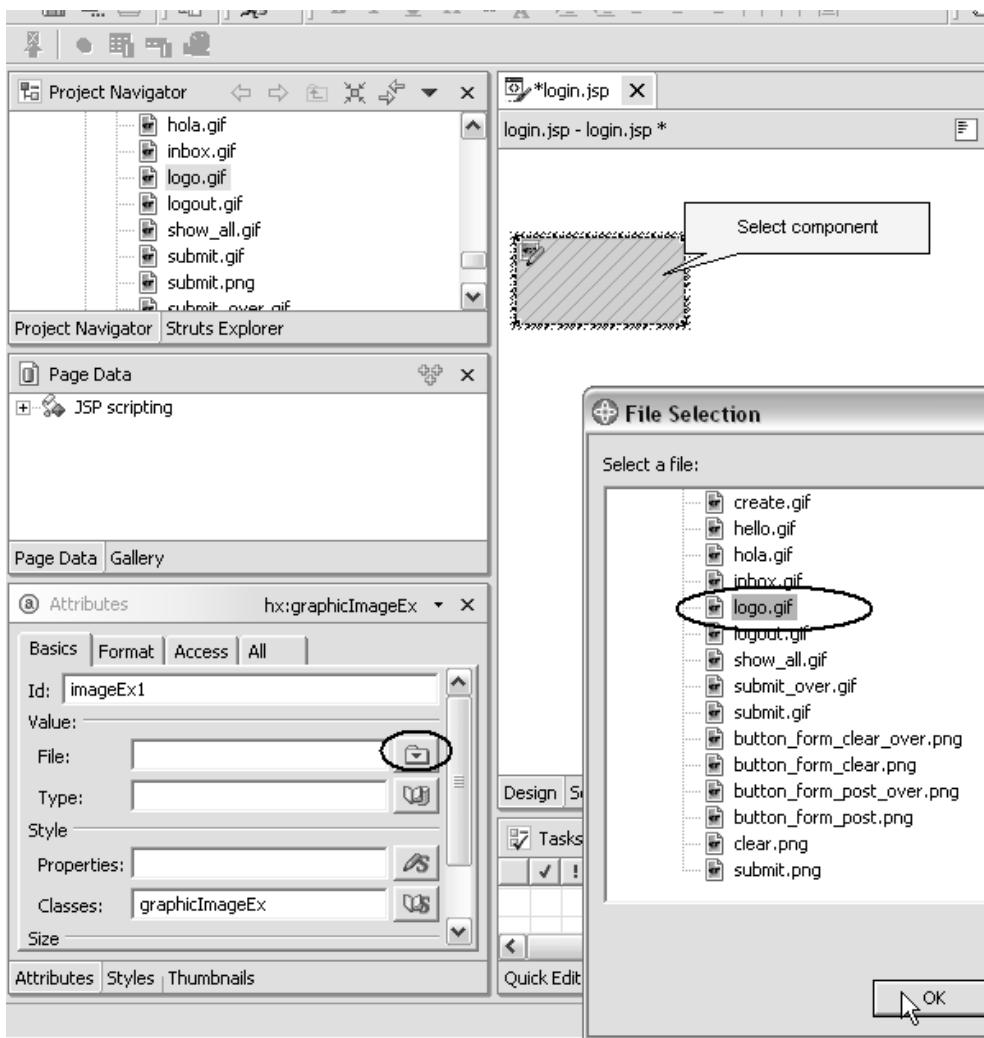


Figure B.38 Associating the `Image` component with `logo.gif`

Binding components to backing beans

UI components properties can be bound to backing beans through the Attributes view. The first step is to define some page data, which you can do easily by using the Page Data view. You just select the desired JSP scope and define variables. These variables can be simple types, such as `String`, or more complex types, such as JavaBeans or web service proxies. For ProjectTrack, you must define two



Figure B.39
The `Image` component with the `ProjectTrack` graphic.

variables: `userName` and `password`. You can do this by right-clicking on the request-Scope node and selecting Add Request Scope Variable, as shown figure B.43.

This will bring up a dialog box asking you to define its name and type. (See figure B.44.)

Once both variables have been added, the Page Data view displays the data available to the page, as shown in figure B.45.

Once you have defined the data on the page, you can bind dynamic controls to the variables. From the Attributes view, go to the Basics tab and select the search box next to the Value text box. This will bring up a dialog box containing the available page data, as shown in figure B.46.

In addition to server-side variables, controls can be bound to client-side JavaScript elements.

By exposing data on the page, you can access it with the standard JSF expression language. In addition, you can access the data in action listeners using special `HashMap` helpers as shown here:

```
String userName = (String)requestScope.get("userName");
String password = (String)requestScope.get("password");
```

Because the login page accesses the `IUserCoordinator`, which is stored in the application scope, you can add this class to the application scope through the Page Data view, as shown in figure B.47.

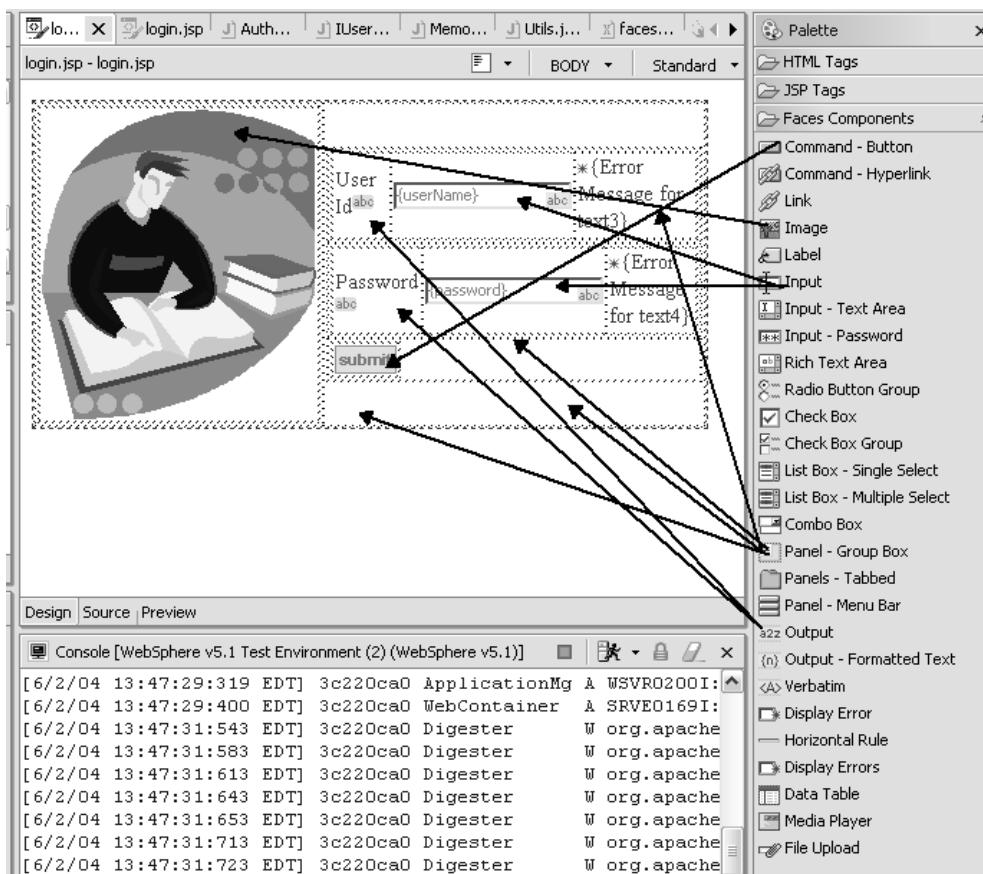


Figure B.40 The various components used in the Login page.

This will bring up a dialog box similar to the one shown in figure B.48. You can use the help box (click the ellipsis button next to the Type text box) to find the appropriate class.

Once the `userCoordinator` has been added, it appears in the Page Data view, as shown in figure B.49.

Adding action listeners

In WebSphere Studio, you can handle UI component events with the Quick Edit view. This view allows you to add snippets of Java code for server events or snippets of JavaScript for client events. The Quick Edit view also shows all possible events (server or client) for a component. For ProjectTrack, we are interested in

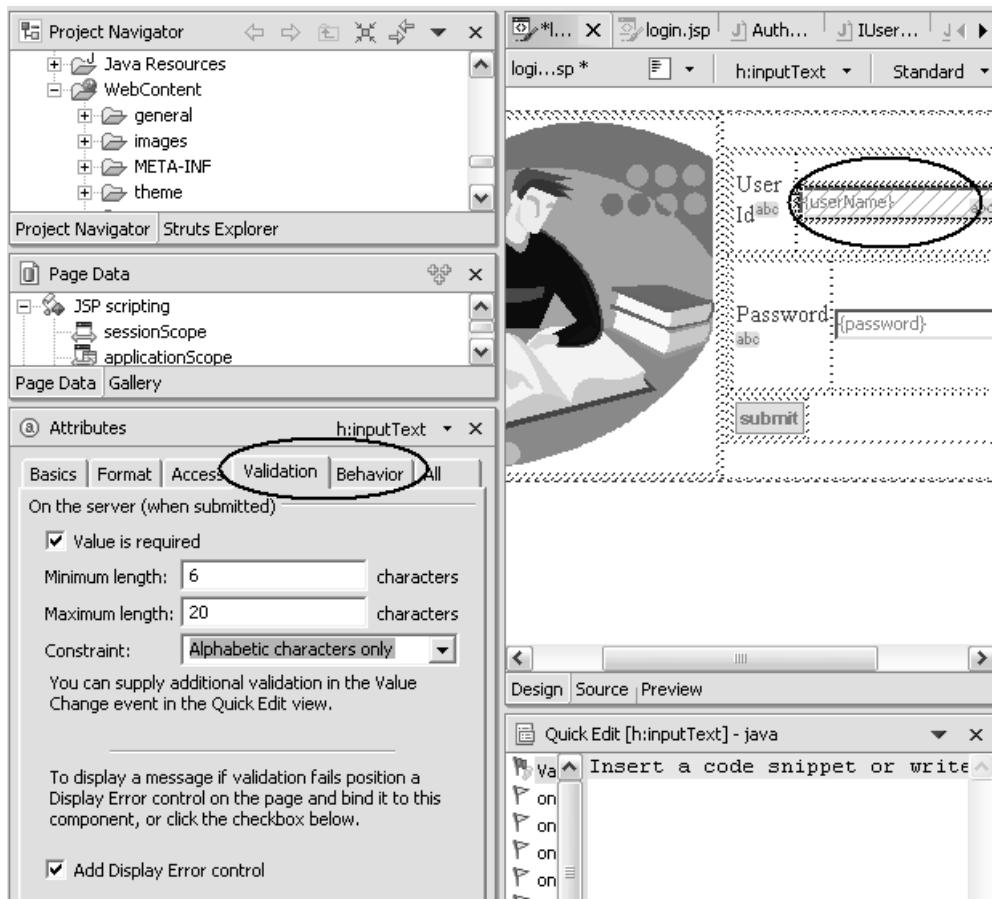


Figure B.41 Adding component validation.

server events. By clicking the Command button, you can access the Quick Edit view and select the Command event. Then you can write the code for the action method that handles the event (called a Java snippet), as shown in figure B.50.

We added the appropriate code into the Quick Edit view, as shown in figure B.51.

The code in listing B.1 is the full Java snippet from figure B.51. This is the same as the login method shown in listing 13.1 from chapter 13, modified to use instance variables generated by WebSphere Studio.

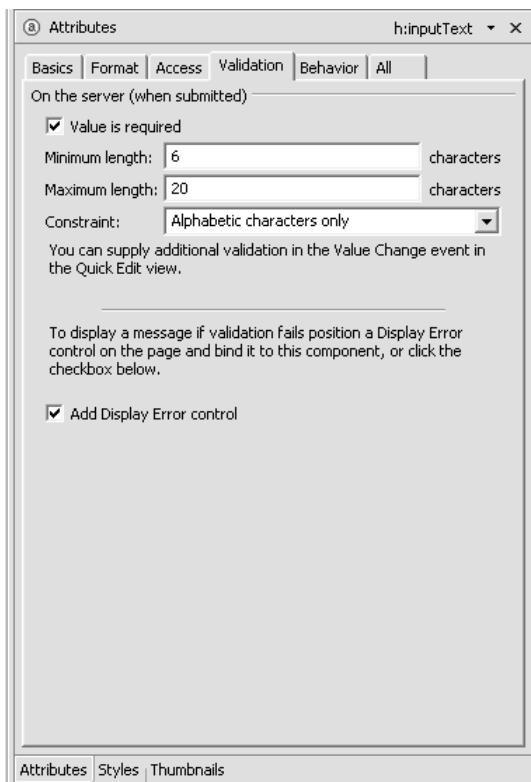


Figure B.42 Adding validation with the Attributes view.

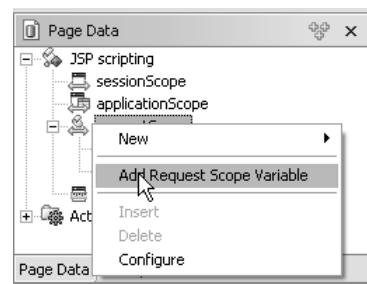


Figure B.43 Defining page data.

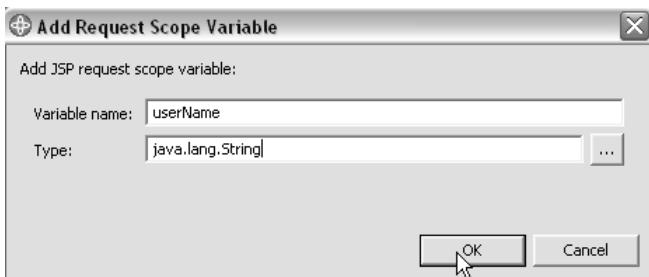


Figure B.44 Adding a request-scoped variable.

Listing B.1 Java code for the login action method under WebSphere Studio

```
// Type Java code to handle command event here
// Note, this code must return an object of type String (or null)
String userName =
    (String)requestScope.get("userName");
String password =
    (String)requestScope.get("password");
IUserCoordinator coordinator = (IUserCoordinator)
    applicationScope.get(Constants.USER_COORDINATOR_KEY);
User newUser;
try
{
    newUser = coordinator.getUser(userName, password);
}
catch (ObjectNotFoundException e) {
    return Constants.FAILURE_OUTCOME;
} catch (DataStoreException e) {
    return Constants.ERROR_OUTCOME;
}
Visit visit = new Visit();
visit.setUser(newUser);
FacesContext fContext = FacesContext.getCurrentInstance();
fContext.getApplication().createValueBinding(
    "#{" + Constants.VISIT_KEY_SCOPE + Constants.VISIT_KEY +
    "}") .setValue(facesContext, visit);
if (newUser.getRole().equals(RoleType.UPPER_MANAGER))
{
    return Constants.SUCCESS_READONLY_OUTCOME;
}
return Constants.SUCCESS_READWRITE_OUTCOME;
```



Figure B.45
The Page Data view displaying request-scoped variables.

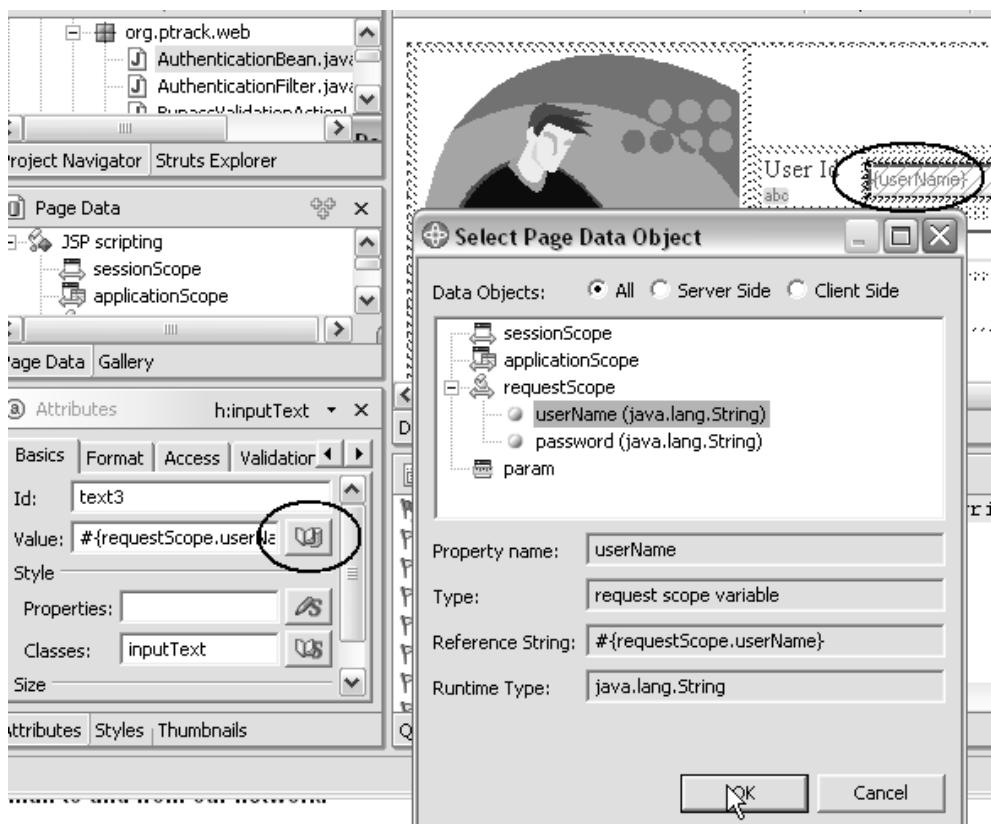


Figure B.46 Binding controls to page data.

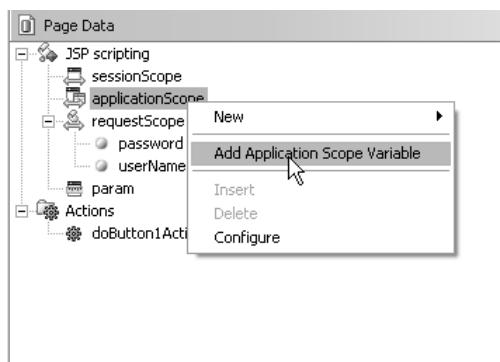


Figure B.47
Adding an application-scoped
variable.

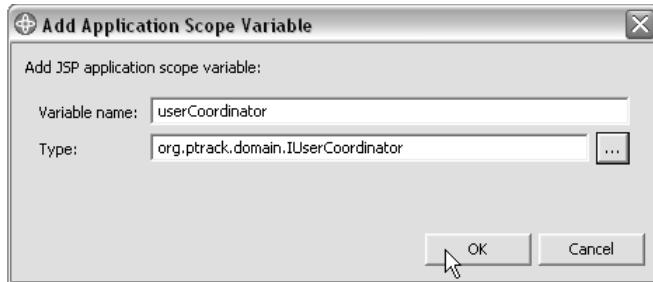


Figure B.48 Adding the `IUserCoordinator` to application scope.

In case you are wondering, WebSphere Studio creates one backing bean for each page, called *page code*. The code snippets are added as event listener methods in the page code. Page code classes subclass a WebSphere Studio-specific class that initializes the data from the component tree and handles many repetitious JSF programming tasks, much like ProjectTrack's `BaseBean` class.

Instead of using backing bean properties, we use the WebSphere Studio-generated instance variables (marked in bold in listing B.1) in the page code to access the `userName` and `password` variables. Also, if you're wondering why we're retrieving these values from the request scope instead of through properties of the backing bean, it's because we added these variables as page data instead of adding properties to the bean itself.

Figure B.52 shows the page code for the Login page and its superclass, `PageCodeBase`.



Figure B.49 Page Data view with `IUserCoordinator`.

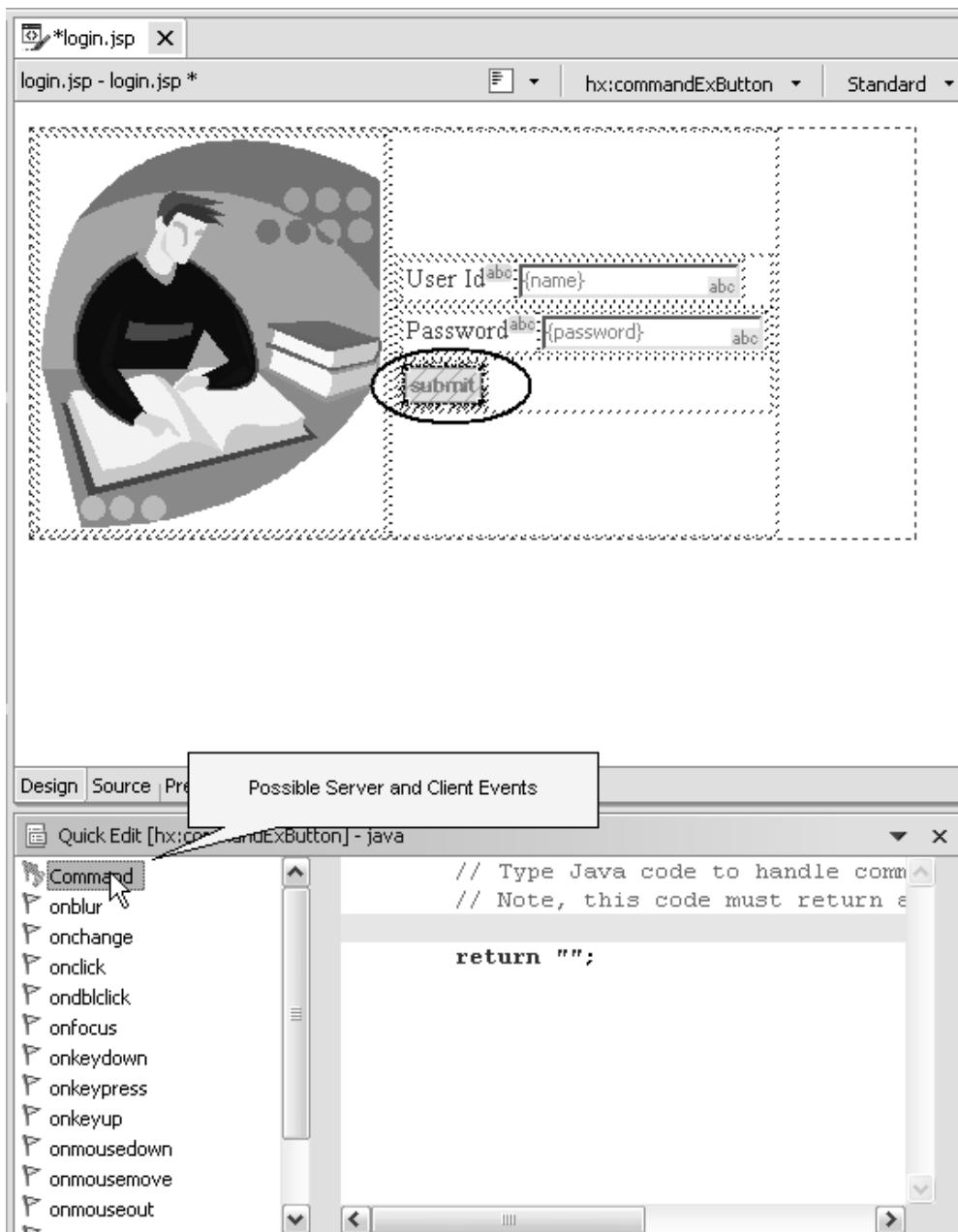


Figure B.50 Selecting a server event.



Figure B.51 Writing an event listener.

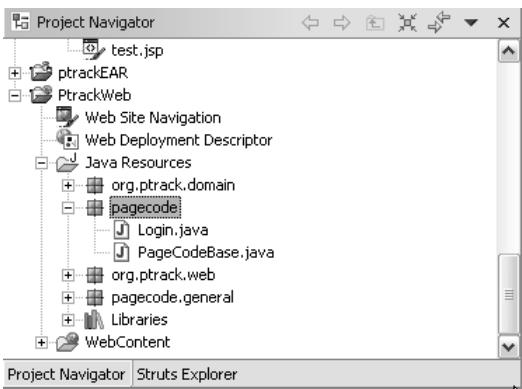


Figure B.52
WebSphere Studio-generated backing beans (called page code) selected in the Project Navigator view.

Page code classes are also added to the faces-config.xml as managed beans. The managed bean declaration for the Login page is shown here:

```
<managed-bean>
    <managed-bean-name>loginBean</managed-bean-name>
    <managed-bean-class>pagecode.Login</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Adding navigation

You can use the Attributes view to quickly add navigation to your JSF application. Select the UI component that triggers the server event (in our case the Submit button) and go to the Attributes view. From there you can select the Navigation tab and quickly add a new rule, as shown in figure B.53. This will bring up a dialog box that allows you to create a standard JSF navigation rule (see figure B.54).

The Attributes view will automatically add the navigation rule to the faces-config.xml file. Listing B.2 contains the navigation rule generated by the dialog box in figure B.54.

Listing B.2 The navigation rule generated by the dialog box in figure B.53

```
<navigation-rule>
    <from-view-id>/login.jsp</from-view-id>
    <navigation-case>
        <from-action>#{pc_Login.doButton1Action}</from-action>
        <from-outcome>success_READONLY</from-outcome>
        <to-view-id>/general/show_all.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

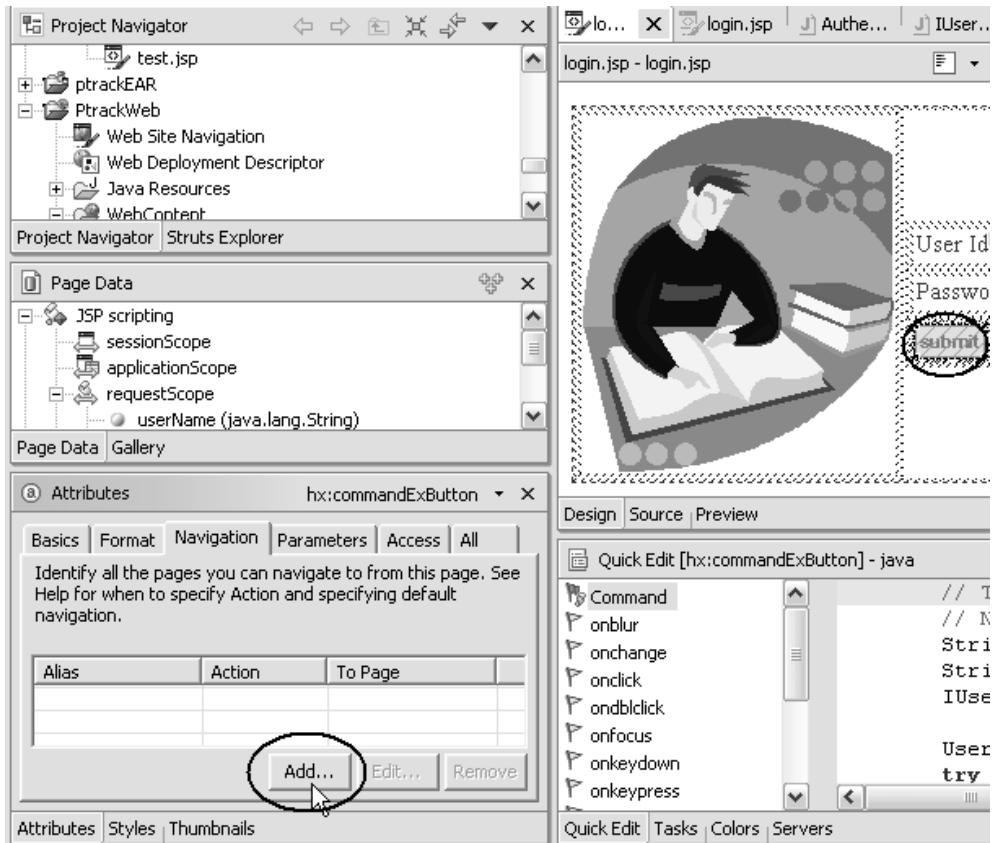


Figure B.53 Adding a navigation rule.

NOTE The dialog box in figure B.54 refers to a logical outcome as an *alias*. IDEs will sometimes use different terminology for standard JSF concepts.

Once a new navigation rule has been added, it is listed in the Attributes view, as shown in figure B.55.

Testing

WebSphere Studio provides the ability to test JSF pages at various levels. You can use the Preview mode to view the page layout, as shown in figure B.56.

In addition, you can deploy the application to application servers such as Tomcat or WebSphere Application Server. You can generate a WebSphere test environment by right-clicking the page and selecting Run On Server, as shown in figure B.57.



Figure B.54
Creating a navigation rule.
This dialog box generates the output shown in listing B.2.

The screenshot shows the 'Navigation' tab of the component editor for an 'hx:commandExButton' component. The tab bar includes 'Attributes', 'Format', 'Navigation' (which is selected), 'Parameters', 'Access', and 'All'. A note says: 'Identify all the pages you can navigate to from this page. See Help for when to specify Action and specifying default navigation.' Below is a table:

Alias	Action	To Page
success_READONLY	#{pc_Login.doButton1Action}	/general/show_all.jsp

At the bottom are 'Add...', 'Edit...', and 'Remove...' buttons, and tabs for 'Attributes', 'Styles', and 'Thumbnails'.

Figure B.55
All navigation rules for the Submit button.



Figure B.56 Previewing the page.

If this is the first time you've attempted to run on a server, WebSphere Studio will allow you to select the target server. In this example we chose WebSphere version 5.1, as shown in figure B.58.

For WebSphere Application Server version 5.1, you can choose to generate a local WebSphere Application Server configuration or attach to a running server, as shown in figure B.59.

Once configured, WebSphere Studio will bring up a browser accessing your application, which you can test as you see fit. The browser is shown in figure B.60.

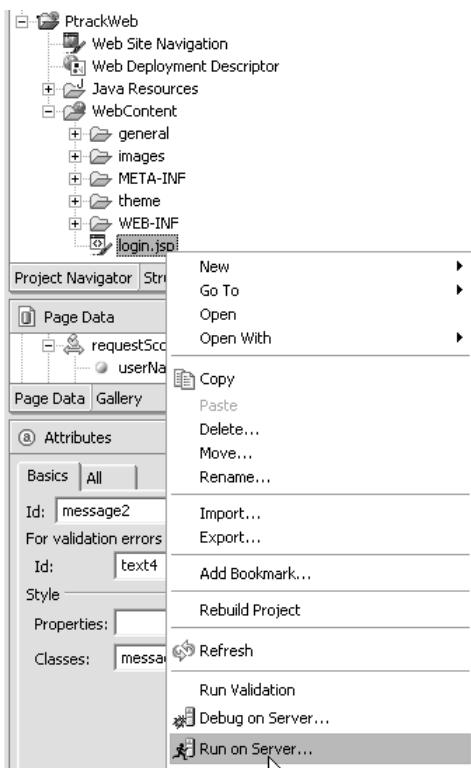


Figure B.57
Running the application on a server.

B.2.3 Wrapping up

In this section, we only touched the surface of WebSphere Studio's feature set. The IDE provides many different ways you can build JSF applications. For example, you can choose to bind more complex data grids to complex objects. Figure B.61 shows the Page Data view and the available complex objects you can bind. You can

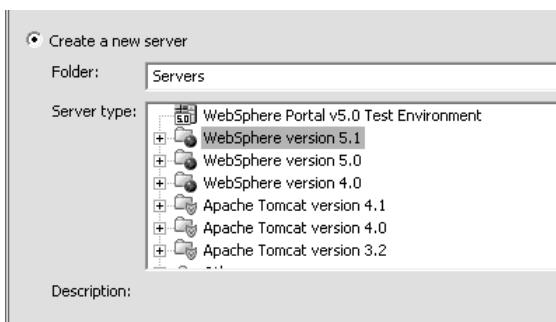


Figure B.58
Selecting an application server.

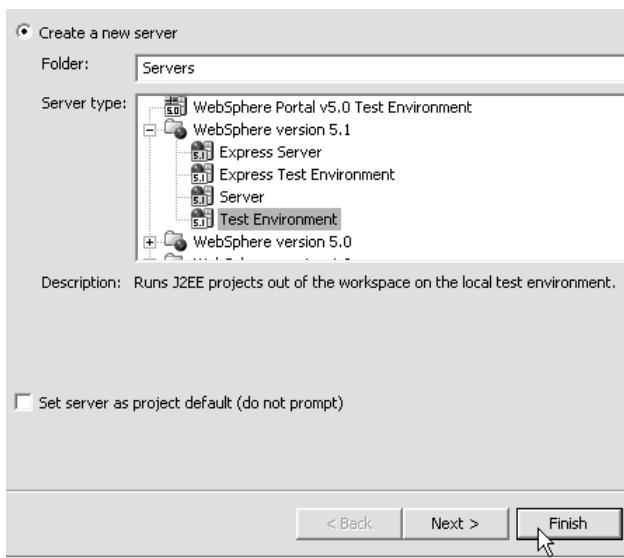


Figure B.59
Selecting a local server.

create data components and then visually drag them from the palette or Page Data view and generate JSF data grid components.



Figure B.60 The Login page shown in a browser.

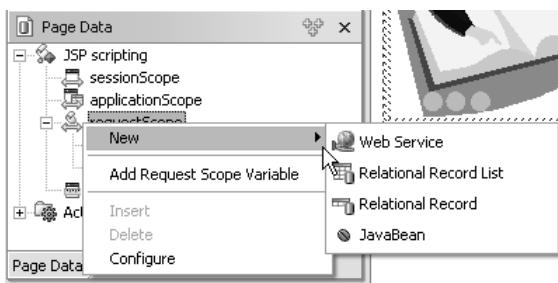


Figure B.61
Creating new data components.

B.3 JSF and Java Studio Creator

At the JavaOne conference in June of 2004, Sun released version 1.0 of its long-awaited Java Studio Creator [Sun, Creator] IDE, which is based on its NetBeans IDE. Creator (formerly code-named *Rave*) is a different breed of Java IDE, simply because it doesn't have a plethora of features. It's designed to visually build Java web applications that communicate directly with a database or a web service. Creator is simple because it allows you to build *only* Faces applications (it doesn't expose any additional NetBeans functionality). There's no support for JSTL or any other custom JSP tag libraries. The product is targeted toward the so-called corporate developer, who is more interested in getting results quickly than wasting time deciding which type of project to build.

B.3.1 Using Java Studio Creator

When you launch Creator, you'll be struck by its similarity to Microsoft Visual Studio .NET. The workspace consists of a set of dockable windows, including a palette, the Server Navigator, a property sheet, and so on. Figure B.62 shows Creator's workspace.

Creator is visually oriented—you can make a lot of progress simply by dragging and dropping UI components, validators, converters, and even data sources onto the page, and then customizing their properties.

The tool has several key windows that can be utilized within the workspace:

- *Palette*—The Palette has two modes: JSF and Clips. The JSF mode displays UI components, validators, converters, and some core JSF tags (such as `<f:loadBundle>`). UI components can be dragged from the Palette and dropped into the Editing Area (when it is in design mode) or the Application Outline. The Clips mode contains code snippets (ranging from simple conditionals to code for manipulating data sources) that can be dragged and dropped into the Editing Area in source mode.

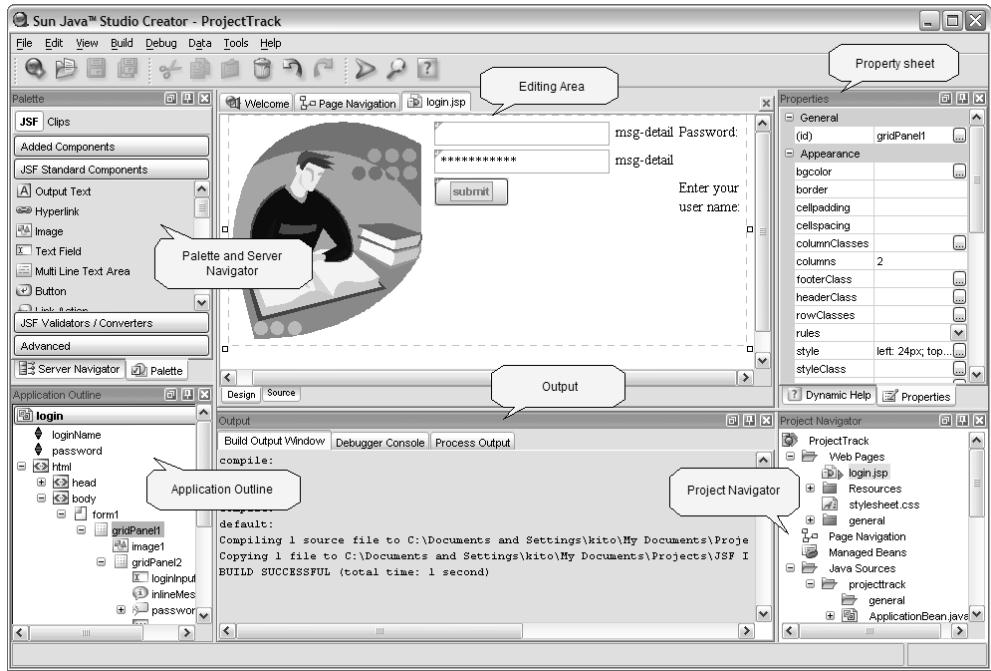


Figure B.62 The Creator workspace consists of several dockable windows.

- **Server Navigator**—Like Visual Studio .NET, Creator allows you to navigate through configured data sources, which include databases and web services. Data sources from the navigator can be dragged and dropped into the Editing Area (in design mode) or the Application Outline.
- **Editing Area**—This is where you manipulate a page with UI components, edit source code (JSP, Java, or XML), or visually build your application's navigation rules.
- **Property sheet**—The property sheet allows you to edit the properties of the selected object. Most often it is used to edit the properties of a UI component, but it can also manipulate validators, converters, the entire page, and even Java classes.
- **Application Outline**—This window shows the tree structure for pages—all of its UI components, as well as HTML elements. It also displays properties exposed in any beans your project contains. You can drag and drop JSF user interface extensions (UI components, validators, and converters), as well as data sources into this window. Any time you select an item, its properties can be edited in the property sheet.

- *Output*—This window has tabs like Build Output and Process Output.
- *Project Navigator*—The Project Navigator lists all of the resources in your project, including JSP pages, images, style sheets, Java source files, and a list of referenced libraries.

Creator has several other useful features, including the following:

- Extensive support for building JavaBeans (including creating BeanInfo)
- The ability to import existing data sources and web services (and generate the appropriate wrappers)
- Deep integration with data sources, including a visual SQL editor, and extensive use of JDBC RowSets
- Easy testing (no need to configure a server)

B.3.2 Building ProjectTrack's Login page

Now that we've discussed Creator from a bird's-eye view, let's delve into some details by building ProjectTrack's Login page.

Creating a new project

When you first start up the IDE, the Editing Area has a welcome page (much like the page in Visual Studio .NET). You can create a new JSF project by clicking the Create New Project button in that page, or by selecting New Project from the File menu. Either way, Creator will display the New Project dialog box, shown in figure B.63.

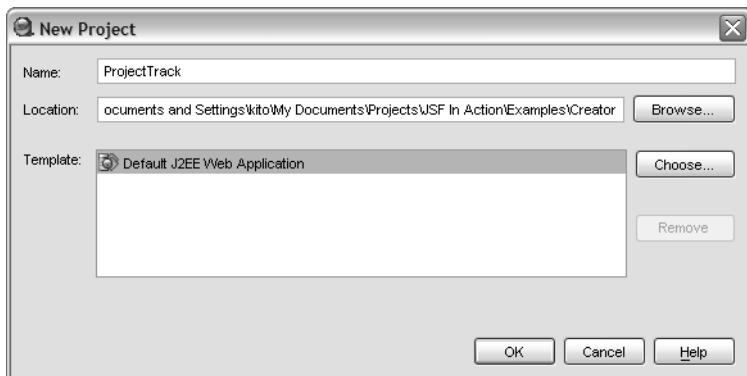


Figure B.63 The New Project dialog box.

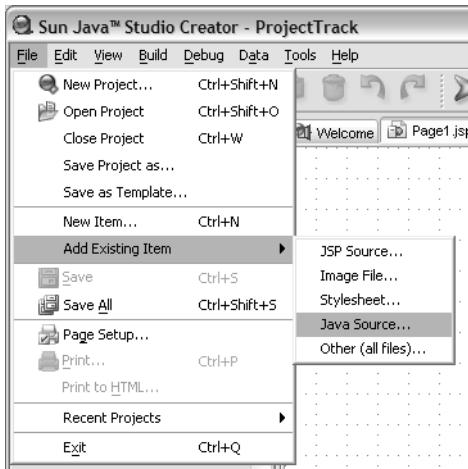


Figure B.64
Importing existing items.

As you can see, there's no choice for the type of application you'd like to create—all Creator projects generate JSF applications. It is possible, however, to define different project templates.

Once you've created a project, you can add existing resources using the File menu's Add Existing Item command, as shown in figure B.64. In our case, this command can be used to import ProjectTrack's domain classes, images, and style sheet.

NOTE The Add Existing Item command allows you to import only a single file at a time. If you have several files, you can copy them into the project's directory structure, and Creator will notice them (if you reopen the project).

When you create a new project, Creator will generate a default view called Page1.jsp. It will also generate three beans: Page1 (the backing bean), SessionBean1 (for session data), and ApplicationBean1 (for application-scoped properties). All three of these classes subclass Creator-specific base classes that have helper methods (much like the ones in ProjectTrack's BaseBean). The superclasses also implement the PhaseListener interface, so that they can handle lifecycle-related events. All of these beans are registered as managed beans:

```
<faces-config>
    <managed-bean>
        <managed-bean-name>Page1</managed-bean-name>
        <managed-bean-class>projecttrack.Page1</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
```

```
</managed-bean>
<managed-bean>
    <managed-bean-name>SessionBean1</managed-bean-name>
    <managed-bean-class>
        projecttrack.SessionBean1
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
<managed-bean>
    <managed-bean-name>ApplicationBean1</managed-bean-name>
    <managed-bean-class>
        projecttrack.ApplicationBean1
    </managed-bean-class>
    <managed-bean-scope>application</managed-bean-scope>
</managed-bean>
</faces-config>
```

Creator generates a default CSS stylesheet as well.

Building pages with JSF components

The first step for the Login page is renaming Page1.jsp as login.jsp, which you can do in the Project Navigator. Changing the page's name automatically changes the name of the corresponding backing bean and updates any dependent resources.

When you're designing the page, you must select one of two component layout types: Grid or Flow. Grid layout is absolute positioning—your components are placed exactly where you drop them on the page. With Flow layout, your components are displayed from left to right, and simply wrap around when the width of the screen has been reached. Creator uses Grid layout by default, but Flow layout is a better choice for web applications that must work with a variety of browsers. You can change the layout in the property sheet for the page, as shown in figure B.65.

Once you have the proper layout, all you need to do is drag the UI components from the Palette into the page. Let's start with a panel, as shown in figure B.66.

Once the panel is on the page, you can customize its properties in the property sheet.

Let's continue building this page by adding an image to the panel, as shown in figure B.67.

As soon as the component is dropped onto the page, Creator pops up a dialog box asking you which image to select (either a file or URL). Also, as with any component, the Application Outline shows the new component within the page's tree view.

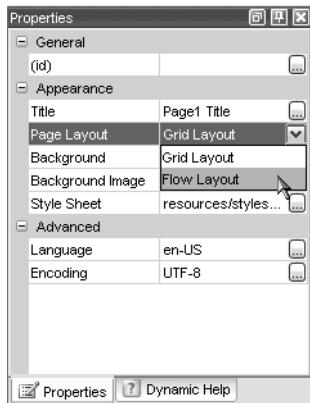


Figure B.65 Changing the page layout with the property sheet.

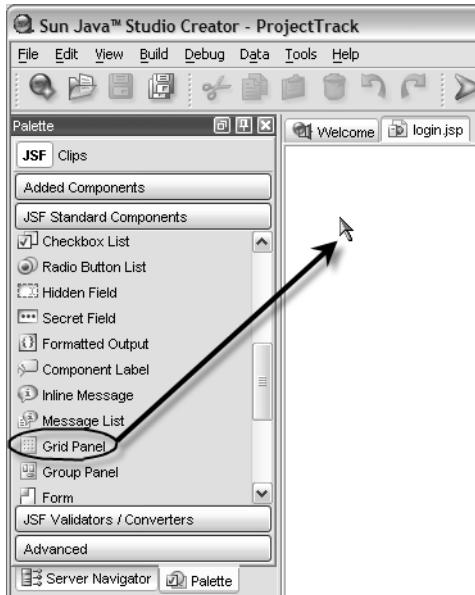


Figure B.66 Adding a panel to the page.

Building the rest of the page's UI is as simple as dropping the remaining components on their page and customizing their properties. Figure B.68 shows the completed page and its corresponding components.

NOTE If you're wondering why some of the controls appear in the wrong order in figure B.68, it's because there is a bug in the version of Creator used in this book that causes components in nested panels to be displayed incorrectly. The generated application works as expected, however.

Adding validation

Now that we have designed the page, let's add validation. In Creator, you can add validators by dragging them from the Palette and dropping them into the Application Outline, or through the property sheet, as shown in figure B.69.

Once you've created a new validator, it appears in the Application Outline. In this window, you can select it and edit its properties through the property sheet, as shown in figure B.70.

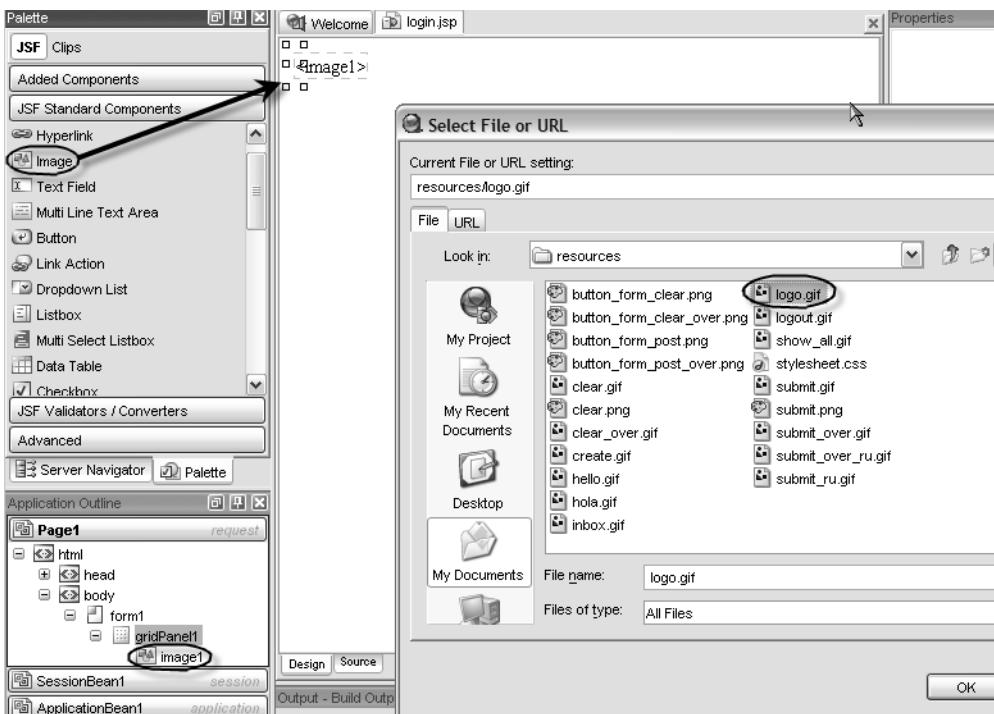


Figure B.67 Adding an image to the panel.

Binding components to properties

In Creator, you usually bind UI components properties to properties of the current page's backing bean. Data sources are added as properties automatically when you drop them onto the page. You can create your own properties by editing the backing bean manually, or by right-clicking on the bean in the Project Navigator, as shown in figure B.71.

This menu brings up a New Property dialog box, like the one shown in figure B.72. We can use this method to create the `loginName` and `password` properties of our backing bean.

You can add properties to any bean in a project, not just backing beans. Project-Track's login process requires access to an `IUserCoordinator` instance, so we must add a reference to it in the project somewhere. Because the `IUserCoordinator` lives for the application's lifespan, we can add it as a property of the `ApplicationBean`. Once we've added the property, we also need to initialize it. We can access a bean's code by double-clicking on the class in the Project Navigator. In figure B.73, we're

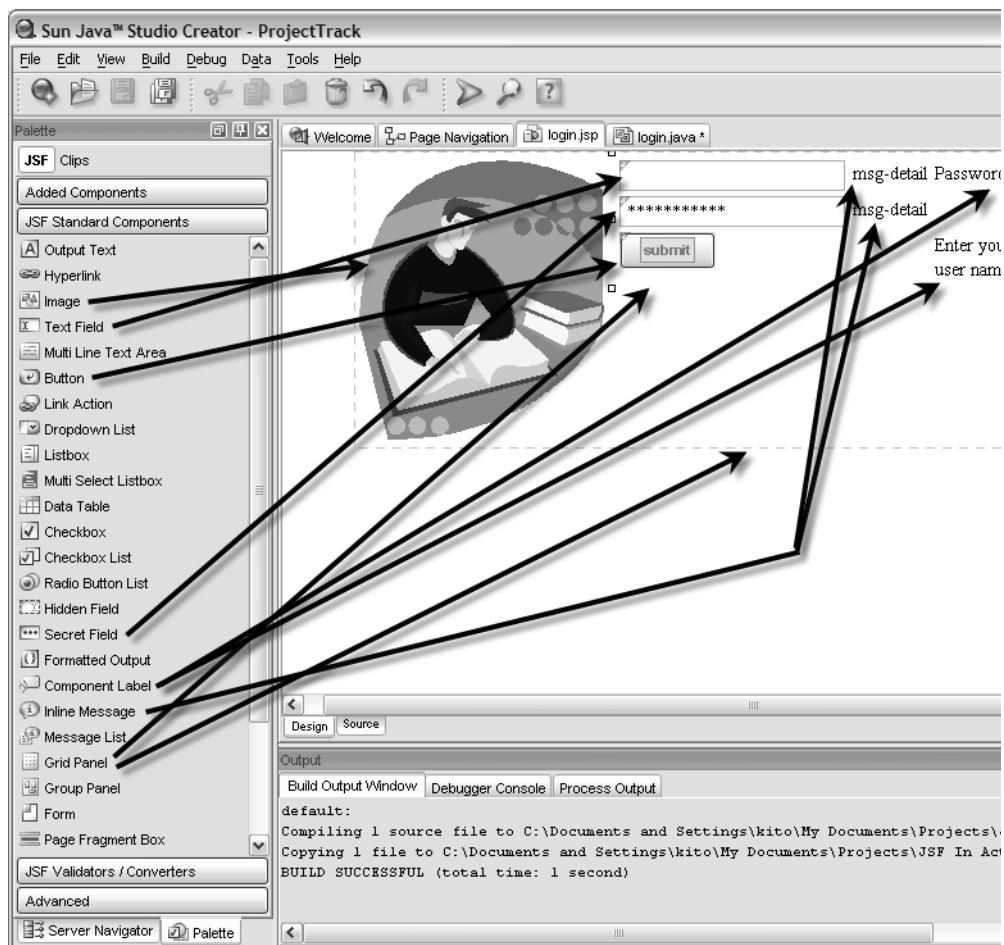


Figure B.68 The completed Login page and its corresponding UI components.

editing ApplicationBean's constructor to create a new MemoryUserCoordinator (which implements IUserCoordinator).

Now that we've added all of the necessary properties, it's time to bind them to UI components. This can be done by editing the value property of the selected component with the property sheet, as shown in figure B.74. The dialog box Creator displays when editing an expression greatly simplifies the process of creating value-binding expressions. You can also set value-binding expressions for many other properties at once with the Property Bindings dialog box (shown in

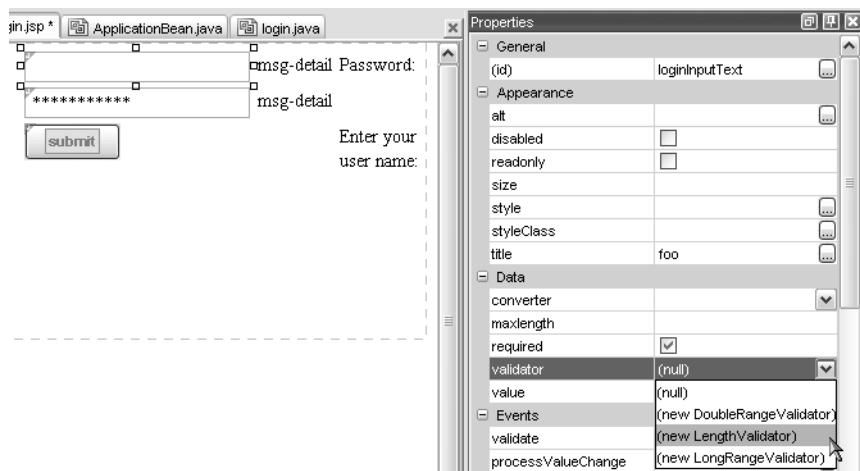


Figure B.69 Creating a new validator for an input field.

figure B.75), which you open by right-clicking on a UI component and choosing Property Bindings. This dialog box should be familiar to users of Visual Studio .NET.

Adding action listeners

In the world of Creator, event listeners are called event *handlers*. If you double-click on a component, the IDE will take you to the default event handler in its backing

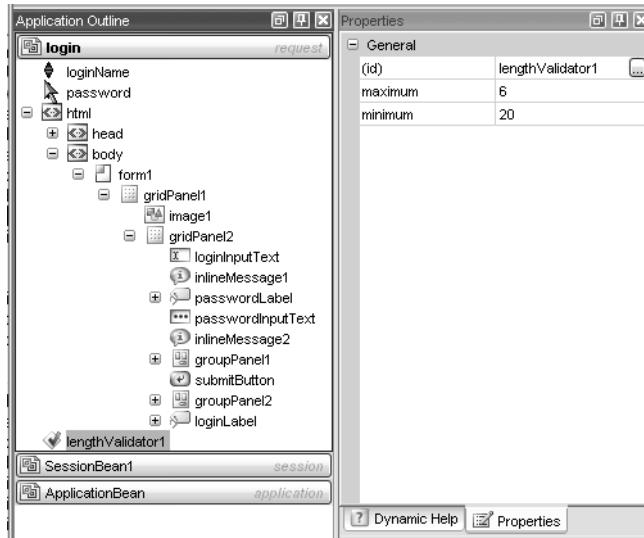


Figure B.70
Editing a validator's properties.

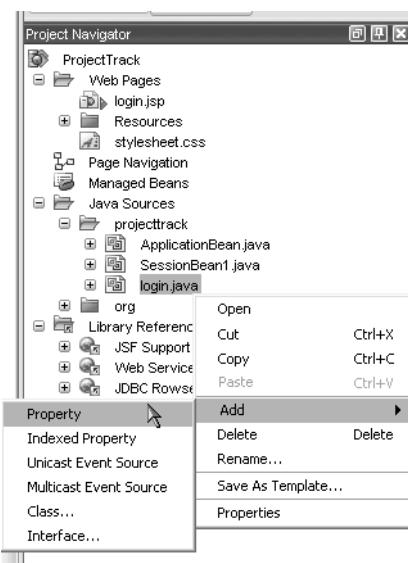


Figure B.71
Adding a new property to
the backing bean.

bean. For Command components, the default event handler is an action method, and for Input components, it is a value-change listener method. You can also access an event handler by right-clicking on a component, as shown in figure B.76.

This displays the code editor inside of the Editing Area. Here, we can fill in the logic for the login action method, which is shown in listing B.3.

Listing B.3 ProjectTrack's login method inside Java Studio Creator

```

        "Error loading User object", d);
        return Constants.ERROR_OUTCOME;
    }

Visit visit = new Visit();
visit.setUser(newUser);

setBean(Constants.VISIT_KEY_SCOPE +
          Constants.VISIT_KEY, visit);

if (newUser.getRole().equals(RoleType.UPPER_MANAGER))
{
    return Constants.SUCCESS_READONLY_OUTCOME;
}

return Constants.SUCCESS_READWRITE_OUTCOME;
}

```

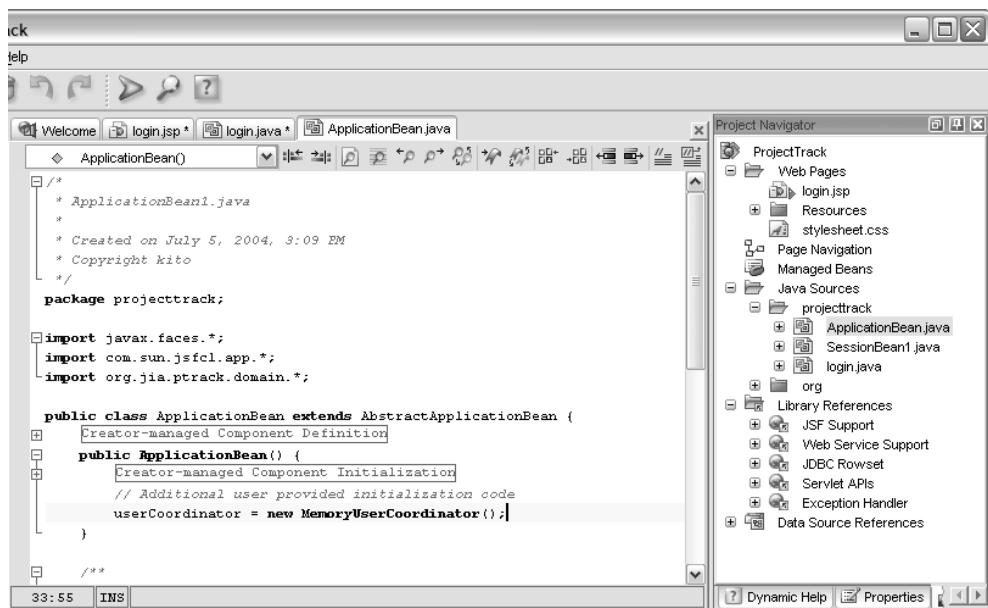
The methods marked in bold are implemented by the backing bean's super-class; note that they provide much of the same functionality as ProjectTrack's BaseBean class.

Adding navigation

Creator has a nice visual editor for the application's navigation rules. You can access the page navigation editor through the Project Navigator, as shown in figure B.77.



Figure B.72
The New Property dialog box.



```

/*
 * ApplicationBean.java
 *
 * Created on July 5, 2004, 3:09 PM
 * Copyright kito
 */
package projecttrack;

import javax.faces.*;
import com.sun.jsfcl.app.*;
import org.jia.ptrack.domain.*;

public class ApplicationBean extends AbstractApplicationBean {
    Creator-managed Component Definition
    public ApplicationBean() {
        Creator-managed Component Initialization
        // Additional user provided initialization code
        userCoordinator = new MemoryUserCoordinator();
    }
}

```

Figure B.73 Creating a new `MemoryUserCoordinator` in `ApplicationBean`'s constructor.

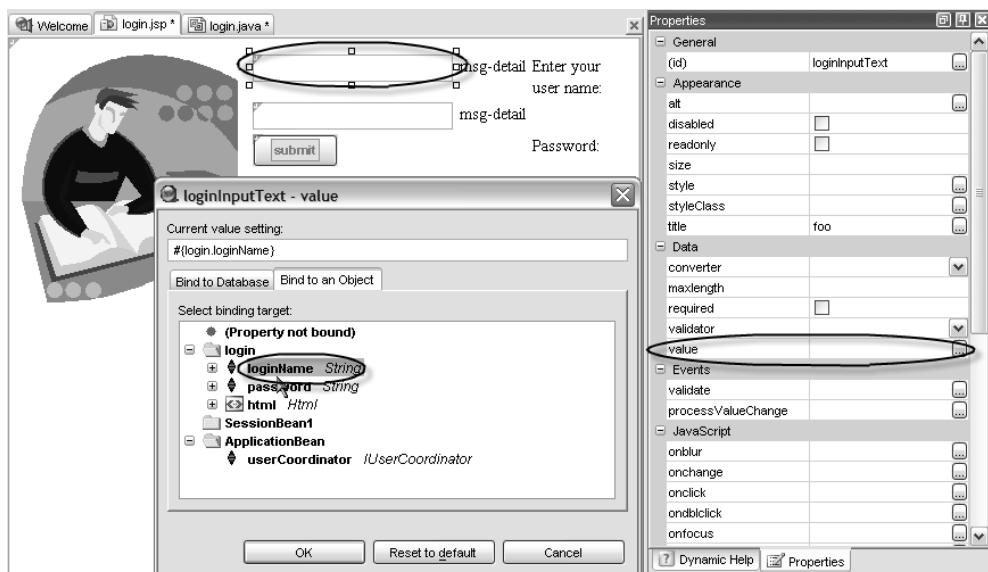


Figure B.74 Binding the button's value to a backing bean property.

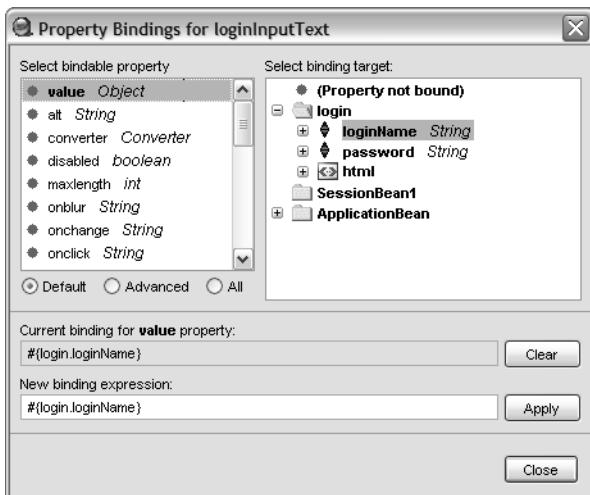


Figure B.75
The Property Bindings dialog box
allows you to visually create a
value-binding for any value-binding
enabled property.

This displays a visual representation of the navigation rules in the Editing Area. When you click on a page inside the editor, all of the page's action sources are displayed as shown in figure B.78.

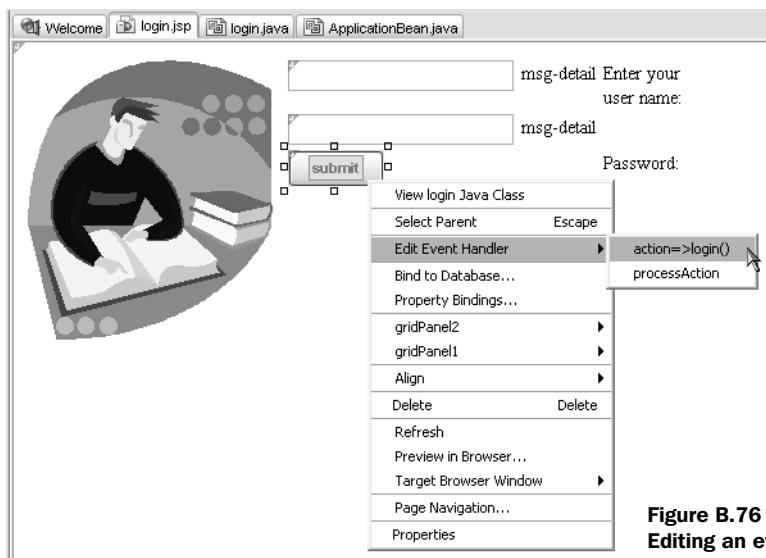


Figure B.76
Editing an event handler.

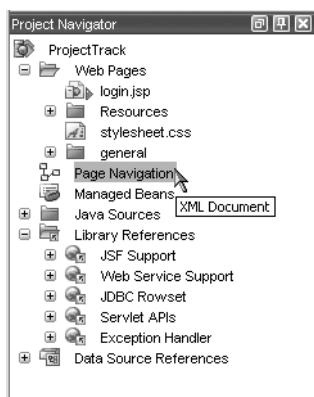


Figure B.77
Selecting page navigation
in the Project Navigator.

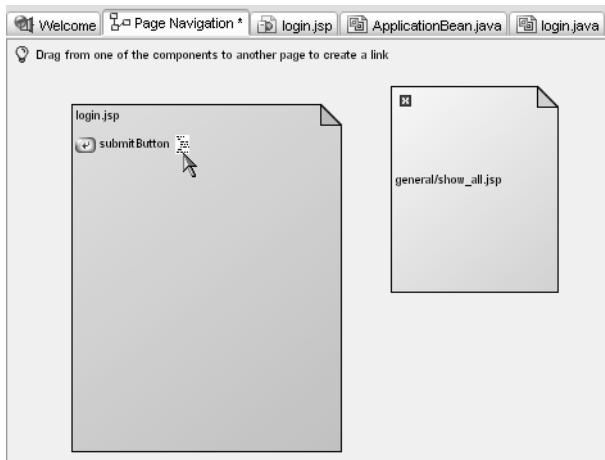


Figure B.78
Clicking on a page in the page
navigation editor displays all
of its action sources.

After you've selected the proper action source, you can draw a line to the next page, as shown in figure B.79.

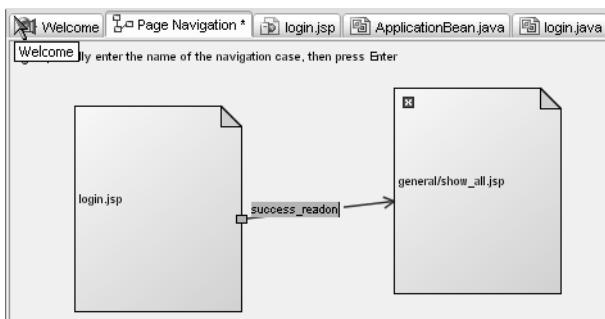


Figure B.79
Drawing a navigation case
between two pages.

Once you've drawn the line, you can fill in its caption, which is the logical outcome for the navigation case. This generates the following navigation rule:

```
<faces-config>
    <navigation-rule>
        <from-view-id>/login.jsp</from-view-id>
        <navigation-case>
            <from-outcome>success_READONLY</from-outcome>
            <to-view-id>/general/show_all.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Testing

Our Login page is now complete. To get a rough idea of how it will appear, choose the Preview in a Browser option that's available when you right-click on the page in design mode. This launches a browser with a static HTML mock-up. You can also launch the application by clicking the Run Project toolbar button, as shown in figure B.80.

Because Creator is bundled with the J2EE SDK, there's no need to configure servers—you can test your application as soon as it's finished. Figure B.81 shows our completed Login page in a browser.

B.3.3 Wrapping up

Sun's Java Studio Creator was designed first and foremost for ease of use. Its simplified development model—based solely on JSF—speeds up the development process by limiting the number of choices. Creator offers an attractive, intuitive, drag-and-drop development environment that should be familiar to users of Visual Studio .NET. In addition, its ability to work natively with web services and relational databases makes it a one-stop shop for building simpler applications. However, Creator alone isn't the tool for heavy-duty J2EE applications or integration with multiple Java technologies.

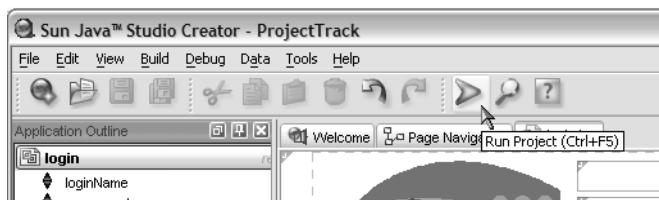


Figure B.80
Running the application.



Figure B.81
The Login page
shown in a browser.

B.4 JSF implementations

Like most Java standards, there are several different JSF implementations. Table B.1 lists a few of the implementations currently on the market.

Table B.1 Several JSF implementations are currently available.

Implementation	URL	Open Source?	Description
Reference Implementation	http://java.sun.com/j2ee/ javaserverfaces	No	The official reference implementation. The source code is now available under the Java Research License [Sun, JRL], which allows for modification and use of the source for educational purposes. Version 1.1 is a maintenance release that fixes several bugs, and is backward compatible with 1.0.
MyFaces	http://www.myfaces.org	Yes	MyFaces is the first free open source implementation of JavaServer Faces. It features extensions to standard components and has many simple-to-use custom components (such as Calendar, FileUpload, or Tabbed Pane), as well as custom validators (credit card, email, and so on). As of this writing, MyFaces is in the process of moving to Apache, so this URL could change.

continued on next page

Table B.1 Several JSF implementations are currently available. (continued)

Implementation	URL	Open Source?	Description
Smile	http://smile.sourceforge.net	Yes	An open source implementation of the JavaServer Faces API. Special attention will be given to the non-JSP programming model. The main goals are specifications compliance, a rich set of GUI controls, and a designer application for creating JSF pages.
Simplica ECruiser	http://www.simplica.com/overview.html	No	Simplica's ECruiser is an implementation of the basic JSF runtime environment and a library of advanced components.
Keel	http://www.keelframework.org	Yes	The Keel Meta-Framework now includes an early implementation of JavaServer Faces that integrates with Cocoon. Keel is a highly extensible backbone for integrating Java projects that's fleshed out and ready to use.

This table includes only existing or announced implementations at the time this book was written. Some of the major vendors, such as IBM and Oracle, will be enhancing the RI with specific optimizations, and you can expect other implementations to appear as the industry matures. You can find an up-to-date list at JSF Central [JSF Central].



*Extending the
core JSF classes*

As we've stated previously, one of JSF's most compelling benefits is its pluggable architecture. Most of its features are handled by classes that can easily be replaced with alternate implementations, which can either enhance or replace the default behavior. This is how you can use display technologies other than JSP (see appendix A), but it also allows you to change the way JSF handles expressions, navigation, state management, the default action listener, and more.

In this appendix, we examine JSF's architecture and discuss how to extend its core features.

BY THE WAY

If you're wondering why we placed coverage of these classes in an appendix instead of a chapter, there's a simple reason: most developers will be happy with the defaults, and don't need to worry about the internals. However, if you're doing advanced JSF development, you're interested in JSF's architecture, or you simply need to customize an implementation in some way, this appendix is for you.

C.1 **The classes behind the scenes**

The core JSF classes can be broken into two categories: infrastructure and pluggable. The infrastructure classes are `Lifecycle`, `RenderKit`, `Application`, and `FacesContext`. `Lifecycle` is used to execute the Request Processing Lifecycle. `RenderKit` is used to manage a set of renderers, and `Application` maintains references to instances of pluggable classes and provides access to additional configuration information. Finally, there is the venerable `FacesContext`, which handles all per-request functionality.

All of these classes are created by factories, which are configurable in an application configuration file, and can be located with the `FactoryFinder` (which is a singleton). These classes are listed in table C.1.

Table C.1 The infrastructure classes form the backbone of JSF's functionality.

Class	Description	Configurable?
<code>javax.faces.FactoryFinder</code>	Creates and stores factory instances. The concrete factory implementations are found by searching available configuration files or retrieved from the first line of a file called <code>META-INF/services/{factory-class-name}</code> (located in the web application itself or any JARs in its library path).	No
<code>javax.faces.lifecycle.LifecycleFactory</code>	Creates and stores <code>Lifecycle</code> instances.	Yes

continued on next page

Table C.1 The infrastructure classes form the backbone of JSF's functionality. (continued)

Class	Description	Configurable?
<code>javax.faces.lifecycle.Lifecycle</code>	Runs the Request Processing Lifecycle. Usually functionality is added through <code>PhaseListeners</code> , which can either be registered in code or in a configuration file.	No
<code>javax.faces.render.RenderKitFactory</code>	Manages a collection of <code>RenderKit</code> instances.	Yes
<code>javax.faces.render.RenderKit</code>	Manages a collection of <code>Renderer</code> instances. Contains a reference to a <code>ResponseStateManager</code> instance, which handles renderer-specific state-management work.	Yes
<code>javax.faces.application.ApplicationFactory</code>	Creates and stores a single <code>Application</code> instance (per web application).	Yes
<code>javax.faces.application.Application</code>	Holds references to instances of pluggable classes and configuration information.	Yes
<code>javax.faces.context.FacesContextFactory</code>	Creates a new (or returns a pooled) <code>FacesContext</code> instance.	Yes
<code>javax.faces.context.FacesContext</code>	Manages all per-request state, including the message queue.	No

Note that the concrete subclasses of the factories (as well as `RenderKit` subclasses) are configurable through an application configuration file. These classes, and their relationships, are shown in figure C.1.

For most developers, there is no need to create customized `Lifecycle`, `Application`, `RenderKit`, or `FacesContext` instances, and consequently no need to customize their factory classes. The ability to replace the factories is useful, however, for developers of JSF implementations.

The only exception to this rule is the `RenderKit` class; you can configure one or more `RenderKit` subclasses in addition (or instead of) the standard `HTML RenderKit`. Note, however, that `RenderKitFactory` only stores `RenderKit` instances—it doesn't create them. Usually, that responsibility is left for the JSF implementation's configuration loader, which creates new `RenderKit` instances based on any application configuration files it finds. The upshot is that while you might write your own `RenderKit`, you would usually not need to implement a new `RenderKitFactory` because it just maintains `RenderKit` instances.

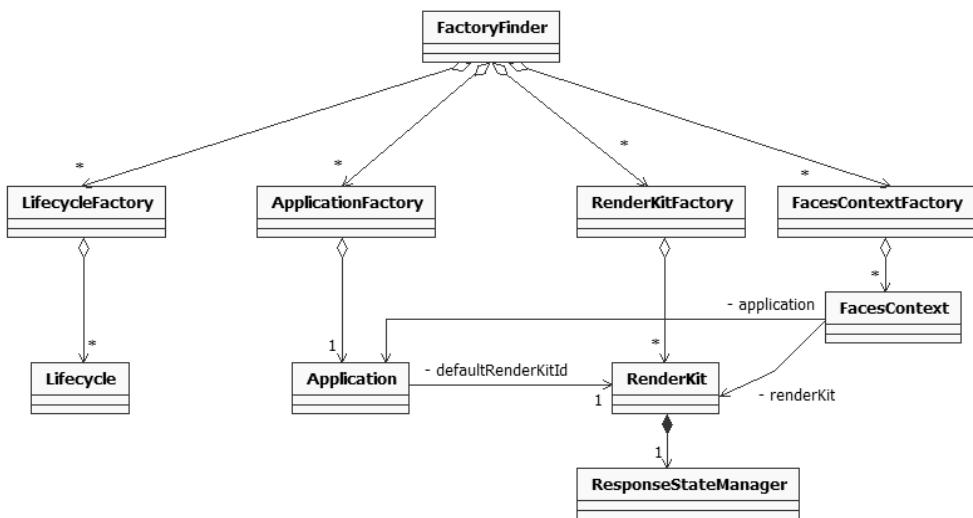


Figure C.1 JSF's infrastructure classes consist of the objects that execute JSF's core processing (`Lifecycle`, `Application`, `RenderKit`, and `FacesContext`) and factories that manage instances of them.

NOTE If you write your own `RenderKit`, you may also have to write a new `ResponseStateManager` to handle saving and restoring state on the client.

You normally wouldn't replace the `LifecycleFactory` implementation, but you can integrate with the JSF lifecycle by writing `PhaseListener` classes, which are executed before and after specific Request Processing Lifecycle phases (see chapter 11 for more information). `PhaseListeners` can be registered in code or in a JSF configuration file.

In general, it's more likely that you (or third parties) will replace the pluggable classes. This is, after all, why the JSF Expert Group made them pluggable in the first place. The classes in this category are `VariableResolver`, `PropertyResolver`, `ActionListener`, `NavigationHandler`, `ViewHandler`, and `StateManager`. These are all abstract base classes, except for `ActionListener`, which is an interface.

The `VariableResolver` is responsible for evaluating the left identifier of a JSF EL expression, and the `PropertyResolver` evaluates identifiers after the `"."` or inside the `"[]"`. So, for the expression `"#{foo.bar}"`, the `VariableResolver` returns the object referenced by `foo`, and the `PropertyResolver` returns the value of the `foo`'s `bar` property. The expression `"#{foo[bar].baz}"` would be evaluated the same, except that the `PropertyResolver` would evaluate `bar`'s `baz` property as well.

The default instance of `VariableResolver` looks for an object stored in different application scopes under the specified key and attempts to create it with the Managed Bean Creation facility if it doesn't already exist. The default `PropertyResolver` searches for properties, `List` items, or `Map` values on the object retrieved by the `VariableResolver`. (This is the standard JSF EL processing we've discussed throughout this book; see chapter 2 for details.)

The pluggable `ActionListener` handles all action events (regardless of any additional action listeners that are registered by the application). The default instance performs the behavior we have discussed so far: it executes an action method, retrieves its logical outcome, and executes the `NavigationHandler` with that outcome.

The `NavigationHandler`'s job is to select a new view based on the current view, the calling action method, and the logical outcome. The default `NavigationHandler` instance selects a new view identifier based on the outcome and any configured navigation rules, uses the `ViewHandler` to create the new view, and sets it as the current view on the `FacesContext`. The selected view is then displayed by the `ViewHandler` during the Render Response phase of the Request Processing Lifecycle. The default `ViewHandler` handles this by forwarding the request to the web container, which processes the JSP page normally.

The `ViewHandler` is also responsible for restoring views when they are requested more than once. It delegates state-saving and restoration to the `StateManager`. The default `StateManager` implementation either saves the component tree in the session if the state-saving mode is `server` or saves it as a hidden form field if the state-saving mode is `client`. State-saving work for the `client` mode is delegated to the `RenderKit`'s `ResponseStateManager`, which saves and restores the state in a renderer-specific manner.

All of these classes are listed in table C.2, and depicted graphically in figure C.2.

Table C.2 The pluggable classes perform key JSF functions. The default implementations can be replaced in an application configuration file (usually `faces-config.xml`).

Class	Description	Default Functionality
<code>javax.faces.el.VariableResolver</code>	Evaluates the leftmost identifier of a JSF EL expression.	Searches for an object in request, session, or application scope with the specified key. If the object does not exist, attempts to create it through the Managed Bean Creation facility.

continued on next page

Table C.2 The pluggable classes perform key JSF functions. The default implementations can be replaced in an application configuration file (usually faces-config.xml). (continued)

Class	Description	Default Functionality
<code>javax.faces.el.PropertyResolver</code>	Evaluates the identifiers of a JSF EL expression after “.” or inside “[]”.	Returns the property, array element, List element, or Map element.
<code>javax.faces.application.ActionListener</code>	Executed for all action events during the Invoke Application phase.	Calls the associated action method (if any), and then executes the NavigationHandler with the retrieved outcome.
<code>javax.faces.navigation.NavigationHandler</code>	Selects a view based on the current view, the action method (if any), and an outcome.	Selects the view identifier based on configured navigation rules, asks the ViewHandler to create the view, and then sets it as the current view on the FacesContext.
<code>javax.faces.application.ViewHandler</code>	Creates, displays, and restores views. Delegates to StateManager to save and restore view state.	Performs basic functionality of creating views and delegating to StateManager, but relies on the JSP container to build and display the component tree. (The tree is built by the JSF component tags.)
<code>javax.faces.applicationStateManager</code>	Stores and restores the view.	If the state-saving mode is server, stores the view in the session; if it is client, delegates to the current RenderKit's ResponseStateManager.

Now that you know which classes perform JSF’s key functionality, let’s look at how you can modify that functionality.

C.2 Replacing or extending the pluggable classes

Most of the time, there’s no need to replace the default implementations of the pluggable classes. However, doing so offers some powerful possibilities and may be necessary, especially if you’re integrating JSF with another framework. Table C.3 lists some possible reasons you may want to replace or augment the default implementations of these classes.

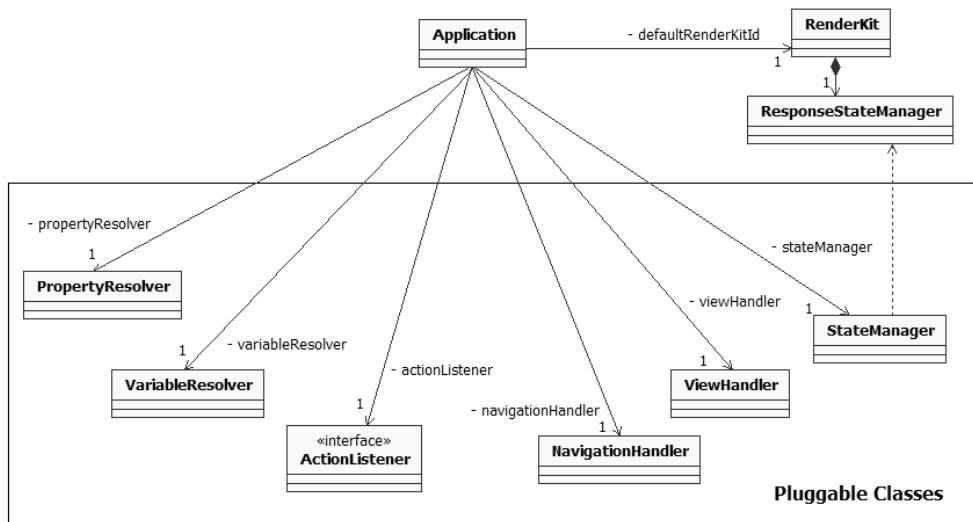


Figure C.2 The `Application` class has references to instances of pluggable classes. These classes perform much of JSF's processing. Each of them can be replaced with alternate implementations.

Table C.3 The pluggable classes perform key JSF functions. In some cases, you may want to replace or decorate them.

Class	Possible Reasons to Replace or Decorate
VariableResolver	<ul style="list-style-type: none"> To add your own implicit variables To provide shorthand or alternate names for commonly used variable names To access variables in other contexts (web applications, web services, EJB servers, and so on) To add a default property
PropertyResolver	<ul style="list-style-type: none"> To provide shorthand or alternate names for commonly used properties To access properties of variables in other contexts (web applications, web services, EJB servers, and so on) To retrieve other attributes of a variable, like methods or other nonproperty attributes To access attributes of specialized objects (like a DOM tree)
ActionListener	<ul style="list-style-type: none"> To add default behavior that is executed for all action events, like logging To execute action methods with different signatures To execute application logic from a different framework (like Struts Actions)
NavigationHandler	<ul style="list-style-type: none"> To select a view identifier based on criteria other than navigation rules To forward control to another resource (bypassing the <code>ViewHandler</code>)

continued on next page

Table C.3 The pluggable classes perform key JSF functions. In some cases, you may want to replace or decorate them. (continued)

Class	Possible Reasons to Replace or Decorate
ViewHandler	<ul style="list-style-type: none"> • To use a non-JSP display technology (such as XUL [XUL], Velocity [ASF, Velocity], or Java classes) • To decorate the view
StateManager	<ul style="list-style-type: none"> • To store state in another manner (such as a data store) • To optimize the state-saving algorithms • To perform customized clustering

Now that it's clear when you may want to replace or decorate a pluggable class, let's examine the process.

C.2.1 Configuring a pluggable class

Replacing the default implementation of any of these pluggable classes is handled in an application configuration file. Every class listed in table C.3 has a corresponding XML element that is a child of the `<application>` node. For example, here is a configuration file snippet that configures a custom action listener and state manager:

```

<application>
    <action-listener>
        com.foo.jsf.MyCustomActionListener
    </action-listener>
    <state-manager>
        com.foo.jsf.MyCustomStateManager
    </state-handler>
</application>

```

Remember, each JSF application can have multiple configuration files (often located in JARs that may contain UI components, renderers, or other extensions). So, if more than one file replaces a pluggable class, how does JSF know which one to use? Recall that JSF searches for the following configuration files in this order:

- 1 Files named `META-INF/faces-config.xml` in the resource path (including JARs)
- 2 Files specified by the `javax.faces.CONFIG_FILES` servlet initialization parameter
- 3 The `WEB-INF/faces-config.xml` file (in the current web application)

Each time the implementation reads a configuration file, it will register any pluggable classes that have been configured, so the most recently parsed file always wins.

For example, let's say there was a custom `ViewHandler` configured in the `META-INF/faces-config.xml` file of a specific JAR, the `WEB-INF/trading_module.xml` file of the current web application, and the `WEB-INF/faces_config.xml` file of the current application. Each `ViewHandler` would be configured but overwritten by the next file that was processed. Consequently, the `ViewHandler` available to the application would be the one defined in `WEB-INF/faces-config.xml`.

If you're simply replacing a pluggable class (rather than extending it), this is all you need to know: the last file processed takes precedence. However, if you want to extend a class, the processing order becomes important; we discuss this next.

C.2.2 Decorating a pluggable class

It's never fun to reinvent the wheel, so JSF allows you to add functionality to the previous pluggable class implementation rather than requiring you to write default logic. This is possible thanks to the decorator pattern [GoF], which allows you to extend a class but delegate most of the work to an existing instance. The word *previous* is key here, because it refers to the last implementation registered. In the scenario we discussed at the end of the preceding section, there would be three distinct `ViewHandler` instances, each of which could add functionality to the one loaded before it. This allows you to chain multiple pluggable classes together, which is important in cases where multiple parties want to customize JSF's functionality.

An example of such a situation is the Struts-Faces integration library [ASF, Struts-Faces], which registers a custom `ActionListener` and `PropertyResolver`. If you develop an application that uses this library, you can still add your own custom implementations of these classes. Assuming you decorated the Struts-Faces functionality, your application would be able to take advantage of the extensions you made in addition to the ones made by Struts-Faces.

NOTE We recommend that you decorate existing classes rather than write entirely new ones, unless you know exactly what you're doing. Otherwise, you may break some existing functionality.

Decorating a pluggable class is straightforward (the process is the same for factories as well). The only requirement is that you have a constructor that takes an instance of the pluggable class as a parameter. JSF will pass in the previously registered instance—all you have to do is execute its methods at the appropriate times. Here's an example:

```

public class MyCustomActionListener implements ActionListener
{
    private ActionListener previous;

    public ActionListenerImpl(ActionListener previous)
    {
        this.previous = previous;
    }

    public void processAction(ActionEvent event)
            throws AbortProcessingException
    {
        // Perform custom processing here
        previous.processAction(event);
    }
}

```

The example shows a custom `ActionListener` that delegates additional processing to the previously registered `ActionListener` after performing some custom processing. In cases where several methods must be implemented, you may decide to delegate directly for some (and add no processing) and either override completely or add custom processing to others.

Now that you understand how to decorate a pluggable class, let's look at a concrete example.

Example: Struts-Faces `ActionListener`

A good example of extending JSF is the Struts-Faces integration library [ASF, Struts-Faces]. This library makes good use of pluggable JSF classes, providing a customized `ActionListener` (for integrating Struts Actions) and `PropertyResolver` (for adding DynaBean support). Let's take a look at the custom `ActionListener`, which delegates processing to Struts if the form that submitted the request was a `FormComponent` instance (which is a Struts-Faces custom component). The source is shown in listing C.1.

Listing C.1 The Struts-Faces custom `ActionListener` implementation (portions omitted for simplicity)

```

package org.apache.struts.faces.application;

import javax.faces.component.*;
import javax.faces.context.FacesContext;
import javax.faces.event.*;
import javax.servlet.ServletContext;
import javax.servlet.http.*;

```

```
import org.apache.struts.Globals;
import org.apache.struts.action.*;
import org.apache.struts.config.ModuleConfig;
import org.apache.struts.faces.Constants;
import org.apache.struts.faces.component.FormComponent;
import org.apache.struts.util.RequestUtils;

public final class ActionListenerImpl implements ActionListener
{

    private ActionListener original;

    public ActionListenerImpl(
        ActionListener original)    | ① Takes
    {                                delegate as
        if (original == null)           argument
        {
            throw new NullPointerException();
        }
        this.original = original;    ← ② Stores
    }                                delegate
                                         instance

    public void processAction(ActionEvent event)    | ③ Overrides
        throws AbortProcessingException          primary
    {

        // If this is an immediate action, or we are NOT nested in a
        // Struts form, perform the standard processing
        UIComponent component = event.getComponent();
        ActionSource source = (ActionSource)component;
        boolean standard = source.isImmediate();
        if (!standard)
        {
            UIComponent parent =
                component.getParent();
            while (parent != null)
            {
                if (parent instanceof UIForm)
                {
                    if (!(parent instanceof
                        FormComponent))
                    {
                        standard = true;
                    }
                    break;
                }
                parent = parent.getParent();
            }
        }
    }
}
```

① Takes delegate as argument

② Stores delegate instance

③ Overrides primary method

④ Determines if standard request

```

if (standard)
{
    original.processAction(event);
    return;
}

// Acquire Servlet API Object References
FacesContext context = FacesContext.getCurrentInstance();
ServletContext servletContext = (ServletContext)
    context.getExternalContext().getContext();
HttpServletRequest request = (HttpServletRequest)
    context.getExternalContext().getRequest();
HttpServletResponse response = (HttpServletResponse)
    context.getExternalContext().getResponse();

// Invoke the appropriate request processor for this request
try
{
    request.setAttribute(
        Constants.ACTION_EVENT_KEY, event);
    RequestUtils.selectModule(request,
        servletContext);
    ModuleConfig moduleConfig =
        (ModuleConfig)request.getAttribute(
            Globals.MODULE_KEY);
    RequestProcessor processor =
        getRequestProcessor(moduleConfig,
            servletContext);
    processor.process(request, response);
    context.responseComplete();
}
catch (Exception e)
{
    // log the exception
}
finally
{
    request.removeAttribute(Constants.ACTION_EVENT_KEY);
}

}

// Protected Methods
protected RequestProcessor getRequestProcessor(ModuleConfig config,
                                                ServletContext context)
{

String key = Globals.REQUEST_PROCESSOR_KEY + config.getPrefix();
RequestProcessor processor =
    (RequestProcessor)context.getAttribute(key);
...

```

**5 If standard,
delegates to
original instance**

**6 Otherwise,
forwards to
Struts
processor**

```

        return (processor);
    }
}

```

This class's goal is to integrate Struts processing with JSF. In order to delegate control to the previous `ActionListener` instance, the constructor has a single `ActionListener` argument (❶), which we store for later use (❷).

The work takes place in the `processAction` method (❸). The first step is to determine whether this is a standard JSF request, or if it should be processed by Struts. Recall from chapter 14 that any page that uses the Struts-Faces tags must use the `<s:form>` component tag. This tag represents the Struts-Faces `FormComponent` class, which has special support for Struts `ActionForms` and `Actions`.

This is a standard JSF request if the UI component that fired the event has its `immediate` property set to `true`, or if it is *not* the child of a `FormComponent` (❹). If this is a standard request, we delegate processing to the original `ActionListener` (❺), which results in normal JSF processing. Otherwise, we delegate processing to the Struts `RequestProcessor`, which handles normal Struts requests (❻). This ensures that the Struts Action associated with the `FormComponent` is executed instead of a JSF action method.

This class is registered in the Struts-Faces configuration file, which is located in the `META-INF` directory of its JAR file. Here's the relevant snippet:

```

<application>
    <action-listener>
        org.apache.struts.faces.application.ActionListenerImpl
    </action-listener>
</application>

```

That's it for our example of decorating a pluggable class. As you can see, it's relatively easy to augment JSF's existing functionality as necessary. Now, let's look at an example that replaces a class's functionality altogether.

C.2.3 Replacing a pluggable class

Most of the time, you will decorate a pluggable class instead of simply replacing its functionality. However, in some cases, such as supporting alternate display technologies, it makes sense to write a class that does not depend on the default implementation. The process is the same as decorating a pluggable class, except that your new class should have a no-argument constructor instead of a constructor that takes the pluggable type as a parameter. In the next section, we provide an example of this process.

Example: XUL ViewHandler

In appendix A, we covered the XML User Interface Language (XUL) [XUL] example included with the JSF reference implementation [Sun, JSF RI]. This example uses XUL (a powerful language for describing user interfaces) to define views instead of JSP. Let's take a look at how this functionality was implemented.

Replacing JSP with an alternative display technology like XUL is as simple as replacing the `ViewHandler`. Table C.4 lists the `ViewHandler` methods, all of which must be overridden by subclasses.

Table C.4 `ViewHandler` methods (all must be overridden by subclasses)

Method	Description
<code>createView</code>	Creates a new view based on a view identifier
<code>renderView</code>	Displays the currently selected view
<code>restoreView</code>	Restores a previously requested view, possibly delegating to the <code>StateManager</code>
<code>getActionURL</code>	Translates a view identifier into a full URL
<code>getResourceURL</code>	Translates a resource path into a full URL
<code>writeState</code>	Writes out state (usually delegating to the <code>StateManager</code>)
<code>calculateLocale</code>	Determines the user's locale for this request (and subsequent requests)
<code>calcluateRenderKit</code>	Returns the correct <code>RenderKit</code> for this request (and subsequent requests)

Listing C.2 shows the source for `XulViewHandlerImpl`, a `ViewHandler` replacement that creates the view from an XUL file.

NOTE

This example is meant to illustrate how to write a `ViewHandler`, but it is not a definitive, production-ready implementation. Consequently, it uses the default `RenderKit`, hardcodes the character encoding, and doesn't handle state management. These are all issues that a production-quality `ViewHandler` must address.

Listing C.2 A `ViewHandler` replacement that parses XUL files instead of JSP (parts have been omitted for simplicity)

```
package nonjsp.application;

import nonjsp.util.RIConstants;
```

```
import org.apache.commons.digester.*;  
  
import javax.faces.*;  
import javax.faces.application.ViewHandler;  
import javax.faces.component.*;  
import javax.faces.context.*;  
import javax.faces.render.*;  
import javax.servlet.*;  
import javax.servlet.http.HttpServletResponse;  
  
import java.io.*;  
import java.net.URL;  
import java.util.*;  
  
public class XulViewHandlerImpl extends ViewHandler  
{  
    // Log instance for this class  
    protected static Log log =  
        LogFactory.getLog(XulViewHandlerImpl.class);  
    protected static final String CHAR_ENCODING = "ISO-8859-1";  
    protected static final String CONTENT_TYPE = "text/html";  
  
    protected boolean validate = false;  
    ...  
    protected XmlDialectProvider dialectProvider = null;  
  
    public XulViewHandlerImpl() ←①  
    {  
        super();  
        dialectProvider = new XulDialectProvider();  
    }  
  
    // Render the components  
    public void renderView(FacesContext context,  
                           UIViewRoot viewToRender) →②  
        throws IOException, FacesException  
    {  
        if (context == null || viewToRender == null)  
        {  
            throw new NullPointerException(  
                "RenderView: FacesContext is null");  
        }  
  
        HttpServletResponse response = (HttpServletResponse)  
            (context.getExternalContext().getResponse());  
  
        RenderKitFactory factory = (RenderKitFactory)  
            FactoryFinder.getFactory(FactoryFinder.RENDER_KIT_FACTORY);  
  
        RenderKit renderKit = factory.getRenderKit(context,  
            calculateRenderKitId(context));
```

```
ResponseWriter writer = renderKit.createResponseWriter(
    response.getWriter(), CONTENT_TYPE,
    CHAR_ENCODING);

context.setResponseWriter(writer);
response.setContentType(CONTENT_TYPE);

createHeader(context);
renderResponse(context);
createFooter(context);

Map sessionMap = getSessionMap(context);
sessionMap.put(RIConstants.REQUEST_LOCALE,
    context.getViewRoot().getLocale());
sessionMap.put(RIConstants.FACES_VIEW, context.getViewRoot());
}

// Create the header components for this page
private void createHeader(FacesContext context) throws IOException { ③
{
    ResponseWriter writer = context.getResponseWriter();

    writer.startElement("html", null);
    writer.writeText("\n", null);
    writer.startElement("head", null);
    writer.writeText("\n", null);
    writer.startElement("title", null);
    writer.writeText(
        context.getExternalContext().getRequestContextPath(),
        null);
    writer.endElement("title");
    writer.writeText("\n", null);
    writer.endElement("head");
    writer.writeText("\n", null);
    writer.startElement("body", null);
    writer.writeText("\n", null);
}

// Create the footer components for this page
private void createFooter(FacesContext context) throws IOException { ③
{
    ResponseWriter writer = context.getResponseWriter();

    writer.endElement("body");
    writer.writeText("\n", null);
    writer.endElement("html");
    writer.writeText("\n", null);
}
```

```

// Render the response content for the completed page
private void renderResponse(FacesContext context) throws IOException { ④
{
    UIComponent root = context.getViewRoot();
    renderResponse(context, root);
}

// Render the response content for an individual component
private void renderResponse(
    FacesContext context,
    UIComponent component)
throws IOException { ④
{
    component.encodeBegin(context);
    if (component.getRendersChildren())
    {
        component.encodeChildren(context);
    }
    else
    {
        Iterator kids = component.getChildren().iterator();
        while (kids.hasNext())
        {
            renderResponse(context, (UIComponent)kids.next());
        }
    }
    component.encodeEnd(context);
}

public UIViewRoot createView(
    FacesContext context,
    String viewId) { ⑤
{
    if (context == null)
    {
        throw new NullPointerException(
            "CreateView: FacesContext is null");
    }

    return restoreView(context, viewId);
}

public UIViewRoot restoreView(
    FacesContext context,
    String viewId) { ⑤
{
    if (context == null)
    {

```

```
        throw new NullPointerException(
            "RestoreView: FacesContext is null");
    }

    UIViewRoot root = null;
    InputStream viewInput = null;
    RuleSetBase ruleSet = null;

    root = new UIViewRoot();
    root.setRenderKitId(calculateRenderKit(context));

    if (null == viewId)
    {
        root.setViewId("default");
        context.setViewRoot(root);
        Locale locale = calculateLocale(context);
        root.setLocale(locale);
        return root;
    }

    try
    {
        viewInput =
            context.getExternalContext().getResourceAsStream(viewId);
        if (null == viewInput)
        {
            throw new NullPointerException();
        }
    }
    catch (Throwable e)
    {
        throw new FacesException("Can't get stream for " + viewId, e);
    }

    Digester digester = new Digester();
    ...
    ruleSet = dialectProvider.getRuleSet();
    digester.addRuleSet(ruleSet);
    ...
    digester.push(root);
    try
    {
        root = (UIViewRoot)digester.parse(viewInput);
    }
    catch (Throwable e)
    {
        throw new FacesException(
            "Can't parse stream for " + viewId, e);
    }

    root.setViewId(viewId);
```

```

        context.setViewRoot(root);

        return root;
    }

    public Locale calculateLocale(FacesContext context) ←⑥
    {
        Locale result = null;
        // determine the locales that are acceptable to the client based on
        // the Accept-Language header and the find the best match among the
        // supported locales specified by the client.
        Enumeration enum = ((ServletRequest)
            context.getExternalContext().getRequest()).getLocales();
        while (enum.hasMoreElements())
        {
            Locale perf = (Locale)enum.nextElement();
            result = findMatch(context, perf);
            if (result != null)
            {
                break;
            }
        }
        // no match is found.
        if (result == null)
        {
            if (context.getApplication().getDefaultLocale() == null)
            {
                result = Locale.getDefault();
            }
            else
            {
                result = context.getApplication().getDefaultLocale();
            }
        }
        return result;
    }

    /**
     * Attempts to find a matching locale based on <code>perf</code>
     * and list of supported locales, using the matching algorithm
     * as described in JSTL 8.3.2.
     */
    protected Locale findMatch(FacesContext context, Locale perf) ←⑥
    {
        Locale result = null;
        Iterator it = context.getApplication().getSupportedLocales();
        while (it.hasNext())
        {
            Locale supportedLocale = (Locale)it.next();
            if (perf.equals(supportedLocale))

```

```

    {
        // exact match
        result = supportedLocale;
        break;
    }
    else
    {
        // Make sure the preferred locale doesn't have a country set,
        // when doing a language match, For ex., if the preferred
        // locale is "en-US", if one of supported locales is "en-UK",
        // even though its language matches that of the preferred
        // locale, we must ignore it.
        if (perf.getLanguage().equals(supportedLocale.getLanguage()) &&
            supportedLocale.getCountry().equals(""))
        {
            result = supportedLocale;
        }
    }
    return result;
}

public String calculateRenderKitId(
    FacesContext context) | 7
{
    return RenderKitFactory.HTML_BASIC_RENDER_KIT;
}

public String getActionURL(
    FacesContext context,
    String viewId) | 8
{
    if (viewId.charAt(0) != '/')
    {
        throw new IllegalArgumentException(
            "Illegal view ID " + viewId + ". the ID must begin with ' /'");
    }
    if (!viewId.startsWith("/faces"))
    {
        viewId = "/faces" + viewId;
    }
    return context.getExternalContext().getRequestContextPath() + viewId;
}

public String getResourceURL(
    FacesContext context,
    String path) | 9
{
    if (path.startsWith("/"))
    {

```

```
        return context.getExternalContext().  
               getRequestContextPath() + path;  
    }  
    else  
    {  
        return (path);  
    }  
}  
  
public void writeState(FacesContext context) ← 10  
    throws IOException  
{  
}  
  
private Map getSessionMap(FacesContext context)  
{  
    // retrieve session map, creating session if necessary  
    ...  
    return sessionMap;  
}  
}
```

-
- ➊ Because we aren't decorating the previous `ViewHandler`, we have a no-argument constructor. (The `dialectProvider` instance variable creates rules that will be used to process the XUL files in ➋.)
 - ➋ The `renderView` method is responsible for displaying the current view to the user. The process is simple: retrieve a new `ResponseWriter` from the current `RenderKit`, use the `ResponseWriter` to render the view, and then place any necessary variables in the session. The real work of displaying the view is performed by `createHeader`, `createFooter` (➌) and `renderResponse` (➍).
 - ➌ The `createHeader` and `createFooter` methods simply output the HTML before and after the actual view (note that the last parameter of `startElement` and `writeText` is `null` because this output is not associated with a specific component).

BY THE WAY

The fact that we're outputting the beginning and end of the HTML document underscores an important point: JSF view definitions do not require hardcoded HTML template text (as is often included in JSPs).

- ➍ These two `renderResponse` methods are responsible for displaying the components in the view. Again the process is simple: for each component, call `encodeBegin`, call `encodeChildren` (if `rendersChildren` is `true`) or render each child individually (if `rendersChildren` is `false`), and then call `encodeEnd`. Recall that each UI component will delegate this work to the associated renderer if necessary.

- ⑤ The `createView` method simply calls `restoreView`, which actually builds the component tree from the XUL file with a filename that matches the view identifier. (The XML file is parsed using the Apache Digester [ASF, Digester], based on a set of rules that were instantiated in the constructor (1).)

Usually, `restoreView` would retrieve the component tree from the session, the request, or some other source. However, our sample `ViewHandler` doesn't support state-saving—it simply rebuilds the tree each time.

- ⑥ The `calculateLocales` method (and the `findMatch` utility method) are used to determine the user's locale based on the locale of the incoming request and the locales configured for the application.
- ⑦ The `calculateRenderKitId` method determines the correct render kit for the current request. Because this implementation only works with the standard HTML render kit, we return its identifier. (The default `viewHandler` implementation simply returns the application's default render kit identifier, or the default render kit if the application has no default configured.)
- ⑧ The `getActionURL` method returns a URL based on a view identifier. In this case, we prefix the view identifier with the request context path and the string "/faces". A complete implementation would also support suffix mapping (.faces), and would make the proper selection based on servlet context init parameters.
- ⑨ The `getResourceURL` method generates the appropriate URL for any resources that are not views. Here, we simply prefix the path with the request context path if it starts with a slash ("/"); this is exactly what the default `ViewHandler` does.
- ⑩ Usually, the `writeState` method would save the view's state, depending on the `StateManager` for the real work. Since this `ViewHandler` doesn't support state saving, this method does nothing.

In the example, this class is registered in code via a `ServletContextListener`, as shown in listing C.3.

Listing C.3 Registering a `ViewHandler` with a `ServletContextListener`

```
package nonjsp.lifecycle;

import nonjsp.application.XulViewHandlerImpl;

import javax.faces.FactoryFinder;
import javax.faces.application.*;
import javax.servlet.*;

public class XulServletContextListener
    implements ServletContextListener
{
    public XulServletContextListener()
```

```
{  
}  
  
public void contextInitialized(ServletContextEvent event)  
{  
    ViewHandler handler = new XulViewHandlerImpl();  
    ApplicationFactory factory = (ApplicationFactory)  
        FactoryFinder.getFactory(FactoryFinder.APPLICATION_FACTORY);  
    Application application = factory.getApplication();  
    application.setViewHandler(handler);  
}  
  
public void contextDestroyed(ServletContextEvent e)  
{  
}  
}
```

All of the work happens in the `contextInitialized` method (which is called when the web application is loaded), where we create a new `XulViewHandlerImpl` instance, retrieve an `Application` instance, and then set its `viewHandler` property to equal our new `XulViewHandlerImpl` instance. We use the `ApplicationFactory` to retrieve the `Application` instance because no `FacesContext` is available when the application initializes.

This `ServletContextListener` is a good example of how to use `FactoryFinder` to retrieve a factory, but there's no usually no need to initialize a pluggable class in code. Instead, you can simply use an application configuration file:

```
<application>  
...  
    <view-handler>nonjsp.application.XulViewHandlerImpl</view-handler>  
...  
</application>
```

That's all there is to it. With this simple declaration, we've replaced JSP with XUL for this application's display technology. This is the power of JSF's pluggable architecture: you can replace or enhance core functionality simply by writing a class and adding it to a configuration file.

JSF configuration

Like most Java web frameworks, JSF application configuration is handled in an XML file, usually called faces-config.xml. In this file, you can configure supported locales, managed beans, and navigation rules, as well as replace pluggable JSF classes. You can also configure UI extensions (components, renderers, validators, and converters) and other advanced features in this file. All of these items can be set in code, although doing so is generally only useful in dynamic scenarios.

BY THE WAY

If you're not fond of editing XML files, tools are available to help you. There's a freely available configuration file editor called the Faces Console [Holmes] that plugs into many popular IDEs and also runs as a stand-alone application. Exadel also offers its JSF Studio product Eclipse plugin [JSF Studio] that simplifies configuration and navigation, among other things. In addition, most IDEs provide visual editors for some or all aspects of configuration. The JSF Central community site [JSF Central] maintains a product directory of JSF tools.

Recall that JSF can support *several* configuration files. By default, it will look for a file named WEB-INF/faces-config.xml, and indeed this is where you'll put most of your application configuration. You can also specify additional files with the javax.faces.CONFIG_FILES context parameter. This parameter can be useful in cases where you want to segment your configuration for easier maintenance. For example, two different teams might be working on different modules that have different configuration files. JSF will also search for configuration files named META-INF/faces-config.xml in JAR files (or any other resource path); this allows you or third parties to create libraries of components, renderers, validators, and/or converters that are automatically registered by your application.

All application configuration files must reference the JSF configuration Document Type Definition (DTD) like so:

```
<!DOCTYPE faces-config PUBLIC  
        "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"  
        "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">
```

(If you're using JSF 1.0, the URI would be "http://java.sun.com/dtd/web-facesconfig_1_0.dtd"). Figure D.1 depicts the basic structure of a configuration file.

The following sections explain each element of the DTD in detail; see chapter 3 for an overview and examples, as well as information about web application configuration via web.xml. Note that many elements are optional, especially for UI extensions. Often, the optional elements are useful for IDE integration.

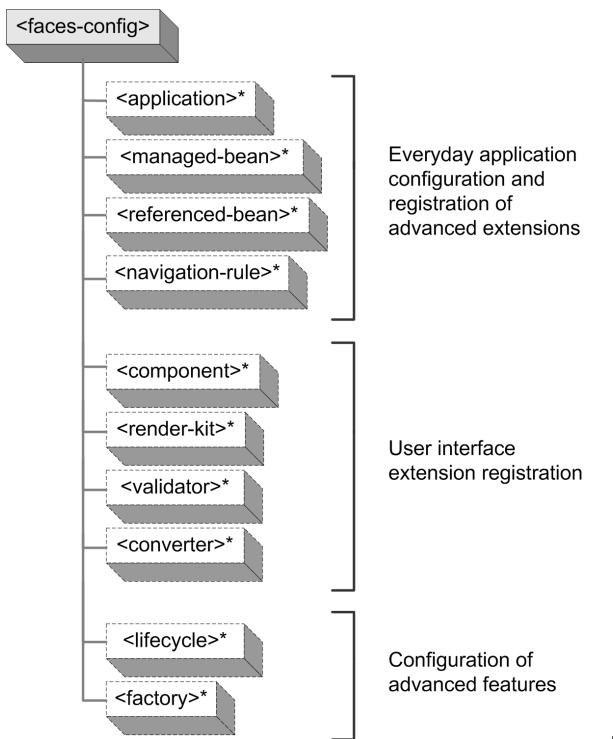


Figure D.1
Basic structure of a
JSF configuration file.

D.1 Common attributes

JSF configuration elements don't have many attributes. As a matter of fact, there are only two (see table D.1), which are used by several different elements.

Table D.1 Common attributes

Attribute Name	Required?	Description	Supported By
<code>id</code>	No	The unique identifier for the element. This is an XML identifier (of type <code>ID</code>), and should not be confused with a JSF identifier.	All elements
<code>xml:lang</code>	No	The locale string for this element's content. A locale string is made up of a language code, a country code (optional), and a variant (optional), with an underscore ("_") or hyphen (" - ") in between. For example, U.S. English is "en_US" (see online extension appendix E for a list of language and country codes).	<icon>, <description>, <display-name>

D.2 Common elements

The following sections describe common child elements that are used by the top-level configuration elements.

D.2.1 <icon>

For integration with tools, several configuration elements support icons. Usually an icon is displayed to help represent the associated element. For example, UI components usually have an icon displayed next to the component's name in a component palette. The `<icon>` element accepts the `xml:lang` attribute, and its child elements are described in table D.2.

Table D.2 Child elements for <icon>

Element Name	Number of Instances	Description
<code><small-icon></code>	0 or 1	The resource path of the small icon (16x16) in either GIF or JPEG format
<code><large-icon></code>	0 or 1	The resource path of the large icon (32x32) in either GIF or JPEG format

D.2.2 <property>

UI extensions, such as UI components and validators, have JavaBean properties that can optionally be described in a configuration file for use by tools. These are defined with the `<property>` element, whose child elements are listed in table D.3.

Table D.3 Child elements for <property>

Element Name	Number of Instances	Description
<code><description></code>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<code><display-name></code>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<code><icon></code>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<code><property-name></code>	1	The canonical name of the property (must be a JavaBean property).
<code><property-class></code>	1	The fully qualified Java class name for this property.

continued on next page

Table D.3 Child elements for <property> (continued)

Element Name	Number of Instances	Description
<default-value>	0 or 1	The property's default value.
<suggested-value>	0 or 1	The property's suggested value. In tools, the property <i>may</i> be initialized to this value (for instance, in a property inspector).
<property-extension>	0 or more	Extra element for use by tools and implementations.

D.2.3 <attribute>

UI extensions, such as UI components and converters, may accept attributes that modify their behavior. These are usually added dynamically (via the `attributes` property), unlike strongly typed properties. The `<attribute>` element's child elements are listed in table D.4.

Table D.4 Child elements for <attribute>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<attribute-name>	1	The attribute name.
<attribute-class>	1	The fully qualified Java class name for this attribute.
<default-value>	0 or 1	The attribute's default value.
<suggested-value>	0 or 1	The attribute's suggested value. In tools, the attribute <i>may</i> be initialized to this value (for instance, in a property inspector).
<attribute-extension>	0 or more	Extra element for use by tools and implementations.

D.2.4 <facet>

UI components and renderers support name child-like components called facets. Facets are declared using the <facet> element, usually to help with tool support; table D.5 lists all of the child elements.

Table D.5 Child elements for <facet>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<facet-name>	1	The facet name.
<facet-extension>	0 or more	Extra element for use by tools and implementations.

D.3 Everyday configuration and pluggable classes

In this section, we examine the configuration elements you use during the course of normal application development.

D.3.1 <application>

The <application> element contains basic configuration information, such as the supported locales, default render kit, and message bundle. It's also where you extend JSF's functionality with pluggable classes. The child elements are listed in table D.6.

Table D.6 Child elements for <application>

Element Name	Number of Instances	Description
<action-listener>	0 or 1	The fully qualified Java class name for a replacement action listener.
<default-render-kit-id>	0 or 1	The default render kit identifier; must map to a render kit identifier defined in a configuration file (either the application's configuration file or one included in a JAR).

continued on next page

Table D.6 Child elements for <application> (continued)

Element Name	Number of Instances	Description
<message-bundle>	0 or 1	The base name of the application's message resource bundle, which should exist somewhere in the resource path. Strings defined in this bundle can replace standard validation and conversion error messages.
<navigation-handler>	0 or 1	The fully qualified Java class name for a replacement NavigationHandler.
<view-handler>	0 or 1	The fully qualified Java class name for a replacement ViewHandler.
<state-manager>	0 or 1	The fully qualified Java class name for a replacement StateManager.
<property-resolver>	0 or 1	The fully qualified Java class name for a replacement PropertyResolver.
<variable-resolver>	0 or 1	The fully qualified Java class name for a replacement VariableResolver.
<locale-config>	0 or 1	The list of supported locales and the default locale. See the <locale-config> section.

For more information on pluggable classes, see online extension appendix C.

<locale-config>

The <locale-config> element is used within an <application> element to define the locales that the application supports. A locale string is made up of a language code, a country code (optional), and a variant (optional), with an underscore (_) or hyphen (-) in between. For example, U.S. English is “en_US” (see online extension appendix E for a list of language and country codes). Table D.7 lists the <locale-config> required elements.

Table D.7 Child elements for <locale-config>

Element Name	Number of Instances	Description
<default-locale>	0 or 1	The locale string for the default locale this application should support. If not specified, the default locale for the VM will be used.
<supported-locale>	0 or more	The locale string for a locale supported by the application.

The user's current locale is determined based on a client application's locale settings and the locales supported by the application itself. Web browsers send an HTTP header that specifies the languages they support. So, for JSF HTML applications, the user's locale is selected based on the union of the browser's locales and the application's supported locales.

For example, if your browser's primary language is Spanish (es_ES) and you access a JSF application that supports French (fr or fr_FR) as the default *and* Spanish, you'll see Spanish text. But, if the JSF application doesn't support Spanish, you'll see French instead.

D.3.2 <managed-bean>

The <managed-bean> element is used to configure a managed bean, which will be created and initialized by the Managed Bean Creation facility the first time it is requested by a JSF EL expression. For a detailed discussion of managed bean configuration, see chapter 3. The supported child elements are listed in table D.8.

Table D.8 Child elements for <managed-bean>

Element name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<managed-bean-name>	1	The name of this bean (used in JSP, JSTL, or JSF EL expressions).
<managed-bean-class>	1	Fully qualified Java class name for this bean.
<managed-bean-scope>	1	Web application scope for this bean (application, session, or request).
<managed-property>	0 or more	Specific properties to be initialized when this bean is created (see the <managed-property> section). Any properties not explicitly declared will retain any values set in the bean's no-argument constructor. If you specify a <managed-property> element, you cannot specify <map-entries> or <list-entries>.

continued on next page

Table D.8 Child elements for <managed-bean> (continued)

Element name	Number of Instances	Description
<map-entries>	0 or 1	List of entries if this managed bean is a Map instance (see the <map-entries> section.) If you specify a <map-entries> element, you cannot specify a <managed-property> or <list-entries> element.
<list-entries>	0 or 1	List of elements if this managed bean is a List (see the <list-entries> section). If you specify a <list-entries> element, you cannot specify a <managed-property> or <map-entries> element.

<managed-property>

The <managed-property> element describes a property that can be initialized when a managed bean is first created; its child elements are listed in table D.9.

Table D.9 Child elements for <managed-property>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<managed-property-name>	1	The canonical name of the property (must be a Java-Bean property).
<managed-property-class>	1	The fully qualified Java class name for this property, or primitive type (<code>int</code> , <code>boolean</code> , <code>char</code> , and so on).
<value>	0 or 1	Value to initialize this property with. Must be convertible to the property's class. You must select either this element or <null-value>, <map-entries>, or <list-entries>.
<null-value>	0 or 1	Initialize this property to <code>null</code> . This is only valid if the property's type is not a primitive. You must select either this element or <value>, <map-entries>, or <list-entries>.

continued on next page

Table D.9 Child elements for <managed-property> (continued)

Element Name	Number of Instances	Description
<map-entries>	0 or 1	Initialize a property of type Map (see the <map-entries> section). If the property is null, a new HashMap will be created. If the property has already been initialized, any specified entries will be added to it. You must select either this element or <value>, <null-value>, or <list-entries>.
<list-entries>	0 or 1	Initialize a property of type List or array (see the <list-entries> section). If the property is null, a new ArrayList will be created. If the property has already been initialized, any specified entries will be added to it. You must select either this element or <value>, <null-value>, or <map-entries>.

<map-entries>

The <map-entries> element initializes either a managed bean or managed bean property that is a Map. Its child elements are listed in table D.10.

Table D.10 Child elements for <map-entries>

Element Name	Number of Instances	Description
<key-class>	0 or 1	The fully qualified Java class name for keys in this Map. If omitted, keys are assumed to be strings.
<value-class>	0 or 1	The fully qualified Java class name for values in this Map. If omitted, values are assumed to be strings.
<map-entry>	0 or more	The actual entry to be added to the Map (see the <map-entry> section).

<map-entry>

The <map-entry> element specifies an individual name/value pair within a <map-entries> element. Its child elements are listed in table D.11.

Table D.11 Child elements for <map-entry>

Element Name	Number of Instances	Description
<key>	0 or 1	The value of this entry's key. Must be convertible to the type specified in the <key-class> element (which is a child of <map-entries>).

continued on next page

Table D.11 Child elements for `<map-entry>` (continued)

Element Name	Number of Instances	Description
<code><value></code>	0 or 1	This entry's value. Must be convertible to the type specified in the <code><value-class></code> element (which is a child of <code><map-entries></code>).
<code><null-value></code>	0 or 1	Sets this entry's value to <code>null</code> .

<list-entries>

The `<list-entries>` element initializes a managed bean that is a List, or a managed bean property that is a List or array. Its child elements are listed in table D.12.

Table D.12 Child elements for `<list-entries>`

Element Name	Number of Instances	Description
<code><value-class></code>	0 or 1	The fully qualified Java class name or primitive type for values in this List or array. If omitted, values are assumed to be strings.
<code><value></code>	0 or more	An entry's value. Must be convertible to the type specified in the <code><value-class></code> element.
<code><null-value></code>	0 or more	Sets an entry's value to <code>null</code> . Not valid if the type specified in the <code><value-class></code> element is a primitive type.

D.3.3 <referenced-bean>

Applications may use scoped variables, which are set in code instead of by the Managed Bean Creation facility. Such variables can be declared with the `<referenced-bean>` element so that tools know when they're available. The child elements for `<referenced-bean>` are listed in table D.13.

Table D.13 Child elements for `<referenced-bean>`

Element Name	Number of Instances	Description
<code><description></code>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<code><display-name></code>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).

continued on next page

Table D.13 Child elements for <referenced-bean> (continued)

Element Name	Number of Instances	Description
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<referenced-bean-name>	1	The name of this bean will be stored under (used in JSP, JSTL, or JSF EL expressions).
<referenced-bean-class>	1	Fully qualified Java class name for this bean.

D.3.4 <navigation-rule>

Navigation rules, configured with the <navigation-rule> element, control the flow from one view to another. The child elements for <navigation-rule> are listed in table D.14.

Table D.14 Child elements for <navigation-rule>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<from-view-id>	0 or 1	The view identifier (such as "/login.jsp") for which this rule applies. For the default view handler (JSP), all identifiers must begin with a slash "/", and you can use trailing wildcards to perform partial matches, such as "/protected/*". Also, "*" (or omitting this element altogether) matches all views. You can have multiple rules with the same view identifier.
<navigation-case>	0 or more	Maps a specific logical outcome and/or action method to another view (see the <navigation-case> section).

<navigation-case>

Within a <navigation-rule> element, a <navigation-case> maps a specific logical outcome and/or action method to another view. Its child elements are shown in table D.15.

Table D.15 Child elements for <navigation-rule>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
from-action	0 or 1	The method-binding expression for the action method executed by the user; must have the proper syntax (i.e., "#{myBean.myMethod}"). If not specified, this case will match regardless of the action method.
from-outcome	0 or 1	The logical outcome for this navigation case. It should be returned by a UI component or action method on the view for which this case applies. If not specified, this case will match regardless of the outcome.
to-view-id	1	The identifier of the view to display to the user if this navigation case matches. With the default view handler (JSP), this should be a JSP patch with a leading slash ("content/about.jsp").
redirect	0 or 1	Indicates whether navigation should be handled via an HTTP redirect as opposed to a forward inside the web container. The primary benefit is that users see the actual URL of the page they are viewing. If not specified, a web container forward will be executed.

D.4 User interface extensions

All of JSF's UI extensions—components, renderers, validators, and converters—are configured through an application configuration file. You can use these configuration elements to add new UI extensions or replace existing ones.

NOTE Even though you can specify metadata for UI extensions with standard JSF configuration files, most IDE vendors require additional work to integrate extensions into their IDEs. This process should be standardized in the future.

D.4.1 <component>

The <component> element registers a single UI component with JSF. Only two child elements are required, but if you are integrating your component with a

tool, you should supply as much metadata as possible. This element's children are listed in table D.16.

Table D.16 Child elements for <component>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<component-type>	1	The unique name for this component (usually a qualified string like "javax.faces.UIOutput"). The "javax.faces" prefix is reserved for use by the JSF implementation, but you can change the class of standard components by using the standard component's type.
<component-class>	1	The fully qualified Java class name for this component. Must implement the javax.faces.component.UICOMPONENT interface.
<facet>	0 or more	A subordinate named component, like a header or footer (see section D.2.4). Used by tools.
<attribute>	0 or more	A named attribute recognized by this UI component (see section D.2.3). Used by tools.
<property>	0 or more	A JavaBean property exposed by this UI component for use with tools (see section D.2.2).
<component-extension>	0 or more	Extra element for use by tools and implementations.

D.4.2 <render-kit>

The <render-kit> element can be used to add new renderers to the standard HTML render kit, or create a new render kit. Its child elements are described in table D.17.

Table D.17 Child elements for <render-kit>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).

continued on next page

Table D.17 Child elements for <render-kit> (continued)

Element Name	Number of Instances	Description
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<render-kit-id>	0 or 1	The unique name for this render kit. If not specified, the default render kit ("HTML_BASIC") will be used.
<render-kit-class>	0 or 1	The fully qualified Java class name for this render kit. Must be a subclass of javax.faces.render.RenderKit. If not specified, the implementation's default RenderKit implementation will be used.
<renderer>	0 or more	Specifies a renderer that should be added to this render kit (see the <renderer> section).

<renderer>

The <renderer> element defines a single renderer, adds it to the render kit referenced by the enclosing <render-kit> element, and associates it with a component family. Its child elements are listed in table D.18.

Table D.18 Child elements for <renderer>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools.
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1).
<renderer-type>	1	The unique name for this renderer (usually a qualified string like "javax.faces.Form"). The "javax.faces" prefix is reserved for use by the JSF implementation, but you can change the class of a standard renderer by using the standard renderer's type.
<renderer-class>	1	The fully qualified Java class name for this renderer. Must be a subclass of javax.faces.render.Renderer.
<facet>	0 or more	A subordinate named component, like a header or footer, expected in the rendered UI component (see section D.2.4). Used by tools.

continued on next page

Table D.18 Child elements for <renderer> (continued)

Element Name	Number of Instances	Description
<attribute>	0 or more	A renderer-specific named attribute expected in the renderer UI component (see section D.2.3). Used by tools.
<renderer-extension>	0 or more	Extra element for use by tools and implementations.

D.4.3 <validator>

The <validator> element defines a named validator class (not a validator method) that can be associated with an input control. Its child elements are shown in table D.19.

Table D.19 Child elements for <validator>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<validator-id>	1	The unique name for this validator, such as "javax.faces.LongRange". The "javax.faces" prefix is reserved for use by the JSF implementation, but you can change the class of standard validators by using the standard validator's type.
<validator-class>	1	The fully qualified Java class name for this validator. Must implement the <code>javax.faces.validator.Validator</code> interface.
<attribute>	0 or more	A named attribute recognized by this validator (see section D.2.3). Used by tools.
<property>	0 or more	A JavaBean property exposed by this validator for use with tools (see section D.2.2).

D.4.4 <converter>

The <converter> element defines a converter, which can be registered by type or identifier. If you want to register a converter by *both* type and identifier, simply declare it twice. The child elements are listed in table D.20.

Table D.20 Child elements for <converter>

Element Name	Number of Instances	Description
<description>	0 or more	A text description, usually only one sentence. You would usually have one instance per language (see section D.1).
<display-name>	0 or more	The name to be displayed in tools. You would usually have one instance per language (see section D.1).
<icon>	0 or more	The icon to be displayed in tools (see section D.2.1). You would usually have one instance per language (see section D.1).
<converter-id>	0 or 1	The unique name for this converter, such as "javax.faces.DateTime". The "javax.faces" prefix is reserved for use by the JSF implementation, but you can change the class of standard converters by using the standard converter's type. You can specify either a <converter-id> element or a <converter-for-class> element.
<converter-for-class>	0 or 1	The fully qualified Java class name this converter handles. If this element is specified, JSF will automatically use this converter every time the class is encountered. You can specify either a <converter-id> element or a <converter-for-class> element.
<converter-class>	1	The fully qualified Java class name for this component. Must implement the javax.faces.convert.Converter interface.
<attribute>	0 or more	A named attribute recognized by this validator (see section D.2.3). Used by tools.
<property>	0 or more	A JavaBean property exposed by this validator for use with tools (see section D.2.2).

D.5 Advanced features

The following sections describe configuration elements that are not commonly used but that are useful for extending JSF's capabilities in some cases.

D.5.1 <lifecycle>

The <lifecycle> element allows you to register one or more phase listeners, which are executed before or after phases of the Request Processing Lifecycle. It accepts a single element, which is described in table D.21.

Table D.21 Child element for <lifecycle>

Element Name	Number of Instances	Description
<phase-listener>	0 or more	The fully qualified Java class name for a PhaseListener implementation.

D.5.2 <factory>

All of JSF's core classes (other than pluggable classes) are created via factories that can be configured with the <factory> element. Multiple factories of the same type can be specified, and they will be chained together (see appendix C). If this element is not specified, the default factories are used. This element is described in table D.22.

Table D.22 Child elements for <factory>

Element Name	Number of Instances	Description
<application-factory>	0 or more	The fully qualified Java class name for an ApplicationFactory subclass
<faces-context-factory>	0 or more	The fully qualified Java class name for a FacesContextFactory subclass
<lifecycle-factory>	0 or more	The fully qualified Java class name for a LifecycleFactory subclass
<render-kit-factory>	0 or more	The fully qualified Java class name for a RenderKitFactory subclass

*Time zone,
country, language,
and currency codes*

In chapter 6, we covered internationalization, localization, and type conversion. All of these features have one unique property: they allow your application to work for users from parts of the world. The world is a big place, so the software development industry has categorized different regions with codes to make life easier. Time zone codes categorize different regions by their relation to Greenwich Mean Time (GMT). Language codes represent specific languages, and country codes represent specific countries. (A language code and a country code combined make up a *locale*.) Currency codes represent specific currencies.

This appendix provides listings of all of these codes. See chapter 6 for details about where to use them.

E.1 Time zone codes

When using the `DateTime` converter, you can specify a `timeZone` property, so that the value is converted to the appropriate time zone. (Converters are covered in chapter 6.) To do this, you need to use a time zone identifier. The valid time zone identifiers, as of JDK 1.4.2, are listed in table E.1.

Table E.1 When using the `timeZone` attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT.

Time zone identifier	Description	Time zone identifier	Description
Etc/GMT+12	GMT-12:00	Africa/Luanda	Western African Time
Etc/GMT+11	GMT-11:00	Africa/Malabo	Western African Time
MIT	West Samoa Time	Africa/Ndjamena	Western African Time
Pacific/Api	West Samoa Time	Africa/Niamey	Western African Time
Pacific/Midway	Samoa Standard Time	Africa/Porto-Novo	Western African Time
Pacific/Niue	Niue Time	Africa/Tunis	Central European Time
Pacific/Pago_Pago	Samoa Standard Time	Africa/Windhoek	Western African Time
Pacific/Samoa	Samoa Standard Time	Arctic/Longyearbyen	Central European Time
US/Samoa	Samoa Standard Time	Atlantic/Jan_Mayen	Eastern Greenland Time
America/Adak	Hawaii-Aleutian Standard Time	CET	Central European Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/Atka	Hawaii-Aleutian Standard Time	ECT	Central European Time
Etc/GMT+10	GMT-10:00	Etc/GMT-1	GMT+01:00
HST	Hawaii Standard Time	Europe/Amsterdam	Central European Time
Pacific/Fakaofo	Tokelau Time	Europe/Andorra	Central European Time
Pacific/Honolulu	Hawaii Standard Time	Europe/Belgrade	Central European Time
Pacific/Johnston	Hawaii Standard Time	Europe/Berlin	Central European Time
Pacific/Rarotonga	Cook Is. Time	Europe/Bratislava	Central European Time
Pacific/Tahiti	Tahiti Time	Europe/Brussels	Central European Time
SystemV/HST10	Hawaii Standard Time	Europe/Budapest	Central European Time
US/Aleutian	Hawaii-Aleutian Standard Time	Europe/Copenhagen	Central European Time
US/Hawaii	Hawaii Standard Time	Europe/Gibraltar	Central European Time
Pacific/Marquesas	Marquesas Time	Europe/Ljubljana	Central European Time
AST	Alaska Standard Time	Europe/Luxembourg	Central European Time
America/Anchorage	Alaska Standard Time	Europe/Madrid	Central European Time
America/Juneau	Alaska Standard Time	Europe/Malta	Central European Time
America/Nome	Alaska Standard Time	Europe/Monaco	Central European Time
America/Yakutat	Alaska Standard Time	Europe/Oslo	Central European Time
Etc/GMT+9	GMT-09:00	Europe/Paris	Central European Time
Pacific/Gambier	Gambier Time	Europe/Prague	Central European Time
SystemV/YST9	Gambier Time	Europe/Rome	Central European Time
SystemV/YST9YDT	Alaska Standard Time	Europe/San_Marino	Central European Time
US/Alaska	Alaska Standard Time	Europe/Sarajevo	Central European Time
America/Dawson	Pacific Standard Time	Europe/Skopje	Central European Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/Ensenada	Pacific Standard Time	Europe/Stockholm	Central European Time
America/Los_Angeles	Pacific Standard Time	Europe/Tirane	Central European Time
America/Tijuana	Pacific Standard Time	Europe/Vaduz	Central European Time
America/Vancouver	Pacific Standard Time	Europe/Vatican	Central European Time
America/Whitehorse	Pacific Standard Time	Europe/Vienna	Central European Time
Canada/Pacific	Pacific Standard Time	Europe/Warsaw	Central European Time
Canada/Yukon	Pacific Standard Time	Europe/Zagreb	Central European Time
Etc/GMT+8	GMT-08:00	Europe/Zurich	Central European Time
Mexico/BajaNorte	Pacific Standard Time	MET	Middle Europe Time
PST	Pacific Standard Time	Poland	Central European Time
PST8PDT	Pacific Standard Time	ART	Eastern European Time
Pacific/Pitcairn	Pitcairn Standard Time	Africa/Blantyre	Central African Time
SystemV/PST8	Pitcairn Standard Time	Africa/Bujumbura	Central African Time
SystemV/PST8PDT	Pacific Standard Time	Africa/Cairo	Eastern European Time
US/Pacific	Pacific Standard Time	Africa/Gaborone	Central African Time
US/Pacific-New	Pacific Standard Time	Africa/Harare	Central African Time
America/Boise	Mountain Standard Time	Africa/Johannesburg	South Africa Standard Time
America/Cambridge_Bay	Mountain Standard Time	Africa/Kigali	Central African Time
America/Chihuahua	Mountain Standard Time	Africa/Lubumbashi	Central African Time
America/Dawson_Creek	Mountain Standard Time	Africa/Lusaka	Central African Time
America/Denver	Mountain Standard Time	Africa/Maputo	Central African Time
America/Edmonton	Mountain Standard Time	Africa/Maseru	South Africa Standard Time
America/Hermosillo	Mountain Standard Time	Africa/Mbabane	South Africa Standard Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/Inuvik	Mountain Standard Time	Africa/Tripoli	Eastern European Time
America/Mazatlan	Mountain Standard Time	Asia/Amman	Eastern European Time
America/Phoenix	Mountain Standard Time	Asia/Beirut	Eastern European Time
America/Shiprock	Mountain Standard Time	Asia/Damascus	Eastern European Time
America/Yellowknife	Mountain Standard Time	Asia/Gaza	Eastern European Time
Canada/Mountain	Mountain Standard Time	Asia/Istanbul	Eastern European Time
Etc/GMT+7	GMT-07:00	Asia/Jerusalem	Israel Standard Time
MST	Mountain Standard Time	Asia/Nicosia	Eastern European Time
MST7MDT	Mountain Standard Time	Asia/Tel_Aviv	Israel Standard Time
Mexico/BajaSur	Mountain Standard Time	CAT	Central African Time
Navajo	Mountain Standard Time	EET	Eastern European Time
PNT	Mountain Standard Time	Egypt	Eastern European Time
SystemV/MST7	Mountain Standard Time	Etc/GMT-2	GMT+02:00
SystemV/MST7MDT	Mountain Standard Time	Europe/Athens	Eastern European Time
US/Arizona	Mountain Standard Time	Europe/Bucharest	Eastern European Time
US/Mountain	Mountain Standard Time	Europe/Chisinau	Eastern European Time
America/Belize	Central Standard Time	Europe/Helsinki	Eastern European Time
America/Cancun	Central Standard Time	Europe/Istanbul	Eastern European Time
America/Chicago	Central Standard Time	Europe/Kaliningrad	Eastern European Time
America/Costa_Rica	Central Standard Time	Europe/Kiev	Eastern European Time
America/El_Salvador	Central Standard Time	Europe/Minsk	Eastern European Time
America/Guatemala	Central Standard Time	Europe/Nicosia	Eastern European Time
America/Managua	Central Standard Time	Europe/Riga	Eastern European Time
America/Menominee	Central Standard Time	Europe/Simferopol	Eastern European Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/Merida	Central Standard Time	Europe/Sofia	Eastern European Time
America/Mexico_City	Central Standard Time	Europe/Tallinn	Eastern European Time
America/Monterrey	Central Standard Time	Europe/Tiraspol	Eastern European Time
America/North_Dakota/Center	Central Standard Time	Europe/Uzhgorod	Eastern European Time
America/Rainy_River	Central Standard Time	Europe/Vilnius	Eastern European Time
America/Rankin_Inlet	Eastern Standard Time	Europe/Zaporozhye	Eastern European Time
America/Regina	Central Standard Time	Israel	Israel Standard Time
America/Swift_Current	Central Standard Time	Libya	Eastern European Time
America/Tegucigalpa	Central Standard Time	Turkey	Eastern European Time
America/Winnipeg	Central Standard Time	Africa/Addis_Ababa	Eastern African Time
CST	Central Standard Time	Africa/Asmera	Eastern African Time
CST6CDT	Central Standard Time	Africa/Dar_es_Salaam	Eastern African Time
Canada/Central	Central Standard Time	Africa/Djibouti	Eastern African Time
Canada/East-Saskatchewan	Central Standard Time	Africa/Kampala	Eastern African Time
Canada/Saskatchewan	Central Standard Time	Africa/Khartoum	Eastern African Time
Chile/EasterIsland	Easter Is. Time	Africa/Mogadishu	Eastern African Time
Etc/GMT+6	GMT-06:00	Africa/Nairobi	Eastern African Time
Mexico/General	Central Standard Time	Antarctica/Syowa	Syowa Time
Pacific/Easter	Easter Is. Time	Asia/Aden	Arabia Standard Time
Pacific/Galapagos	Galapagos Time	Asia/Baghdad	Arabia Standard Time
SystemV/CST6	Central Standard Time	Asia/Bahrain	Arabia Standard Time
SystemV/CST6CDT	Central Standard Time	Asia/Kuwait	Arabia Standard Time
US/Central	Central Standard Time	Asia/Qatar	Arabia Standard Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/Bogota	Colombia Time	Asia/Riyadh	Arabia Standard Time
America/Cayman	Eastern Standard Time	EAT	Eastern African Time
America/Detroit	Eastern Standard Time	Etc/GMT-3	GMT+03:00
America/Eirunepe	Acre Time	Europe/Moscow	Moscow Standard Time
America/Fort_Wayne	Eastern Standard Time	Indian/Antananarivo	Eastern African Time
America/Grand_Turk	Eastern Standard Time	Indian/Comoro	Eastern African Time
America/Guayaquil	Ecuador Time	Indian/Mayotte	Eastern African Time
America/Havana	Central Standard Time	W-SU	Moscow Standard Time
America/Indiana/Indianapolis	Eastern Standard Time	Asia/Riyadh87	GMT+03:07
America/Indiana/Knox	Eastern Standard Time	Asia/Riyadh88	GMT+03:07
America/Indiana/Marengo	Eastern Standard Time	Asia/Riyadh89	GMT+03:07
America/Indiana/Vevay	Eastern Standard Time	Mideast/Riyadh87	GMT+03:07
America/Indianapolis	Eastern Standard Time	Mideast/Riyadh88	GMT+03:07
America/Iqaluit	Eastern Standard Time	Mideast/Riyadh89	GMT+03:07
America/Jamaica	Eastern Standard Time	Asia/Tehran	Iran Standard Time
America/Kentucky/Louisville	Eastern Standard Time	Iran	Iran Standard Time
America/Kentucky/Monticello	Eastern Standard Time	Asia/Aqtau	Aqtau Time
America/Knox_IN	Eastern Standard Time	Asia/Baku	Azerbaijan Time
America/Lima	Peru Time	Asia/Dubai	Gulf Standard Time
America/Louisville	Eastern Standard Time	Asia/Muscat	Gulf Standard Time
America/Montreal	Eastern Standard Time	Asia/Oral	Oral Time
America/Nassau	Eastern Standard Time	Asia/Tbilisi	Georgia Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/New_York	Eastern Standard Time	Asia/Yerevan	Armenia Time
America/Nipigon	Eastern Standard Time	Etc/GMT-4	GMT+04:00
America/Panama	Eastern Standard Time	Europe/Samara	Samara Time
America/Pangnirtung	Eastern Standard Time	Indian/Mahe	Seychelles Time
America/Port-au-Prince	Eastern Standard Time	Indian/Mauritius	Mauritius Time
America/Porto_Acre	Acre Time	Indian/Reunion	Reunion Time
America/Rio_Branco	Acre Time	NET	Armenia Time
America/Thunder_Bay	Eastern Standard Time	Asia/Kabul	Afghanistan Time
Brazil/Acre	Acre Time	Asia/Aqtobe	Aqtobe Time
Canada/Eastern	Eastern Standard Time	Asia/Ashgabat	Turkmenistan Time
Cuba	Central Standard Time	Asia/Ashkhabad	Turkmenistan Time
EST	Eastern Standard Time	Asia/Bishkek	Kirgizstan Time
EST5EDT	Eastern Standard Time	Asia/Dushanbe	Tajikistan Time
Etc/GMT+5	GMT-05:00	Asia/Karachi	Pakistan Time
IET	Eastern Standard Time	Asia/Samarkand	Turkmenistan Time
Jamaica	Eastern Standard Time	Asia/Tashkent	Uzbekistan Time
SystemV/EST5	Eastern Standard Time	Asia/Yekaterinburg	Yekaterinburg Time
SystemV/EST5EDT	Eastern Standard Time	Etc/GMT-5	GMT+05:00
US/East-Indiana	Eastern Standard Time	Indian/Kerguelen	French Southern & Antarctic Lands Time
US/Eastern	Eastern Standard Time	Indian/Maldives	Maldives Time
US/Indiana-Starke	Eastern Standard Time	PLT	Pakistan Time
US/Michigan	Eastern Standard Time	Asia/Calcutta	India Standard Time
America/Anguilla	Atlantic Standard Time	IST	India Standard Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/Antigua	Atlantic Standard Time	Asia/Katmandu	Nepal Time
America/Aruba	Atlantic Standard Time	Antarctica/Mawson	Mawson Time
America/Asuncion	Paraguay Time	Antarctica/Vostok	Vostok Time
America/Barbados	Atlantic Standard Time	Asia/Almaty	Alma-Ata Time
America/Boa_Vista	Amazon Standard Time	Asia/Colombo	Sri Lanka Time
America/Caracas	Venezuela Time	Asia/Dacca	Bangladesh Time
America/Cuiaba	Amazon Standard Time	Asia/Dhaka	Bangladesh Time
America/Curacao	Atlantic Standard Time	Asia/Novosibirsk	Novosibirsk Time
America/Dominica	Atlantic Standard Time	Asia/Omsk	Omsk Time
America/Glace_Bay	Atlantic Standard Time	Asia/Qyzylorda	Qyzylorda Time
America/Goose_Bay	Atlantic Standard Time	Asia/Thimbu	Bhutan Time
America/Grenada	Atlantic Standard Time	Asia/Thimphu	Bhutan Time
America/Guadeloupe	Atlantic Standard Time	BST	Bangladesh Time
America/Guyana	Guyana Time	Etc/GMT-6	GMT+06:00
America/Halifax	Atlantic Standard Time	Indian/Chagos	Indian Ocean Territory Time
America/La_Paz	Bolivia Time	Asia/Rangoon	Myanmar Time
America/Manaus	Amazon Standard Time	Indian/Cocos	Cocos Islands Time
America/Martinique	Atlantic Standard Time	Antarctica/Davis	Davis Time
America/Montserrat	Atlantic Standard Time	Asia/Bangkok	Indochina Time
America/Port_of_Spain	Atlantic Standard Time	Asia/Hovd	Hovd Time
America/Porto_Velho	Amazon Standard Time	Asia/Jakarta	West Indonesia Time
America/Puerto_Rico	Atlantic Standard Time	Asia/Krasnoyarsk	Krasnoyarsk Time
America/Santiago	Chile Time	Asia/Phnom_Penh	Indochina Time
America/Santo_Domingo	Atlantic Standard Time	Asia/Pontianak	West Indonesia Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/St_Kitts	Atlantic Standard Time	Asia/Saigon	Indochina Time
America/St_Lucia	Atlantic Standard Time	Asia/Vientiane	Indochina Time
America/St_Thomas	Atlantic Standard Time	Etc/GMT-7	GMT+07:00
America/St_Vincent	Atlantic Standard Time	Indian/Christmas	Christmas Island Time
America/Thule	Atlantic Standard Time	VST	Indochina Time
America/Tortola	Atlantic Standard Time	Antarctica/Casey	Western Standard Time (Australia)
America/Virgin	Atlantic Standard Time	Asia/Brunei	Brunei Time
Antarctica/Palmer	Chile Time	Asia/Chongqing	China Standard Time
Atlantic/Bermuda	Atlantic Standard Time	Asia/Chungking	China Standard Time
Atlantic/Stanley	Falkland Is. Time	Asia/Harbin	China Standard Time
Brazil/West	Amazon Standard Time	Asia/Hong_Kong	Hong Kong Time
Canada/Atlantic	Atlantic Standard Time	Asia/Irkutsk	Irkutsk Time
Chile/Continental	Chile Time	Asia/Kashgar	China Standard Time
Etc/GMT+4	GMT-04:00	Asia/Kuala_Lumpur	Malaysia Time
PRT	Atlantic Standard Time	Asia/Kuching	Malaysia Time
SystemV/AST4	Atlantic Standard Time	Asia/Macao	China Standard Time
SystemV/AST4ADT	Atlantic Standard Time	Asia/Macau	China Standard Time
America/St_Johns	Newfoundland Standard Time	Asia/Makassar	Central Indonesia Time
CNT	Newfoundland Standard Time	Asia/Manila	Philippines Time
Canada/Newfoundland	Newfoundland Standard Time	Asia/Shanghai	China Standard Time
AGT	Argentine Time	Asia/Singapore	Singapore Time
America/Araguaina	Brazil Time	Asia/Taipei	China Standard Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
America/Belem	Brazil Time	Asia/Ujung_Pandang	Central Indonesia Time
America/Buenos_Aires	Argentine Time	Asia/Ulaanbaatar	Ulaanbaatar Time
America/Catamarca	Argentine Time	Asia/Ulan_Bator	Ulaanbaatar Time
America/Cayenne	French Guiana Time	Asia/Urumqi	China Standard Time
America/Cordoba	Argentine Time	Australia/Perth	Western Standard Time (Australia)
America/Fortaleza	Brazil Time	Australia/West	Western Standard Time (Australia)
America/Godthab	Western Greenland Time	CTT	China Standard Time
America/Jujuy	Argentine Time	Etc/GMT-8	GMT+08:00
America/Maceio	Brazil Time	Hongkong	Hong Kong Time
America/Mendoza	Argentine Time	PRC	China Standard Time
America/Miquelon	Pierre & Miquelon Standard Time	Singapore	Singapore Time
America/Montevideo	Uruguay Time	Asia/Choibalsan	Choibalsan Time
America/Paramaribo	Suriname Time	Asia/Dili	East Timor Time
America/Recife	Brazil Time	Asia/Jayapura	East Indonesia Time
America/Rosario	Argentine Time	Asia/Pyongyang	Korea Standard Time
America/Sao_Paulo	Brazil Time	Asia/Seoul	Korea Standard Time
Antarctica/Rothera	Rothera Time	Asia/Tokyo	Japan Standard Time
BET	Brazil Time	Asia/Yakutsk	Yakutsk Time
Brazil/East	Brazil Time	Etc/GMT-9	GMT+09:00
Etc/GMT+3	GMT-03:00	JST	Japan Standard Time
America/Noronha	Fernando de Noronha Time	Japan	Japan Standard Time
Atlantic/South_Georgia	South Georgia Standard Time	Pacific/Palau	Palau Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
Brazil/DeNoronha	Fernando de Noronha Time	ROK	Korea Standard Time
Etc/GMT+2	GMT-02:00	ACT	Central Standard Time (Northern Territory)
America/Scoresbysund	Eastern Greenland Time	Australia/Adelaide	Central Standard Time (South Australia)
Atlantic/Azores	Azores Time	Australia/Broken_Hill	Central Standard Time (South Australia/New South Wales)
Atlantic/Cape_Verde	Cape Verde Time	Australia/Darwin	Central Standard Time (Northern Territory)
Etc/GMT+1	GMT-01:00	Australia/North	Central Standard Time (Northern Territory)
Africa/Abidjan	Greenwich Mean Time	Australia/South	Central Standard Time (South Australia)
Africa/Accra	Greenwich Mean Time	Australia/Yancowinna	Central Standard Time (South Australia/New South Wales)
Africa/Bamako	Greenwich Mean Time	AET	Eastern Standard Time (New South Wales)
Africa/Banjul	Greenwich Mean Time	Antarctica/Dumont-D'Urville	Dumont-d'Urville Time
Africa/Bissau	Greenwich Mean Time	Asia/Sakhalin	Sakhalin Time
Africa/Casablanca	Western European Time	Asia/Vladivostok	Vladivostok Time
Africa/Conakry	Greenwich Mean Time	Australia/ACT	Eastern Standard Time (New South Wales)
Africa/Dakar	Greenwich Mean Time	Australia/Brisbane	Eastern Standard Time (Queensland)
Africa/El_Aaiun	Western European Time	Australia/Canberra	Eastern Standard Time (New South Wales)

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
Africa/Freetown	Greenwich Mean Time	Australia/Hobart	Eastern Standard Time (Tasmania)
Africa/Lome	Greenwich Mean Time	Australia/Lindeman	Eastern Standard Time (Queensland)
Africa/Monrovia	Greenwich Mean Time	Australia/Melbourne	Eastern Standard Time (Victoria)
Africa/Nouakchott	Greenwich Mean Time	Australia/NSW	Eastern Standard Time (New South Wales)
Africa/Ouagadougou	Greenwich Mean Time	Australia/Queen-sland	Eastern Standard Time (Queensland)
Africa/Sao_Tome	Greenwich Mean Time	Australia/Sydney	Eastern Standard Time (New South Wales)
Africa/Timbuktu	Greenwich Mean Time	Australia/Tasmania	Eastern Standard Time (Tasmania)
America/Danmarkshavn	Greenwich Mean Time	Australia/Victoria	Eastern Standard Time (Victoria)
Atlantic/Canary	Western European Time	Etc/GMT-10	GMT+10:00
Atlantic/Faeroe	Western European Time	Pacific/Guam	Chamorro Standard Time
Atlantic/Madeira	Western European Time	Pacific/Port_Moresby	Papua New Guinea Time
Atlantic/Reykjavik	Greenwich Mean Time	Pacific/Saipan	Chamorro Standard Time
Atlantic/St_Helena	Greenwich Mean Time	Pacific/Truk	Truk Time
Eire	Greenwich Mean Time	Pacific/Yap	Yap Time
Etc/GMT	GMT+00:00	Australia/LHI	Load Howe Standard Time
Etc/GMT+0	GMT+00:00	Australia/Lord_Howe	Load Howe Standard Time
Etc/GMT-0	GMT+00:00	Asia/Magadan	Magadan Time
Etc/GMT0	GMT+00:00	Etc/GMT-11	GMT+11:00
Etc/Greenwich	Greenwich Mean Time	Pacific/Efate	Vanuatu Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
Etc/UCT	Coordinated Universal Time	Pacific/Guadalcanal	Solomon Is. Time
Etc/UTC	Coordinated Universal Time	Pacific/Kosrae	Kosrae Time
Etc/Universal	Coordinated Universal Time	Pacific/Noumea	New Caledonia Time
Etc/Zulu	Coordinated Universal Time	Pacific/Ponape	Ponape Time
Europe/Belfast	Greenwich Mean Time	SST	Solomon Is. Time
Europe/Dublin	Greenwich Mean Time	Pacific/Norfolk	Norfolk Time
Europe/Lisbon	Western European Time	Antarctica/Mcmurdo	New Zealand Standard Time
Europe/London	Greenwich Mean Time	Antarctica/South_Pole	New Zealand Standard Time
GB	Greenwich Mean Time	Asia/Anadyr	Anadyr Time
GB-Eire	Greenwich Mean Time	Asia/Kamchatka	Petropavlovsk-Kamchatski Time
GMT	Greenwich Mean Time	Etc/GMT-12	GMT+12:00
GMT0	GMT+00:00	Kwajalein	Marshall Islands Time
Greenwich	Greenwich Mean Time	NST	New Zealand Standard Time
Iceland	Greenwich Mean Time	NZ	New Zealand Standard Time
Portugal	Western European Time	Pacific/Auckland	New Zealand Standard Time
UCT	Coordinated Universal Time	Pacific/Fiji	Fiji Time
UTC	Coordinated Universal Time	Pacific/Funafuti	Tuvalu Time

continued on next page

Table E.1 When using the time zone attribute with a Date-related converter, you must use one of these identifiers. Ordered by distance from Greenwich Mean Time (GMT), starting with 12 hours behind GMT. (continued)

Time zone identifier	Description	Time zone identifier	Description
Universal	Coordinated Universal Time	Pacific/Kwajalein	Marshall Islands Time
WET	Western European Time	Pacific/Majuro	Marshall Islands Time
Zulu	Coordinated Universal Time	Pacific/Nauru	Nauru Time
Africa/Algiers	Central European Time	Pacific/Tarawa	Gilbert Is. Time
Africa/Bangui	Western African Time	Pacific/Wake	Wake Time
Africa/Brazzaville	Western African Time	Pacific/Wallis	Wallis & Futuna Time
Africa/Ceuta	Central European Time	NZ-CHAT	Chatham Standard Time
Africa/Douala	Western African Time	Pacific/Chatham	Chatham Standard Time
Africa/Kinshasa	Western African Time	Etc/GMT-13	GMT+13:00
Africa/Lagos	Western African Time	Pacific/Enderbury	Phoenix Is. Time
Africa/Libreville	Western African Time	Pacific/Tongatapu	Tonga Time

E.2 Language codes

Anytime you want to support additional languages, you'll need to know the International Organization for Standardization (ISO) language code, which is a lower-case two-letter string. The two tables that follow list the ISO language codes and a description of the language they represent, as of JDK 1.4.2. Table E.2 lists them sorted by language code, and table E.3 lists them sorted by language description.

Table E.2 ISO language codes, as specified by ISO 639-1 (sorted by code)

Code	Language	Code	Language	Code	Language
aa	Afar	id	Indonesian	ro	Romanian
ab	Abkhazian	ie	Interlingue	ru	Russian
af	Afrikaans	ik	Inupiaq	rw	Kinyarwanda
am	Amharic	is	Icelandic	sa	Sanskrit

continued on next page

Table E.2 ISO language codes, as specified by ISO 639-1 (sorted by code) (continued)

Code	Language	Code	Language	Code	Language
ar	Arabic	it	Italian	sd	Sindhi
as	Assamese	iu	Inuktitut (Eskimo)	sg	Sangro
ay	Aymara	ja	Japanese	sh	Serbo-Croatian
az	Azerbaijani	jw	Javanese	si	Singhalese
ba	Bashkir	ka	Georgian	sk	Slovak
be	Byelorussian	kk	Kazakh	sl	Slovenian
bg	Bulgarian	kl	Greenlandic	sm	Samoan
bh	Bihari	km	Cambodian	sn	Shona
bi	Bislama	kn	Kannada	so	Somali
bn	Bengali Bangla	ko	Korean	sq	Albanian
bo	Tibetan	ks	Kashmiri	sr	Serbian
br	Breton	ku	Kurdish	ss	Siswati
ca	Catalan	ky	Kirghiz	st	Sesotho
co	Corsican	la	Latin	su	Sudanese
cs	Czech	ln	Lingala	sv	Swedish
cy	Welsh	lo	Laothian	sw	Swahili
da	Danish	lt	Lithuanian	ta	Tamil
de	German	lv	Latvian Lettish	te	Tegulu
dz	Bhutani	mg	Malagasy	tg	Tajik
el	Greek	mi	Maori	th	Thai
en	English American	mk	Macedonian	ti	Tigrinya
eo	Esperanto	ml	Malayalam	tk	Turkmen
es	Spanish	mn	Mongolian	tl	Tagalog
et	Estonian	mo	Moldavian	tn	Setswana
eu	Basque	mr	Marathi	to	Tonga

continued on next page

Table E.2 ISO language codes, as specified by ISO 639-1 (sorted by code) (continued)

Code	Language	Code	Language	Code	Language
fa	Persian	ms	Malay	tr	Turkish
fi	Finnish	mt	Maltese	ts	Tsonga
fj	Fiji	my	Burmese	tt	Tatar
fo	Faeroese	na	Nauru	tw	Twi
fr	French	ne	Nepali	ug	Uigur
Frisian	nl	Dutch	uk	Ukrainian	
ga	Irish	no	Norwegian	ur	Urdu
gd	Gaelic Scots Gaelic	oc	Occitan	uz	Uzbek
gl	Galician	om	Oromo Afan	vi	Vietnamese
gn	Guarani	or	Oriya	vo	Volapuk
gu	Gujarati	pa	Punjabi	wo	Wolof
ha	Hausa	pl	Polish	xh	Xhosa
he	Hebrew	ps	Pashto Pushto	yi	Yiddish
hi	Hindi	pt	Portuguese	yo	Yoruba
hr	Croatian	qu	Quechua	za	Zhuang
hu	Hungarian	rm	Rhaeto-Romance	zh	Chinese
hy	Armenian	rn	Kirundi	zu	Zulu
ia	Interlingua				

Table E.3 ISO language codes, as specified by ISO 639-1 (sorted by language description)

Language	Code	Language	Code	Language	Code
Abkhazian	ab	Hungarian	hu	Romanian	ro
Afar	aa	Icelandic	is	Russian	ru
Afrikaans	af	Indonesian	id	Samoan	sm
Albanian	sq	Interlingua	ia	Sangro	sg

continued on next page

Table E.3 ISO language codes, as specified by ISO 639-1 (sorted by language description) (continued)

Language	Code	Language	Code	Language	Code
Amharic	am	Interlingue	ie	Sanskrit	sa
Arabic	ar	Inuktitut (Eskimo)	iu	Serbian	sr
Armenian	hy	Inupiaq	ik	Serbo-Croatian	sh
Assamese	as	Irish	ga	Sesotho	st
Aymara	ay	Italian	it	Setswana	tn
Azerbaijani	az	Japanese	ja	Shona	sn
Bashkir	ba	Javanese	jw	Sindhi	sd
Basque	eu	Kannada	kn	Singhalese	si
Bengali Bangla	bn	Kashmiri	ks	Siswati	ss
Bhutani	dz	Kazakh	kk	Slovak	sk
Bihari	bh	Kinyarwanda	rw	Slovenian	sl
Bislama	bi	Kirghiz	ky	Somali	so
Breton	br	Kirundi	rn	Spanish	es
Bulgarian	bg	Korean	ko	Sudanese	su
Burmese	my	Kurdish	ku	Swahili	sw
Byelorussian	be	Laothian	lo	Swedish	sv
Cambodian	km	Latin	la	Tagalog	tl
Catalan	ca	Latvian Lettish	lv	Tajik	tg
Chinese	zh	Lingala	ln	Tamil	ta
Corsican	co	Lithuanian	lt	Tatar	tt
Croatian	hr	Macedonian	mk	Tegulu	te
Czech	cs	Malagasy	mg	Thai	th
Danish	da	Malay	ms	Tibetan	bo
Dutch	nl	Malayalam	ml	Tigrinya	ti
English American	en	Maltese	mt	Tonga	to

continued on next page

Table E.3 ISO language codes, as specified by ISO 639-1 (sorted by language description) (continued)

Language	Code	Language	Code	Language	Code
Esperanto	eo	Maori	mi	Tsonga	ts
Estonian	et	Marathi	mr	Turkish	tr
Faeroese	fo	Moldavian	mo	Turkmen	tk
Fiji	fj	Mongolian	mn	Twi	tw
Finnish	fi	Nauru	na	Uigur	ug
French	fr	Nepali	ne	Ukrainian	uk
Frisian	Norwegian	no	Urdu	ur	
Gaelic Scots Gaelic	gd	Occitan	oc	Uzbek	uz
Galician	gl	Oriya	or	Vietnamese	vi
Georgian	ka	Oromo Afan	om	Volapuk	vo
German	de	Pashto Pushto	ps	Welsh	cy
Greek	el	Persian	fa	Wolof	wo
Greenlandic	kl	Polish	pl	Xhosa	xh
Guarani	gn	Portuguese	pt	Yiddish	yi
Gujarati	gu	Punjabi	pa	Yoruba	yo
Hausa	ha	Quechua	qu	Zhuang	za
Hebrew	he	Rhaeto-Romance	rm	Zulu	zu
Hindi	hi				

E.3 Country codes

When you're localizing a JSF application, you can specify a country code in addition to a language code.

Country codes are uppercase, two-letter strings, defined by the ISO. This allows your application to handle specific dialects of a language, such as Mexican Spanish. The two tables below list the ISO country codes and a description of the country they represent, as of JDK 1.4.2. Table E.4 lists them sorted by country code, and table E.5 lists them sorted by country description.

Table E.4 ISO country codes, as specified by ISO-3166 (sorted by code)

Code	Country	Code	Country	Code	Country
AD	Andorra, Principality of	GM	Gambia	NR	Nauru
AE	United Arab Emirates	GN	Guinea	NU	Niue
AF	Afghanistan, Islamic State of	GP	Guadeloupe	NZ	New Zealand
AG	Antigua and Barbuda	GQ	Equatorial Guinea	OM	Oman
AI	Anguilla	GR	Greece	PA	Panama
AL	Albania	GS	South Georgia and the South Sandwich Islands	PE	Peru
AM	Armenia	GT	Guatemala	PF	French Polynesia
AN	Netherlands Antilles	GU	Guam	PG	Papua New Guinea
AO	Angola	GW	Guinea-Bissau	PH	Philippines
AQ	Antarctica	GY	Guyana	PK	Pakistan
AR	Argentina	HK	Hong Kong	PL	Poland
AS	American Samoa	HM	Heard and McDonald Islands	PM	Saint Pierre and Miquelon
AT	Austria	HN	Honduras	PN	Pitcairn
AU	Australia	HR	Croatia (Hrvatska)	PR	Puerto Rico
AW	Aruba	HT	Haiti	PT	Portugal
AZ	Azerbaijan	HU	Hungary	PW	Palau
BA	Bosnia-Herzegovina	ID	Indonesia	PY	Paraguay
BB	Barbados	IE	Ireland	QA	Qatar
BD	Bangladesh	IL	Israel	RE	Réunion
BE	Belgium	IN	India	RO	Romania
BF	Burkina Faso	IO	British Indian Ocean Territory	RU	Russian Federation
BG	Bulgaria	IQ	Iraq	RW	Rwanda
BH	Bahrain	IR	Iran, Islamic Republic of	SA	Saudi Arabia

continued on next page

APPENDIX E
Time zone, country, language, and currency codes

Table E.4 ISO country codes, as specified by ISO-3166 (sorted by code) (continued)

Code	Country	Code	Country	Code	Country
BI	Burundi	IS	Iceland	SB	Solomon Islands
BJ	Benin	IT	Italy	SC	Seychelles
BM	Bermuda	JM	Jamaica	SD	Sudan
BN	Brunei Darussalam	JO	Jordan	SE	Sweden
BO	Bolivia	JP	Japan	SG	Singapore
BR	Brazil	KE	Kenya	SH	Saint Helena
BS	Bahamas	KG	Kyrgyzstan	SI	Slovenia
BT	Bhutan	KH	Cambodia	SJ	Svalbard and Jan Mayen Islands
BV	Bouvet Island	KI	Kiribati	SK	Slovakia
BW	Botswana	KM	Comoros	SL	Sierra Leone
BY	Belarus	KN	Saint Kitts and Nevis	SM	San Marino
BZ	Belize	KP	Korea, Democratic People's Republic of (N. Korea)	SN	Senegal
CA	Canada	KR	Korea, Republic of (S. Korea)	SO	Somalia
CC	Cocos (Keeling) Islands	KW	Kuwait	SR	Suriname
CF	Central African Republic	KY	Cayman Islands	ST	Sao Tome and Principe
CG	Congo, People's Republic of	KZ	Kazakhstan	SV	El Salvador
CH	Switzerland	LA	Laos	SY	Syrian Arab Republic
CI	Cote D'Ivoire (Ivory Coast)	LB	Lebanon	SZ	Swaziland
CK	Cook Islands	LC	Saint Lucia	TC	Turks and Caicos Islands
CL	Chile	LI	Liechtenstein	TD	Chad
CM	Cameroon	LK	Sri Lanka	TF	French Southern Territories

continued on next page

Table E.4 ISO country codes, as specified by ISO-3166 (sorted by code) (continued)

Code	Country	Code	Country	Code	Country
CN	China	LR	Liberia	TG	Togo
CO	Colombia	LS	Lesotho	TH	Thailand
CR	Costa Rica	LT	Lithuania	TJ	Tajikistan
CU	Cuba	LU	Luxembourg	TK	Tokelau
CV	Cape Verde	LV	Latvia	TM	Turkmenistan
CX	Christmas Island	LY	Libyan Arab Jamahiriya	TN	Tunisia
CY	Cyprus	MA	Morocco	TO	Tonga
CZ	Czech Republic	MC	Monaco	TP	East Timor
DE	Germany	MD	Moldova, Republic of	TR	Turkey
DJ	Djibouti	MG	Madagascar	TT	Trinidad and Tobago
DK	Denmark	MH	Marshall Islands	TV	Tuvalu
DM	Dominica	MK	Macedonia	TW	Taiwan, Province of China
DO	Dominican Republic	ML	Mali	TZ	Tanzania, United Republic of
DZ	Algeria	MM	Myanmar	UA	Ukraine
EC	Ecuador	MN	Mongolia	UG	Uganda
EE	Estonia	MO	Macau	UM	US Minor Outlying Islands
EG	Egypt	MP	Northern Mariana Islands	US	United States of America
EH	Western Sahara	MQ	Martinique	UY	Uruguay
ER	Eritrea	MR	Mauritania	UZ	Uzbekistan
ES	Spain	MS	Montserrat	VA	Holy See (Vatican City State)
ET	Ethiopia	MT	Malta	VC	St. Vincent and the Grenadines
FI	Finland	MU	Mauritius	VE	Venezuela
FJ	Fiji	MV	Maldives	VG	Virgin Islands (British)

continued on next page

APPENDIX E
Time zone, country, language, and currency codes

Table E.4 ISO country codes, as specified by ISO-3166 (sorted by code) (continued)

Code	Country	Code	Country	Code	Country
FK	Falkland Islands (Malvinas)	MW	Malawi	VI	Virgin Islands (US)
FM	Micronesia, Federated States	MX	Mexico	VN	Viet Nam
FO	Faroe Islands	MY	Malaysia	VU	Vanuatu
FR	France	MZ	Mozambique	WF	Wallis and Futuna Islands
FX	France, Metropolitan	NA	Namibia	WS	Samoa
GA	Gabon	NC	New Caledonia	YE	Yemen
GB	United Kingdom	NE	Niger	YT	Mayotte
GD	Grenada	NF	Norfolk Island	YU	Yugoslavia
GE	Georgia	NG	Nigeria	ZA	South Africa
GF	French Guiana	NI	Nicaragua	ZM	Zambia
GH	Ghana	NL	Netherlands	ZR	Zaire
GI	Gibraltar	NO	Norway	ZW	Zimbabwe
GL	Greenland	NP	Nepal		

Table E.5 ISO country codes, as specified by ISO-3166 (sorted by country)

Country	Code	Country	Code	Country	Code
Afghanistan, Islamic State of	AF	Germany	DE	Norway	NO
Albania	AL	Ghana	GH	Oman	OM
Algeria	DZ	Gibraltar	GI	Pakistan	PK
American Samoa	AS	Greece	GR	Palau	PW
Andorra, Principality of	AD	Greenland	GL	Panama	PA
Angola	AO	Grenada	GD	Papua New Guinea	PG
Anguilla	AI	Guadeloupe	GP	Paraguay	PY
Antarctica	AQ	Guam	GU	Peru	PE

continued on next page

Table E.5 ISO country codes, as specified by ISO-3166 (sorted by country) (continued)

Country	Code	Country	Code	Country	Code
Antigua and Barbuda	AG	Guatemala	GT	Philippines	PH
Argentina	AR	Guinea	GN	Pitcairn	PN
Armenia	AM	Guinea-Bissau	GW	Poland	PL
Aruba	AW	Guyana	GY	Portugal	PT
Australia	AU	Haiti	HT	Puerto Rico	PR
Austria	AT	Heard and McDonald Islands	HM	Qatar	QA
Azerbaijan	AZ	Holy See (Vatican City State)	VA	Réunion	RE
Bahamas	BS	Honduras	HN	Romania	RO
Bahrain	BH	Hong Kong	HK	Russian Federation	RU
Bangladesh	BD	Hungary	HU	Rwanda	RW
Barbados	BB	Iceland	IS	Saint Helena	SH
Belarus	BY	India	IN	Saint Kitts and Nevis	KN
Belgium	BE	Indonesia	ID	Saint Lucia	LC
Belize	BZ	Iran, Islamic Republic of	IR	Saint Pierre and Miquelon	PM
Benin	BJ	Iraq	IQ	Saint Vincent and the Grenadines	VC
Bermuda	BM	Ireland	IE	Samoa	WS
Bhutan	BT	Israel	IL	San Marino	SM
Bolivia	BO	Italy	IT	Sao Tome and Principe	ST
Bosnia-Herzegovina	BA	Jamaica	JM	Saudi Arabia	SA
Botswana	BW	Japan	JP	Senegal	SN
Bouvet Island	BV	Jordan	JO	Seychelles	SC
Brazil	BR	Kazakhstan	KZ	Sierra Leone	SL
British Indian Ocean Territory	IO	Kenya	KE	Singapore	SG

continued on next page

APPENDIX E
Time zone, country, language, and currency codes

Table E.5 ISO country codes, as specified by ISO-3166 (sorted by country) (continued)

Country	Code	Country	Code	Country	Code
Brunei Darussalam	BN	Kiribati	KI	Slovakia	SK
Bulgaria	BG	Korea, Democratic People's Republic of (N. Korea)	KP	Slovenia	SI
Burkina Faso	BF	Korea, Republic of (S. Korea)	KR	Solomon Islands	SB
Burundi	BI	Kuwait	KW	Somalia	SO
Cambodia	KH	Kyrgyzstan	KG	South Africa	ZA
Cameroon	CM	Laos	LA	South Georgia and the South Sandwich Islands	GS
Canada	CA	Latvia	LV	Spain	ES
Cape Verde	CV	Lebanon	LB	Sri Lanka	LK
Cayman Islands	KY	Lesotho	LS	Sudan	SD
Central African Republic	CF	Liberia	LR	Suriname	SR
Chad	TD	Libyan Arab Jamahiriya	LY	Svalbard and Jan Mayen Islands	SJ
Chile	CL	Liechtenstein	LI	Swaziland	SZ
China	CN	Lithuania	LT	Sweden	SE
Christmas Island	CX	Luxembourg	LU	Switzerland	CH
Cocos (Keeling) Islands	CC	Macau	MO	Syrian Arab Republic	SY
Colombia	CO	Macedonia	MK	Taiwan, Province of China	TW
Comoros	KM	Madagascar	MG	Tajikistan	TJ
Congo, People's Republic of	CG	Malawi	MW	Tanzania, United Republic of	TZ
Cook Islands	CK	Malaysia	MY	Thailand	TH
Costa Rica	CR	Maldives	MV	Togo	TG
Cote D'Ivoire (Ivory Coast)	CI	Mali	ML	Tokelau	TK

continued on next page

Table E.5 ISO country codes, as specified by ISO-3166 (sorted by country) (continued)

Country	Code	Country	Code	Country	Code
Croatia (Hrvatska)	HR	Malta	MT	Tonga	TO
Cuba	CU	Marshall Islands	MH	Trinidad and Tobago	TT
Cyprus	CY	Martinique	MQ	Tunisia	TN
Czech Republic	CZ	Mauritania	MR	Turkey	TR
Denmark	DK	Mauritius	MU	Turkmenistan	TM
Djibouti	DJ	Mayotte	YT	Turks and Caicos Islands	TC
Dominica	DM	Mexico	MX	Tuvalu	TV
Dominican Republic	DO	Micronesia, Federated States	FM	Uganda	UG
East Timor	TP	Moldova, Republic of	MD	Ukraine	UA
Ecuador	EC	Monaco	MC	United Arab Emirates	AE
Egypt	EG	Mongolia	MN	United Kingdom (Great Britain)	GB
El Salvador	SV	Montserrat	MS	US Minor Outlying Islands	UM
Equatorial Guinea	GQ	Morocco	MA	United States of America	US
Eritrea	ER	Mozambique	MZ	Uruguay	UY
Estonia	EE	Myanmar	MM	Uzbekistan	UZ
Ethiopia	ET	Namibia	NA	Vanuatu	VU
Falkland Islands (Malvinas)	FK	Nauru	NR	Venezuela	VE
Faroe Islands	FO	Nepal	NP	Viet Nam	VN
Fiji	FJ	Netherlands	NL	Virgin Islands (British)	VG
Finland	FI	Netherlands Antilles	AN	Virgin Islands (US)	VI
France	FR	New Caledonia	NC	Wallis and Futuna Islands	WF
France, Metropolitan	FX	New Zealand	NZ	Western Sahara	EH
French Guiana	GF	Nicaragua	NI	Yemen	YE
French Polynesia	PF	Niger	NE	Yugoslavia	YU

continued on next page

Table E.5 ISO country codes, as specified by ISO-3166 (sorted by country) (continued)

Country	Code	Country	Code	Country	Code
French Southern Territories	TF	Nigeria	NG	Zaire	ZR
Gabon	GA	Niue	NU	Zambia	ZM
Gambia	GM	Norfolk Island	NF	Zimbabwe	ZW
Georgia	GE	Northern Mariana Islands	MP		

E.4 Currency codes

When you're dealing with monetary values, you sometimes need to specify the specific type of currency. Because words like "dollar" aren't terribly specific (several countries call their currency "dollar"), the ISO has defined capitalized three-letter strings that represent each currency. These are listed in table E.6 (as of JDK 1.4.2). Because some of the currencies have been replaced by other codes, or represent something besides an actual currency (e.g., precious metals or funds), there are additional columns of information for each code.

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code)

Code	Description	Notes	Discontinued?
ADP	Andorran Peseta	Euro Currency	Yes
AED	United Arab Emirates Dirham		
AFA	Afghanistan Afghani	Replaced by AFN	Yes
ALL	Albanian Lek		
AMD	Armenian Dram		
ANG	Netherlands Antillian Guilder		
AOA	Angolan Kwanza		
ARS	Argentine Peso		
ATS	Austrian Schilling	Euro Currency	Yes
AUD	Australian Dollar		

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
AWG	Aruban Guilder		
AZM	Azerbaijani Manat		
BAM	Bosnia-Herzegovina Convertible Marks		
BBD	Barbados Dollar		
BDT	Bangladesh Taka		
BEF	Belgian Franc	Euro Currency	Yes
BGL	Bulgarian Lev	Before 05-Jul-1999	Yes
BGN	Bulgarian Lev	Since 05-Jul-1999	
BHD	Bahraini Dinar		
BIF	Burundi Franc		
BMD	Bermudian Dollar		
BND	Brunei Dollar		
BOB	Bolivian Boliviano		
BOV	Bolivian Mvdol	Funds Code	
BRL	Brazilian Real		
BSD	Bahamian Dollar		
BTN	Bhutan Ngultrum		
BWP	Botswana Pula		
BYB	Belarussian Ruble		
BYR	Belarussian Ruble		
BZD	Belize Dollar		
CAD	Canadian Dollar		
CDF	Congolese Franc		
CHF	Swiss Franc		
CLF	Chilean Unidades de Fomento	Funds Code	

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
CLP	Chilean Peso		
CNY	Chinese Renminbi		
COP	Colombian Peso		
CRC	Costa Rican Colón		
CUP	Cuban Peso		
CVE	Cape Verde Escudo		
CYP	Cypriot Pound		
CZK	Czech Koruna		
DEM	Deutsche Mark	Euro Currency	Yes
DJF	Djibouti Franc		
DKK	Danish Krone		
DOP	Dominican Republic Peso		
DZD	Algerian Dinar		
EEK	Estonian Kroon		
EGP	Egyptian Pound		
ERN	Eritrean Nakfa		
ESP	Spanish Peseta	Euro Currency	Yes
ETB	Ethiopian Birr		
EUR	Euro		
FIM	Finnish Markka	Euro Currency	Yes
FJD	Fiji Dollar		
FKP	Falkland Pound		
FRF	French Franc	Euro Currency	Yes
GBP	British Pound Sterling		
GEL	Georgian Lari		

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
GHC	Ghana Cedi		
GIP	Gibraltar Pound		
GMD	Gambian Dalasi		
GNF	Guinean Franc		
GRD	Greek Drachma	Euro Currency	Yes
GTQ	Guatemalan Quetzal		
GWP	Guinea-Bissau Peso		
GYD	Guyana Dollar		
HKD	Hong Kong Dollar		
HNL	Honduran Lempira		
HRK	Croatian Kuna		
HTG	Haitian Gourde		
HUF	Hungarian Forint		
IDR	Indonesian Rupiah		
IEP	Irish Punt	Euro Currency	Yes
ILS	New Israeli Shekel		
INR	Indian Rupee		
IQD	Iraqi Dinar		
IRR	Iranian Rial		
ISK	Icelandic Króna		
ITL	Italian Lira	Euro Currency	Yes
JMD	Jamaican Dollar		
JOD	Jordanian Dinar		
JPY	Japanese Yen		
KES	Kenyan Shilling		

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
KGS	Kyrgyzstan Som		
KHR	Cambodian Riel		
KMF	Comorian Franc		
KPW	North Korean Won		
KRW	South Korean Won		
KWD	Kuwaiti Dinar		
KYD	Cayman Islands Dollar		
KZT	Kazakhstan Tenge		
LAK	Laotian Kip		
LBP	Lebanese Pound		
LKR	Sri Lankan Rupee		
LRD	Liberian Dollar		
LSL	Lesotho Loti		
LTL	Lithuanian Litus		
LUF	Luxembourg Franc	Euro Currency	Yes
LVL	Latvian Lat		
LYD	Libyan Dinar		
MAD	Moroccan Dirham		
MDL	Moldovian Leu		
MGF	Malagasy Franc		
MKD	Macedonian Denar		
MMK	Myanmar Kyat		
MNT	Mongolian Tugrik		
MOP	Macau Pataca		
MRO	Mauritania Ouguiya		

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
MTL	Maltese Lira		
MUR	Mauritius Rupee		
MVR	Maldives Rufiyaa		
MWK	Malawian Kwacha		
MXN	Mexican Peso		
MXV	Mexican Unidad de Inversion (UDI)	Funds Code	
MYR	Malaysian Ringgit		
MZM	Mozambique Metical		
NAD	Namibian Dollar		
NGN	Nigerian Naira		
NIO	Nicaraguan Córdoba Oro		
NLG	Dutch Guilder	Euro Currency	Yes
NOK	Norwegian Krone		
NPR	Nepalese Rupee		
NZD	New Zealand Dollar		
OMR	Omani Rial		
PAB	Panama Balboa		
PEN	Peruvian New Sol		
PGK	Papua New Guinea Kina		
PHP	Philippine Peso		
PKR	Pakistani Rupee		
PLN	Polish New Zloty		
PTE	Portuguese Escudo	Euro Currency	Yes
PYG	Paraguay Guarani		
QAR	Qatari Rial		

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
ROL	Romanian Leu		
RUB	Russian Ruble		
RUR	Russian Ruble	Replaced by RUB	Yes
RWF	Rwanda Franc		
SAR	Saudi Riyal		
SBD	Solomon Islands Dollar		
SCR	Seychelles Rupee		
SDD	Sudanese Dinar		
SEK	Swedish Krona		
SGD	Singapore Dollar		
SHP	St. Helena Pound		
SIT	Slovenian Tolar		
SKK	Slovak Koruna		
SLL	Sierra Leonean Leone		
SOS	Somali Shilling		
SRG	Suriname Guilder	Replaced by SRD, which is not supported by JDK 1.4.2.	Yes
STD	São Tomé and Príncipe Dobra		
SVC	El Salvadorian Colón		
SYP	Syrian Pound		
SZL	Swaziland Lilangeni		
THB	Thai Baht		
TJS	Tajikistani Somoni		
TMM	Turkmenistani Manat		
TND	Tunisian Dinar		
TOP	Tongan Pa'anga		

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
TPE	Timorian Escudo		
TRL	Turkish Lira		
TTD	Trinidad and Tobago Dollar		
TWD	New Taiwanese Dollar		
TZS	Tanzanian Shilling		
UAH	Ukrainian Hryvnia		
UGX	Ugandan Shilling		
USD	United States Dollar		
USN	United States Dollar	Funds Code—Next Day	
USS	United States Dollar	Funds Code—Same Day	Yes
UYU	Uruguayan Peso		
UZS	Uzbekistani Sum		
VEB	Venezuelan Bolivar		
VND	Viet Nam Dong		
VUV	Vanuatu Vatu		
WST	Samoan Tala		
XAF	Communauté Financière Africaine Francs		
XAG	Silver Troy Ounce	Precious Metal	
XAU	Gold Troy Ounce	Precious Metal	
XBA	European Composite Unit	Bonds Market Unit—EURCO	
XBB	European Monetary Unit	Bonds Market Unit—E.M.U.-6	
XBC	European Unit of Account 9	Bonds Market Unit—E.U.A.-9	
XBD	European Unit of Account 17	Bonds Market Unit—E.U.A.-17	
XCD	East Caribbean Dollar		
XDR	Special Drawing Rights	International Monetary Fund	

continued on next page

Table E.6 ISO currency codes, as specified by ISO 4217 (sorted by code) (continued)

Code	Description	Notes	Discontinued?
XFO	Gold-Franc	Replaced by XDR	Yes
XFU	UIC Franc	Special settlement currency	
XOF	West African Franc		
XPD	Palladium Troy Ounce	Precious Metal	
XPF	CFP Franc		
XPT	Platinum Troy Ounce	Precious Metal	
XTS	N/A	Reserved for testing purposes	
XXX	N/A	No Currency	
YER	Yemeni Rial		
YUM	New Yugoslavian Dinar		
ZAR	South African Rand		
ZMK	Zambian Kwacha		
ZWD	Zimbabwean Dollar		

references

- [Alur, Crupi, Malks] Alur, Deepak, John Crupi, and Dan Malks. 2003. *Core J2EE Patterns*. Upper Saddle River, NJ: Prentice Hall.
- [ASF, Cocoon] Apache Software Foundation. “Cocoon XML web application framework.” <http://cocoon.apache.org>.
- [ASF, Digester] Apache Software Foundation. “Jakarta Commons Digester (reads XML files into Java objects).” <http://jakarta.apache.org/commons/digester>.
- [ASF, Jelly] Apache Software Foundation. “Jakarta Commons Jelly Java and XML-based scripting and processing engine.” <http://jakarta.apache.org/commons/jelly>.
- [ASF, Struts] Apache Software Foundation. “Struts web application framework.” <http://struts.apache.org>.
- [ASF, Struts-Faces] Apache Software Foundation. “Struts-Faces integration library.” <http://cvs.apache.org/builds/jakarta-struts/nightly/struts-faces>.
- [ASF, Tapestry] Apache Software Foundation. “Tapestry web application framework.” <http://jakarta.apache.org/tapestry/index.html>.
- [ASF, Tiles] Apache Software Foundation. “Tiles JSP templating framework.” http://jakarta.apache.org/struts/userGuide/dev_tiles.html.
- [ASF, Tomcat] Apache Software Foundation. “Tomcat web container.” <http://jakarta.apache.org/tomcat/index.html>.
- [ASF, Velocity] Apache Software Foundation. “Velocity template engine.” <http://jakarta.apache.org/velocity/index.html>.

- [Barcia Series 2004] Barcia, Roland. 2004. "Developing JSF Applications Using WebSphere Studio V5.1.1 (5-part series)." http://www-106.ibm.com/developerworks/websphere/techjournal/0401_barcia/barcia.html.
- [Bayern] Bayern, Shawn. 2002. *JSTL in Action*. Greenwich, CT: Manning.
- [BEA, WebLogic] BEA. "WebLogic J2EE application server." <http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/server>.
- [Bergsten] Bergsten, Hans. 2004. "Improving JSF by Dumping JSP." <http://www.onjava.com/pub/a/onjava/2004/06/09/jsf.html>.
- [Friedl] Friedl, Jeffrey E. F. 2002. *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly & Associates, Inc.
- [Fowler, Dependency Injection] Fowler, Martin. 2004. "Inversion of Control Containers and the Dependency Injection Pattern." <http://www.martinfowler.com/articles/injection.html>.
- [Fowler, Enterprise] Fowler, Martin. 2003. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley.
- [GoF] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- [Grand] Grand, Mark. 1998. *Patterns in Java, Vol. 1*. New York: John Wiley & Sons.
- [Groovy] "Groovy Java scripting language." <http://groovy.codehaus.org>.
- [Hunter] Hunter, Jason. 2001. *Java Servlet Programming*. Sebastopol, CA: O'Reilly & Associates.
- [Husted] Husted, Ted. 2003. *Struts in Action*. Greenwich, CT: Manning.
- [Holmes] Holmes, James. "Faces Console JSF configuration editor." <http://www.jamesholmes.com/JavaServerFaces/console>.
- [IBM, WAS] IBM. "WebSphere Application Server." <http://www-306.ibm.com/software/info1/websphere/index.jsp?tab=products/appserv>.
- [IBM, WSAD] IBM. "WebSphere Studio Application Developer J2EE IDE." <http://www-306.ibm.com/software/awdtools/studioappdev>.
- [Informa] "Informa Java RSS library." <http://informa.sourceforge.net>.
- [JBoss, Hibernate] JBoss. "Hibernate object/relational persistence and query engine." <http://www.hibernate.org>.
- [Jython] "Jython Java-based Python implementation." <http://www.jython.org>.
- [JSF Central] "JSF Central JavaServer Faces community and FAQ." <http://www.jsfcentral.com>.
- [JSR 227] "Java Specification Request 227: A Standard Data Binding & Data Access Facility for J2EE." <http://www.jcp.org/en/jsr/detail?id=227>.
- [Keel] "Keel meta-framework for server-side applications." <http://www.keelframework.org>.
- [Kobrix] "Kobrix Software. Tag Interface Component Library." <http://www.kobrix.com>.
- [Microsoft, ASP.NET] Microsoft. "ASP.NET web application framework." <http://www.asp.net>.

- [MyFaces] “MyFaces open source JSF implementation.” <http://www.myfaces.org>.
- [Nash] Nash, Michael. 2004. “Spinning Your Code with XSLT and JSF in Cocoon.” http://www.developer.com/lang/article.php/10924_3348311_1.
- [New Atlanta, ServletExec] New Atlanta Communications. “ServletExec web container.” <http://www.newatlanta.com/products/servletexec/index.jsp>.
- [OpenSymphony, SiteMesh] OpenSymphony. “SiteMesh web-page layout and decorating framework.” <http://www.opensymphony.com/sitemesh>.
- [OpenSymphony, WebWork] OpenSymphony. “WebWork web application framework.” <http://www.opensymphony.com/webwork>.
- [Oracle, ADF] Oracle. “Application Development Framework.” <http://otn.oracle.com/products/jdev/index.html>.
- [Oracle, ADF UIX] Oracle. “ADF UIX components.” http://otn.oracle.com/products/jdev/collateral/papers/9.0.5.0/adfuix_roadmap/adfuix_roadmap.html.
- [Oracle, AS] Oracle. “Oracle Application Server.” <http://otn.oracle.com/products/ias/index.html>.
- [Oracle, JDeveloper] Oracle. “JDeveloper J2EE IDE.” <http://otn.oracle.com/products/jdev/index.html>.
- [Salmon, SOFIA] Salmon. “Salmon Open Framework for Internet Applications.” <http://www.salmonllc.com/website/Jsp/vanity/Sofia.jsp>.
- [Schalk] Schalk, Chris. 2004. “How to Use JSF with JDeveloper 10g.” http://otn.oracle.com/products/jdev/howtos/10g/jsf_howto/jsf.html.
- [Smile] “Smile open source JSF implementation (with Java views).” <http://smile.sourceforge.net>.
- [Spring-Faces] “JSF integration code for Spring (open source).” <http://jsf-spring.sourceforge.net>.
- [Sun, Creator] Sun Microsystems. “Java Studio Creator JSF IDE.” <http://wwws.sun.com/software/products/jscreator/index.html>.
- [Sun, i18n] Sun Microsystems. “Java Tutorial, Internationalization Trail.” <http://java.sun.com/docs/books/tutorial/i18n/index.html>.
- [Sun, JDO] Sun Microsystems. “Java Data Objects specification.” <http://java.sun.com/products/jdo/index.jsp>.
- [Sun, JRL] Sun Microsystems. “Java Research License.” <http://www.java.net/jrl.html>.
- [Sun, JSF Spec] Sun Microsystems. “JavaServer Faces specification.” <http://java.sun.com/j2ee/javaserverfaces>.
- [Sun, JSF RI] Sun Microsystems. “JSF reference implementation.” <http://java.sun.com/j2ee/javaserverfaces>.

- [Sun, JSP] Sun Microsystems. “JavaServer Pages specification.” <http://java.sun.com/products/jsp/index.jsp>.
- [Sun, JSAS] Sun Microsystems. “Java System Application Server.” http://wwws.sun.com/software/products/appsvr/home_appsrvr.html.
- [Sun, JSTL] Sun Microsystems. “JavaServer Pages Template Library.” <http://java.sun.com/products/jsp/jstl/index.jsp>.
- [Sun, Portlet] Sun Microsystems. “Portlet specification.” <http://www.jcp.org/en/jsr/detail?id=162>.
- [Sun, Servlet] Sun Microsystems. “Servlet specification.” <http://java.sun.com/products/servlet/index.jsp>.
- [Syndic8] “Syndic8 RSS and Atom news feed aggregator.” <http://www.syndic8.com>.
- [Szyperski] Szyperski, Clemens. 2002. *Component Software, Beyond Object-Oriented Programming*. New York: Addison-Wesley.
- [TheServerSide] The Middleware Company. “TheServerSide.com enterprise Java community.” <http://www.theserverside.com>.
- [W3Schools] Refsnes Data. “W3Schools web technology tutorial site.” <http://www.w3schools.com>.
- [WebMacro] Semiotek. “WebMacro open source template language.” <http://www.webmacro.org>.
- [XUL] The Mozilla Organization. “XML User Interface Language.” <http://www.mozilla.org/projects/xul/>.

index

Symbols

???<key>??? 243

A

Abstract Factory (GoF) 643
accessibility 42
accessor methods (getters) 14, 463
Action class 410
action events
 action sources 51, 129, 433
 execution through
 ActionSource instances 448
 firing by ActionSource
 interface 623
 generated by Command
 family components 187
 generation and firing of 186
 handling 432
 navigation handler and 129
 user command representation
 with 428
action listener classes
 combining with action
 listener methods and action
 methods 188
 registering 188
action listener methods
 action methods compared to
 52
 activity of 28

combining with action
 listener classes and action
 methods 188
 defined 34
of backing beans
 (ProjectTrack) 358
overview 432
registration with an action
 source 433
sorting project lists with 371
typical uses 53
action listeners
 adding with Java Studio
 Creator 926
 adding with WebSphere
 Studio 904
 default 51
 relationship with action
 listeners 66
 types of 51
action methods
 activity of 29
 combining with action
 listener methods and action
 listener classes 188
 defined 34, 51
invocation by the default
 ActionListener 433
navigation cases associated
 with 133
of backing beans
 (ProjectTrack) 358
organized by function 565
overview 433
pages that reference
 (ProjectTrack) 359
reducing dependencies
 between sets of objects
 and 501
referenced by Inbox page
 (ProjectTrack) 377
required class 433
storage or retrieval of objects
 with 478
typical uses 52
updating data store with 355
action source components 186,
 433
ActionEvent class 429, 432
ActionListener interface 938,
 944
 implementing 809
 overview 433
ActionSource interface 623, 803
 overview 448
ActiveX (Microsoft) 21
Adapter pattern 485
adapting objects for use with
 JSF 485
ADF Faces Components
 (Oracle) 875
ADF UIX (Application
 Development Framework),
 Oracle 7, 19, 874–875
advanced extensions
 registration 98

Index entries with page numbers 703 and higher refer to the online extension.

- advanced features configuration 99
- Apache Software Foundation
Cocoon 15, 672
Jelly 672
Struts. *See* Struts (Apache Software Foundation)
Tapestry 19
Tiles. *See* Tiles (Apache Software Foundation)
Tomcat 8, 12, 89, 92
Velocity 15, 672
- APIs (application programming interfaces)
Informa 759, 762
Portlet 410, 422, 424
Servlet 410, 422, 424, 511, 536
- Application class 411, 413, 936
- application configuration (JSF)
attributes of elements 961
basic file structure 94, 959
elements of 98, 960
file specifying parameter 94, 959
location in the META-INF directory 661
location of 959
managed beans in 357
overview 959
pluggable classes 963
segmenting 101, 959
with web.xml 36, 92
- application configuration
categories 98
- application configuration entry
application 36
categories 98
- application configuration files
defined 97
errors 102
locating 101
specifying 101
- Application Development
Framework UIX (ADF), Oracle 7, 19, 874–875
- application environment 357
- application errors 501
- Application instances,
initializing 606
- application layer classes (ProjectTrack) 477
- application layers 457
- application logic 51, 288, 411, 434, 457, 840
- application messages. *See* messages
- application scopes 80
- <application> element 963
- application-related classes (JSF) 411, 413
- applications
access to pages 360
adding Cascading Style Sheets 303
adding JavaScript using pass-through properties 301
client-side JavaScript integration with JSF 301
commands, executing 219
configuration. *See* application configuration, JSF
connecting views to data sources 355
consistency enforcement 236
development approaches 355
enhancing pages 300
ensuring access to backing beans 472
error reporting 422
form-centric development of small 356
internationalization of 235, 398, 424, 551
layered 457
localization of 402
message bundles for 269
messages. *See* messages
migrating over time 572
object-oriented development of large system 356
request and response handling 572
requirements for JSF 289
resource bundles, creating for 238
- simple two-layer 457
- small 458
- splitting into multi-layers 458
- states, encapsulation of 419
- Struts-Faces 573
- text adaptation to locales 236
- text strings for specific locales 238
- UI component development compared to code development for 607
- Apply Request Values phase 69, 428, 622
- Approve a Project page (ProjectTrack)
- access authorization 546
- beans used for 381
- integration with application 382
- navigation rule update 384
- navigation rules for 337, 384
- overview 331
- project updating 522
- purpose of 379
- architectures, application alternatives and consequences 562
- consistency of 459
- layers 457
- ASP.NET Web Forms (Microsoft) 5, 46, 95, 224, 500
- <attribute> element 959, 962
- attributes, JSF
accessing 443
defined 441
properties compared to 441, 610
- authentication 545
- authentication keys (ProjectTrack) 480
- AuthenticationBean backing bean class (ProjectTrack)
action methods of 500
code for 505
initializing and referencing 509
login method code 550

AuthenticationBean backing bean class (*ProjectTrack*)
(continued)
 managed bean configuration 509
 properties of 500
 Visit object creation and storage 550
 authenticationBean scoped variable (*ProjectTrack*)
 associated views 359
 description 358
 header page and 365–366
 Login page and 360–361
 summary 360
 AuthenticationFilter 547, 550
 authorization 545, 547
 filters 547

B

backing beans 39, 48
 action methods implemented by 433, 564
 adapters and property exposure of 460
 advantage of 500
 application logic organized into 411
 as adapters 460
 association with application views 355
 automatic construction of 460
 base class for 494
 basic structure of 509
 binding to a component instance 46
 binding to a component's value 46
 business objects compared to 460
 component wiring to data-sources with 355
 configuring as managed beans 110, 472
 data layer objects treated as black boxes 540

data-source association with 355
 declaring 501
 defined 14, 45
 event listener methods and 358
 event listeners and 186
 exposing as scoped variables 472
 exposing, using ValueBinding objects 473
 form-centric development and 355
 initializing 565
 JavaDocs for 359
 JSF compared to other frameworks 33
 keys 480
 Managed Bean Creation facility and 460, 494
 not showing up 368
 of Project Track 357
 organizing by form 564
 organizing by function 563
 packaging action methods as properties of 501
 properties of 357
 property values, converting to a String 461
 registering methods of single 452
 relationship with Update Model Values phase 66
 retrieval of business objects 459
 serializability 462
 similarity to ASP.NET Web Forms code-behind classes 46
 simple layered architecture and 457
 stored as scoped variables 494
 storing in request scopes 538
 storing in sessions 538
 support for validator methods in 451
 thread safety of 501
 UI component manipulation with 441
 UI component synchronization with Java objects using 6
 validation methods and 245
 writing to be stateless 538
 Barcia, Roland 895
 base classes 441, 627
 JSTL in Action 104
 Bayern, Shawn 104
 Bergsten, Hans 672
 binary data output 422
 bind, defined 46
 blogs 757
 Borland
 C++Builder 5
 Delphi 4–5
 Delphi Visual Component Library 21
 broadcast method 624
 business layers
 access alternative 562
 defined 458
 interaction with views 459
 overview 473
 business logic 457, 563, 599
 business objects
 adapting to work with JSF components 484
 backing beans compared to 460
 exposing through backing beans 472
 properties of 466
 reusable in non-JSF applications 485
 serializability 462
 unit testing of 474
 Button renderer 728
 buttons
 associating model objects with 323
 cancel 448
 components used for header 320
 configured dynamically 321
 creating 219

buttons (*continued*)
 highlighting selected 321
 localizing labels for 244
 navigation rules for header 323
 normal compared to rollover 728
 reset 221, 736
 rollover-enabled 728
 using graphics for 301

C

C++Builder (Borland) 5
 Camino open source browser 670
 cancel buttons 448
 Cascading Style Sheets (CSS) 69
 adding to JSF applications 303
 browsers and 69
 class attribute and 143
 editors for 143
 UI component integration with 138
 CBP (class-based pages) 666
 cell phones 5
 ChannelDataModel class 765
 ChannelIF interface
 DataModel class and 764
 Informa API and 761
 RSS channel representation in 761
 character class 841
 characters, regular expression 841
 checkboxes, creating 198, 205
 child components
 accessing 442, 445
 ensuring unique client identifiers 623
 manipulating 445
 rendering with encoding methods 815
 using standard components with 758
 child-parent relationships 412
 choice formats 163–164

class-based pages (CBP) 666
 classes
 action methods of 433
 adapter 477, 485
 application layer 477
 application-related (JSF) 411
 base 441
 base, for backing beans 494
 context-related (JSF) 411
 converter 416
 core JSF 936
 enumerated type 476
 event 429
 event-handling (JSF) 411
 infrastructure category of core JSF 936
 JSF set of 410
 pluggable category of core JSF 938
 renderer-related 637
 UI component 411–412, 439, 441
 utility 477
 classes, shared 92
 client certificate authentication 546
 client identifiers
 defined 30, 64
 derivation of 611
 referencing components with 73
 rendering of 613
 client-side validation, JSF and 245, 840
 Cocoon (Apache Software Foundation) 15, 672
 combining with value-change listener 187
 combo box, creating 217
 Command family components 187
 CommandRolloverButtonTag
 adding to the tag library 745
 tag validator for 744
 writing a 741
 Common Object Request Broker Architecture (CORBA) 13
 common/lib directory 92
 commons-beanutils.jar files 91
 commons-collections.jar files 92
 commons-digester.jar files 92
 commons-logging.jar files 92
 component identifiers
 referencing in Java code 76
 specifying 442
 using with findComponent 75
 component palettes 145, 246
 component tags, JSF 107, 142, 145
 component trees 445
 <component> element 970
 components
 defined 21
 user interface (UI). *See UI components*
 composite components 759
 com.sun.faces.NUMBER_OF_VIEWS_IN_SESSION 96
 com.sun.faces.validateXml 96
 com.sun.faces.verifyObjects 96
 configuration elements 961
 configuration, JSF
 application. *See application configuration*
 consistency enforcement 236
 constants, storing strings as 478
 constructors, no-argument 460
 consuming feeds 760
 containers 12, 21
 context 419
 context-related classes (JSF)
 access to user request data with 411
 FacesContext class 411
 summary of 419
 controller architecture, JSF 599
 controls. *See UI components*
 convenience wrappers around methods 424
 convertClientId method 642
 Converter 450
 converter classes 416
 converter development
 need for 854
 registration of 461
 writing 488, 654, 854

- Converter interface 654, 856
 <converter> element 973
 converters
 accessing 246
 application configuration files
 for 98, 970
 association with components
 252
 attributes used by 443
 custom 252, 854
 defined 48
 description 39
 development of 840, 854
 for formatting 48
 identifiers for 450
 JSP integration of 658, 840,
 866
 localization 48
 manipulating inside JSF IDEs
 254
 object assignment caution 254
 purpose of 252
 registration of 252, 416, 491,
 657, 865
 relationship with renderers 48
 specifying 253
 standard (JSF) 252, 255, 854
 standard for basic Java types
 253
 states of 620
 third-party vendor 252
 type conversion, handling by
 654
 UI component association
 with single 654
 unit tests for 656
 user input conversion by 252
 ConverterTag class 659
 cookies 12–13, 81
 CORBA (Common Object
 Request Broker
 Architecture) 13
 core tag library 142
 country codes (ISO) 236, 994
 course-grained components 21
 Create a Project page
 (ProjectTrack)
 access authorization 546
 action methods referenced by
 386
 bean used for 386
 CreateProjectBean and 528
 integration with application
 386
 navigation rule update 390
 navigation rules for 390
 overview 341
 project updating 522
 purpose of 379
 CreateProjectBean backing
 bean class (ProjectTrack)
 code analysis 528
 managed bean configuration
 532
 createProjectBean scoped
 variable (ProjectTrack)
 associated views 359
 Create a Project page and
 386
 description 358
 header page and 366
 summary 386
 currency codes (ISO) 1002
 custom tag handlers 607, 627
 custom tag libraries 102, 634
 custom tags 102, 822
 customizer 14
-
- D**
- data format types 163
 data grids. *See* HtmlDataTable
 component
 data layer objects 540
 data model events 41, 53, 428
 data sets, displaying 223
 data sources
 backing beans association
 with 355
 component wiring to 355
 form-centric development
 and 355
 data store 457
 data store logic 457
 data-aware controls 356, 759
 databases 458
 working with 407
 DataModel class 53, 764
 date collection 706
 date format patterns 163, 259
 date input controls 706
 Date objects 251, 708
 DateTime converter
 description 256
 inputting a Date object with
 708
 properties 257
 usage 253, 256
 using with date format
 patterns 259
 DateTime converters 977
 decimal format patterns 266
 declarative 46
 decode method 613, 729, 820
 decoding 43, 636
 delegated implementation
 rendering model 636
 Delphi (Borland) 5, 176, 356
 Delphi Visual Component
 Library (Borland) 21
 Dependency Injection pattern
 112
 deployment descriptors
 (web.xml)
 basic requirements 290
 updating 397
 usage 289
<description> element
 113–114, 135
 design time 21
 development approaches (JSF)
 355, 457
 development tools 145
 direct implementation
 rendering model 636,
 707
 directory structures 289, 360
 display strings 552
 display technologies 15
 JavaServer Pages. *See* JSP
 (JavaServer Pages)
 Velocity 15
 WebMacro 15
 XSLT 15

display technologies, alternative
class-based pages (CBP) 666
other options 671
XUL 670
<display-name> element
113–114, 135
doAfterBody method 633
Document Type Definition
(DTD), JSF configuration
959
documentation for front-end
developers 359
doFilter method 549
doInitBody method 633
DoubleRange validator 248, 250
DynaActionForms, Struts-Faces
support for 581
DynaBeans, Struts-Faces
support for 581
dynamic content 11
dynamic includes 103
dynamic resources 317

E

EAR (enterprise archive) 90
EditableValueHolder interface
438, 451, 621, 708
EJBs (Enterprise JavaBeans) 9,
18–19, 458
EL. *See* expression language (EL)
encodeBegin method 632, 721,
729, 813
encodeChildren method 632,
813
encodeEnd method 632, 813
encodeIcon method 819
encodeItem utility method 816
encoding 43, 636
encoding logic 613
encrypted logins 546
enterprise archive (EAR) 90
Enterprise JavaBeans (EJBs) 9,
18, 458
error messages. *See* messages
error page (ProjectTrack) 390
adding an 396
navigation rule update 397

navigation rules for 397
errors
classes of 501
handling 501
methods for logging 428
reporting to UI 422
serious 504
evaluation expressions 417
event classes 429
event handling 39, 55, 428,
615, 623
event listeners 28, 49
adding to UI components
615
description 186
event handling with 428
handling by
UIComponentBase 623
method signatures for 432,
434, 436, 502
referencing type-safe
properties 566
registering 186, 437, 593
superclasses and 441
event/listener pairs 430
event-handling classes 411
events 49
action. *See* action events
broadcasting of 430
data model. *See* data model
events
generation with UI
components 142
handling 428
important to JSF
development 186
interfaces that fire 623
phase. *See* phase events
registering for later
broadcasting 431
representation by subclasses
430
types of 428
value-change. *See* value-
change events
exceptions, handling 502
expression language (EL), JSF
and quotes 78
associating backing beans
with components via 46
embedding expressions 78
evaluation of 411, 417
hooking objects to
components 357
implicit variables 83
in relation to backing beans 78
in relation to JavaBeans 78
in relation to JSP 2.0 EL 76,
81
in relation to JSTL EL 76, 81
in relation to model objects 78
JSF application development
and 86
JSP 2.0 EL and 27
managed beans used with 107
method-binding expressions.
See method-binding
expressions
nested properties 78
relationship with Update
Model Values phase 66
sharing variables with custom
tags 106
usage in JSF applications 76
using with components 86
value-binding expressions. *See*
value-binding expressions
expressions, JSF EL. *See*
expression language (EL),
JSF
expressions, regular. *See* regular
expressions
eXtensible Markup Language
(XML) 15, 148
eXtensible Style Sheet
Language Transformations
(XSLT) 15
extension mapping 93
external environments, access to
424
ExternalContext 420, 424

F

<f:verbatim> element
<from-action> element 133

- <f:verbatim> element
(continued)
- <from-view-id> element 132, 134
- nesting custom tags within 105
- Faces Console (James Holmes) 97, 959
- Faces requests 571
- Faces responses 571
- Faces servlet 30, 57, 93
- .faces suffix 93
- faces/prefix 93, 290
- faces-config.xml. *See* application configuration (JSF)
- FacesContext class 82, 420, 936
- FacesEvent 429–430
- FacesMessage 420, 422
- <facet> element 959, 963
- facets
- compared to child components 143
 - defined 42
 - UIComponent methods for 445
 - using HtmlPanelGroup inside 183
- factories (JSF) 936
- factory classes 436
- factory configuration 99
- factory methods 414–415
- <factory> element 975
- FactoryFinder class 936
- families, UI component
- defined 141
 - exposure of 611
 - list of 141
 - renderer types and corresponding 637
 - renderers and 612
- feeds
- caching 764
 - consuming 760
 - RSS 759
- filters, authentication 360
- fine-grained 61
- Firefox open source browser 670
- formatting 48
- form-based authentication 546
- form-based development 355–356, 457
- forms
- adding to ProjectTrack 295
 - components for submitting 320
 - creating input 331
 - creating input. *See also* HtmlForm component
- foundation frameworks 19, 570
- foundation technologies 10
- frameworks, web development 18
- Application Development Framework UIX 7, 19, 146
 - Cocoon 15
 - JSF and 19
 - request processing lifecycles of 571
 - SOFIA 19
 - Struts. *See* Struts
 - Tapestry 19
 - types of 18, 570
 - WebWork 17–18
- front-end development 457, 501
-
- G**
- getAsObject method 655
- getAsString method 655
- getClientId method 611
- getter methods (accessors) 14, 463
- GMT (Greenwich Mean Time) 977
- GridLayout component (Swing) 176
- Groovy scripting language 672
-
- H**
- header page (ProjectTrack)
- action methods referenced by 369
 - AuthenticationBean class and 505
 - backing beans for 365
- CreateProjectBean and 528
- integration with application 366
- internationalizing 400
- navigation rule update 369
- navigation rules for 369
- properties of 366
- purpose of 365
- headers
- built as dynamic resource 317
 - button 320, 323
 - control of text and graphic links in 318
 - navigation 317
 - spacing of buttons in 318
 - using custom components for 321
- HeadlineViewerTableTag 781
- Hello, world! application
- backing bean referenced by 32
 - description 22
 - goodbye page 31
 - main page 24
- hidden fields, declaring 197
- Holmes, James 97, 325, 959
- HTML component subclasses 439
- HTML JSP custom tag library (JSF) 27, 102, 142
- HTML renderers 441
- HTML templates 672
- HTML views, components for building 139
- HtmlBaseTag class 738
- HtmlCommandButton component
- action events generated by 187
 - buttons, declaring with 219
 - description 139
 - for form submission 321
 - summary 219
- UICommand class and 439
- using for Submit button in forms 295, 298
- value-binding expressions and 301

- HtmlCommandLink component**
 action events generated by 187
 action links, creating with 221
 column headings and 371
 description 139
 for header buttons 320
 functions performed by 370
 re-sorting table data with 328
 summary 221
UICommand class and 439
 using instead of
 HtmlCommandButton 301
- HtmlDataTable component**
 association requirement 324
 converting from
 HtmlPanelGrid 371, 391
 data grids 224, 370
 data sets, displaying with 224
 data tables
 creating blank headings,
 component used for 328
 prototyping with panels 324
 re-sorting data, component used for 328
 spanning columns of 335
 description 139
 displaying dynamic lists of data with 324
 summary 225
UIData class and 439
- HtmlForm component**
 basic usage 191
 description 139, 190, 192
 <hidden> field 300
 method attribute of 192
 requirement for header buttons 320
 summary 190
UIForm class and 439
 using for Login page 295, 297
- HtmlGraphicImage component**
 basic usage 169
 description 139
 displaying images with 168
 for header buttons 320
 summary 168
- UIGraphic class** and 439
 URL rewriting and 168
 use in ProjectTrack 292
- HtmlInputHidden component**
 basic usage 198
 converter support by 252
 description 139, 192
 hidden fields, declaring with 197
 summary 197
UIInput class and 439
- HtmlInputSecret component**
 basic usage 196
 converter support by 252
 description 139
 password fields, displaying with 195
 summary 195
UIInput class and 439
 using for password in forms 295
- HtmlInputText component**
 basic usage 193
 converter support by 252
 description 139, 192
 registering email validators 247
 specifying input fields with 345
 summary 193
 text fields, declaring with 193
UIInput class and 439
 using for username in forms 295, 297
 value-change events generated by 187
- HtmlInputTextarea component**
 converter support by 252
 description 139, 192
 memo fields, creating with 194
 summary 194
UIInput class and 439
- HtmlMessage component**
 basic usage 175
 debugging with 175
 description 139
 displaying application messages with 173
- displaying component messages with 169
 showing detail with styles 172
 summary 170, 173
UIMessage class and 439
 validation messages displayed with 245
- HtmlOutputFormat component**
 compared to
 HtmlOutputText 160
 converter support by 252
 description 140
 message format patterns and 161
 parameterizing strings with 243
 relationship with
 MessageFormat class 162
 simple parameter substitution 162
 summary 161
UIOutput class and 440
 using choice format for plurals 165
 using date format patterns with 260
 using extended syntax for message format elements 163
 using for parameterized text 160
 using with number format patterns 266
- HtmlOutputLabel component**
 basic usage 160
 converter support by 252
 creating input labels with 158
 description 140
 for form submission 321
 summary 159
UIOutput class and 440
- HtmlOutputLink component**
 and URL rewriting 165
 compared to normal hyperlinks 167
 converter support by 252
 description 140
 displaying hyperlinks with 165

- HtmlOutputLink component**
(continued)
- linking to relative URL 166
 - passing URL parameters to 167
 - summary 166
 - UIOutput class and 440
- HtmlOutputText component**
- compared to
 - HtmlOutputFormat 160
 - converter support by 252
 - description 140
 - displaying ordinary text with 153
 - for header buttons 320
 - for text in headers 318
 - referencing localized strings with 242
 - summary 154
 - turning off text escaping 155
 - UIOutput class and 440
 - use in ProjectTrack 292
 - used as placeholder for
 - application messages 328
- HtmlPanelGrid component**
- as main container for headers 318, 320
 - basic usage 180
 - converting to HtmlDataTable 371, 391
 - creating footers with 333
 - creating layouts with 333
 - creating tables with 178
 - description 140, 176
 - GridLayout compared to 176
 - improving layouts with 308
 - prototyping data tables with 324
 - simulation of HtmlDataTable 328
 - summary 179
 - UIPanel class and 440
 - using with headers, footers, and styles 181
- HtmlPanelGroup component**
- as a placeholder 177
 - as container for combo box in headers 318
- basic usage** 177
- creating blank table headings**
- with 328
- description** 140, 176
- grouping components** with 176
- summary** 177
- UIPanel class** and 440
- using with styles** 178
- HtmlSelectBooleanCheckbox component**
- checkboxes, creating with 198
 - converter support by 253
 - description 140
 - summary 198
 - UISelectBoolean class and 440
- HtmlSelectManyCheckbox component**
- checkbox groups, displaying items with 205, 336
 - converter support by 253
 - description 141
 - specifying input fields with 345
 - summary 206
 - UISelectMany class and 440
 - value-change events generated by 187
- HtmlSelectManyListbox component**
- converter support by 253
 - description 141
 - listboxes, displaying several items in 208
 - summary 208
 - UISelectMany class and 440
- HtmlSelectManyMenu component**
- converter support by 253
 - description 141
 - listboxes, displaying single items in 210
 - summary 210
 - UISelectMany class and 440
- HtmlSelectOneListbox component**
- converter support by 253
 - description 141
 - single-select listboxes, using with 215
 - specifying input fields with 345
 - summary 215
 - UISelectOne class and 440
- HtmlSelectOneMenu component**
- combo boxes, declaring with 217
 - converter support by 253
 - description 141
 - summary 217
 - UISelectOne class and 440
- HtmlSelectOneRadio component**
- radio buttons, displaying items with 212
 - summary 213
 - UISelectOne class and 440
- HTTP** 12
- HTTP basic authentication** 546
- HTTP digest authentication** 546
- HttpRequest** 410
- HttpResponse** 410
- HttpServlet** 410
- HttpSession** 410
- Husted, Ted** 573
-
- I**
- i18n (internationalization)**
- abbreviation 236
- IBM**
- WebSphere Application Developer. *See WebSphere Application Developer (IBM)*
 - WebSphere Application Server 8, 12, 89
- <icon> element** 113–114, 135, 959, 961

IDEs (Integrated Development Environments). *See* Integrated Development Environments (IDEs) for JSF
 image files 289
 implementations, JSF 89, 92, 95, 936
 implicit includes 103
 implicit variables 83, 581
 Inbox page (ProjectTrack)
 access authorization 546
 action methods referenced by 378
 data table for 325
 directory location 360
 integration with application 375
 listing projects for 511
 navigation rule for 329, 378, 519
 navigation rule update 378
 overview 325
 project information listing 541
 SelectProjectBean and 519
 toolbar button disabling for 464
 inboxBean scoped variable (ProjectTrack)
 Inbox page and 370
 summary 370
 includes, JSP
 directory of 360
 functions performed by 382
 used by project pages 379
 using with JSF 103
 Informa API 759, 762
 infrastructure JSF classes 936
 initial request 63
 input controls 194
 date 706
 disabling 194
 EditableValueHolder interface and 621
 generation of value-change events 186
 registration of validators for 246

requirements for 192
 types of values of 621
 UIInput class and 708
 validation method
 associations 245
 validator acceptance by 248
 validators and 648
 input errors 501
 Input family components 192, 249
 input forms 379
 input validation. *See* validation
 input values, requirement of 249
 InputDateTag 718
 Integrated Development Environments (IDEs) for JSF
 component palette of 246
 importing custom components 627
 importing custom renderers 647
 Java Studio Creator. *See* Java Studio Creator (Sun)
 JDeveloper. *See* JDeveloper (Oracle)
 JSF support of 874
 manipulating converters inside 254
 WebSphere Studio Application Developer. *See* WebSphere Studio (IBM)
 integration
 ad hoc performance of 359
 defined 569
 effect of filtering on 360
 JSF with Struts applications 572
 integration layers 458, 476
 Intercepting Filter pattern 547
 International Organization of Standardization (ISO). *See* ISO (International Organization of Standardization)
 internationalization 235
 application 398
 defined 235, 398
 dynamic text and 244
 i18n abbreviation 236
 informational resources on 241
 supporting in code 552
 UI extensions 660
 Inversion of Control (IoC) pattern 112
 Invoke Application phase 69
 ISO (International Organization of Standardization)
 country codes 994
 currency codes 1002
 language codes 990
 isTrue method 721
 isValid method 650, 745
 items
 creating single 200, 453
 defining lists of 199
 displaying in checkboxes 205
 displaying in listboxes 208
 displaying single 200
 groups of 199, 453
 handling multiple 205
 lists of 199, 453

J

J2EE. *See* Java 2, Enterprise Edition (J2EE)
 Jacobi, Jonas 874
 JAR files 660
 JAR libraries 90
 Java 2, Enterprise Edition (J2EE)
 containers 10
 implementations shipped with 90
 JDeveloper (Oracle) and 874
 JSF applications and 89
 JSF as part of 4, 19
 manipulation of components during runtime 21
 web applications 12, 36
 Java Community Process (JCP) 4, 10, 13, 19
 Java Studio Creator (Sun)
 adding data sources to backing beans 356

- Java Studio Creator (Sun)
(continued)
 description 8
 displaying hyperlinks with 165
 dockable component palette
 of 146
 overview 918
 page navigation editing in 130
 ProjectTrack Login page,
 creating with 920
 registering validators with 246
 table creation with 178
 visual editors for 97
- Java System Application Server
 (Sun) 12
- JavaBeans
 accessing properties of 463
 accessor methods (getters)
 14, 463
 adapters 460
- JavaDocs 359
- JavaScript
 component referencing with 27
 encapsulating functionality
 within components 728
 image rollover function 301
 integrating JSF applications
 with client-side 301
 JSF expression language and 76
 MyFaces configuration
 parameters and 96
 UI component integration
 with 138
- JavaServer Faces (JSF)
 architecture, extending core
 features of 940
 as standard Java web
 applications 289
 benefits of 438
 core classes 936
 defined 5
 display technologies,
 handling 666
 EL syntax, flexibility of 357
 goals of 874
 history of 10
 IDE support of 874
 IDEs and 19
- implementations 171
 industry support 10
 integration with non-Struts
 applications 600
 Java Studio Creator used with
 918
 JSP 1.2 and 607
 JSP-based pages and JSF tag
 library 294
 key concepts 41, 57
 libraries 91
 limitations in functionality 876
 main parts of 569
 non-JSP display technologies
 with 666
 Oracle JDeveloper 10g used
 with 874
 other web frameworks and 19
 pluggable architecture of
 666, 936
 portability of 147
 power of 606
 purpose of 5, 570–571
 RAD and 10
 requirements for applications
 289
 Struts compared to 564
 Struts integration with 573
 terminology 666
 two primary features of 570
 underlying technologies 11
 using without JSP 570
 WebSphere Studio used with
 895
 when to integrate with other
 frameworks 569
- JavaServer Pages (JSP). *See* JSP
 (JavaServer Pages)
- JavaServer Pages Standard Tag
 Library (JSTL)
 API classes 92
 <c:out> tag compared to
 HtmlOutputText
 component 155
 constraints when using JSF
 tags with 109
 controlling component
 visibility with 108
- custom tags and 15
 dependence of JSF on 92
 dynamic includes and 103
 impact of <fmt setLocale>
 tag on 110
 information about 105
 JSF combined with 110
 JSF custom tags
 demonstration with 104
 managed bean referencing
 and 111
 mixing with JSF tags 80,
 104–105
 using JSF tags without 109
 using with JSF and backing
 beans 106
- JavaServer Pages Standard
 Template Library (JSTL)
 104
- javax.faces.CONFIG_FILES 94,
 101, 959
- javax.faces.DEFAULT_SUFFIX
 94
- javax.faces.LIFECYCLE_ID 94
- javax.faces.STATE_SAVING_
 METHOD 94
- JDeveloper (Oracle)
 child component
 manipulation with 225
 debugging with 894
 design-time benefits of 876
 JSF and 7, 146
 overview 874
 ProjectTrack Login page,
 creating with 879
 workspace 878
- JDK 1.5 (Tiger) 798
- Jelly (Apache Software
 Foundation) 672
- JSF Central community site 10,
 959
- JSF classes, categories of 411
- JSF Console (Holmes) 325
- JSF implementations 933
- JSF RI (reference
 implementations), Sun 10
- as standard for all
 implementations 89

JSF RI (reference implementations), Sun (*continued*)
 description 933
 effecting configuration changes 101
 ensuring same behavior 181
 internationalization and localization support with 236
 JSF library non-support by 92
 language support for 269
 quirks of 171
 RenderKit support by 448
 UI extension defining 606
JSF. *See* JavaServer Faces (JSF)
 jsf-api.jar file 91
 jsf-impl.jar file 91–92
JSP (JavaServer Pages)
 component integration with 822
 controlling access to 360
 converter integration with 658, 866
 custom tag implementation 607
 custom tags 718
 defined 15
 placement in root directory 289
 renderer integration with 647
 technical restrictions and “class” 143
 Tiles pages, converting to JSF 593
 UI component integration with 627
 UI extension integration with 607
 UIHeadlineViewer component integration with 781
 UIInputDate component integration with 718
 validator integration with 652, 847
 version 1.2 and JSF 89, 607
 version 2.0 and JSF 607
 writing component tags for 627

JSP custom tags
 goal of Struts-Faces 577
 removing components used with conditional 107
 JSP includes 317
 JSP integration process 737
 JSP page directive 110
 jstl.jar file 92
 Jython scripting language 672

K

Keel meta-framework 672
 <key> element 120
 keys, storing as constants (ProjectTrack) 480

L

110n (localization) abbreviation 236
 language codes (ISO) 236, 990
 language of views (localization) 151
 languages, support for multiple.
See localization
 layers, application separation of 457
 types of 457
LDAP (Lightweight Directory Access Protocol) 546
Length validator 248, 250–251, 345
 libraries (JAR) 91
 libraries, resale/distribution 840
Lifecycle class 936
 <lifecycle> element 974
 links, action 221
 listboxes, creating 208
 listener classes, support for 624
 listener methods, support for 624
 listeners
 parameterizing 516
 states of 620
 <list-entries> element 968
 lists of data
 controls for displaying dynamic 324

lists of items
 configuring dynamically 465
 exposing objects from business tier in 466
 selecting items from 465
 local values 621
 <locale-config> element 964
 locales 151
 configuration of 237
 defined 235–236, 977
 determining user 238
 displaying numbers for 262
 importance of specifying 238
 keeping track of supported 414
 overriding JSF selected 238
 resource bundles and 238
localization 235
 application 402
 defined 235, 398, 552
 informational resources on 241
 JSF handling of 447
 110n abbreviation 236
logic
 business 457–458, 563, 599
 converter 847
 data store access 457
 encoding 613
 intermixing, problems with 458
 separating validation logic from application 840
logical outcomes 34, 129, 132–133
login failures 363
Login page (ProjectTrack)
 action listeners, adding with Java Studio Creator 926
 action listeners, adding with WebSphere Studio 904
AuthenticationBean class and 505
 components, binding to backing beans with JDeveloper 890
 components, binding to backing beans with WebSphere Studio 902

Login page (ProjectTrack)
(continued)
 components, binding to properties with Java Studio Creator 924
 configuration file editing with JDeveloper 890
 creating page with Java Studio Creator 920
 creating page with JDeveloper 879
 creating page with WebSphere Studio 898
 input field requirements 305
 integration with application 361
 layout 292
 layout improvement 308
 navigation rule for 364
 navigation rule update 364
 navigation, adding with Java Studio Creator 928
 navigation, adding with JDeveloper 892
 navigation, adding with WebSphere Studio 912
 outcome possibilities 364
 overview 291
 testing with Java Studio Creator 932
 testing with JDeveloper 894
 testing with WebSphere Studio 913
 validation, adding with Java Studio Creator 923
 validation, adding with JDeveloper 888
 validation, adding with WebSphere Studio 901
 logins, encrypted 546
 LongRange validator 248, 250–251

M

Managed Bean Creation facility 47
 automatic bean creation with 460

automatic bean exposure with 472
 benefits of 111
 combining backing beans with 494
 description 98
 initializing backing bean properties with 565
 object creation with 482
 purpose of 110
 managed beans
 at application startup 112
 configuration with XML elements 111
 configuring backing beans as 472
 converting types used for map keys 122
 defined 35, 47, 110
 Dependency Injection pattern and 112
 expressions used with 107
 implicit variables associating with 128
 Inversion of Control (IoC) pattern and 112
 JSTL and referencing 111
 referenced beans 114
 restricted scoped variables, associating with 128
 Setter Injection pattern and 112
 web application scopes 113
 <managed-bean> element 114, 965
 <managed-bean-class> element 113
 <managed-bean-name> element 113
 <managed-bean-scope> element 113
 <managed-property> element 114, 966
 <map-entries> element 120, 967
 <map-entry> element 120, 967
 mapping prefix 93
 mapping suffix 93
 McClanahan, Craig 19
 memo fields, creating 194
 message bundles 269
 message format elements 161–163
 message format patterns 161
 message format styles 163
 message format types 163
 MessageFormat class 162
 messages
 adding to a user interface 304
 application 169–170, 328
 conversions and 252
 customizing validation 307
 description 40
 for application errors 55
 informational 56
 internationalizing 557
 localizing for Russian 561
 login failures 364
 methods for logging 428
 overriding default 414
 overriding standard 270
 properties of 557
 reporting to UI 422
 severity levels of 169, 269, 423
 sources of 55
 standard (JSF RI) 307
 standard (JSF) 270
 user input errors 55
 validation and conversion 27, 236, 414
 validation error 30
 MethodBinding class 413, 417–418
 method-binding expressions 79
 encapsulation of 415
 method-binding expressions 85
 methods
 action listener. *See* action listener methods
 component decoding 612
 component encoding 612
 convenience wrappers around 424
 event handling 615
 validator. *See* validator methods

methods (*continued*)
 value-change. *See* value-change methods

Microsoft
 ActiveX 21
 ASP.NET 5
 ASP.NET Web Forms 5, 95, 500
 .NET Framework 5
 Visual Basic 4, 8, 37, 176
 Visual Studio.NET 5, 918

model objects
 as managed beans 47
 binding to a component's value 47
 defined 47
 HtmlNavigator custom component and 323
 properties exposure 357, 359
 relationship with Update Model Values phase 66
 SelectItem class 453
 working with UIComponents compared to 828

Model-View-Controller (MVC)
 design pattern
 defined 17
 enforcing architecture of 41
 Model 2 variation 18

Mozilla open source browser 670

mutator methods (setters) 14, 463

MVC. *See* Model-View-Controller (MVC) design pattern

MyFaces open-source JSF implementation 90, 95, 933

myfaces_allow_designmode 96

myfaces_allow_javascript 96

myfaces_pretty_html 96

myfaces.jar files 92

N

naming containers 72–73, 623

NamingContainer interface 439, 623

Nash, Michael 672

navigation
 adding with Java Studio Creator 928
 adding with JDeveloper 892
 adding with WebSphere Studio 912
 control 188
 defined 56
 description 40
 global rules 134
 headers for 317
 logical outcomes 41, 51, 129, 132–133
 redirecting to the next view 132
 separate configuration files 134
 similarity to Struts 57
 visual configuration in an IDE 130

navigation cases
 defined 36, 56
 selection of 129
 storage of 57

navigation handlers 56, 129

navigation rules
 defined 36, 56, 98, 129
 global 134
 hardcoding 288
 union of rules for all pages 329

<navigation-case> element 130, 133

NavigationHandler class 938

<navigation-rule> element 130, 969

Navigator_ToolbarTag custom tag handler class
 configuration via JSP tags 826
 properties 822
 referencing value-binding expressions for 826
 tag handler for 823
 tag library descriptor for 831

NavigatorActionListener custom component helper class 809

NavigatorItem custom component model class 796

NavigatorItemList custom component model class 798, 834

NavigatorItemTag custom tag handler class
 NavigatorItemList initialization with 834
 tag handler 827
 tag library descriptor for 833
 nested tags 104–105

.NET Framework (Microsoft) 5, 223

Netscape browser 670

New Atlanta
 ServletExec 89

news sites 757

none scope 113

Non-Faces requests 571

Non-Faces responses 571

<null-value> element 114, 117, 120

Number converter 110
 description 256
 displaying proper currency signs 265
 properties 262
 using the 253, 262
 using with decimal format patterns 266

O

object-based development 356

object-oriented development 356

objects
 accessed through value-binding expressions 462
 associating with each other 565
 business layer 458
 conversion into strings for display 252
 custom converters for 461
 exposing properties 462
 hooking to components 357

objects (*continued*)
 JavaBean property exposure
 and JSF-integrated 460
 JSF application 413
 property publishing and JSF
 interaction 460
 reducing dependencies
 between action methods and
 sets of 501
 requirements for sharing or
 persisting to disk 462
 retrieval by value-binding
 expressions 566
 retrieval from application
 scopes 478, 566
 retrieval from data stores 476
 storage 428, 478
 OpenSymphony
 SiteMesh 317
 WebWork 17–18
 Operation model object
 (ProjectTrack) 475
 Operations 475
 Oracle
 access with integration layers 458
 Application Development Framework UIX 7, 19, 874–875
 Application Server 8, 89
 JDeveloper. *See* JDeveloper (Oracle)
 view on JSF 875

P

page code 909
 panel components 176
 parent forms 820
 pass-through properties 142, 301
 pass-through rule exception 143
 password input fields
 creating 195
 displaying 292
 PDA (Personal Digital Assistant) 5

Perl language 841
 phase events 55, 428, 435
 phase listener registration 99, 974
 PhaseId 430
 PhaseListeners 436
 phone numbers, expressions for 841
 plain old java objects (POJOs) 18, 458
 pluggable JSF classes
 configuring 942
 decorating 943
 extending 940
 overview 938
 replacing 947
 POJOs (plain old java objects) 18, 458
 portal 13
 Portlet API 410, 422, 424
 portlets 14
 postback 61, 192, 300, 837
 PowerBuilder (Sybase) 4
 prefix mapping 93
 Previous buttons 448
 Process Validations phase 69, 622, 642
 Project Details page
 (ProjectTrack)
 bean used for 390
 components used by 390
 integration with application 390
 navigation rule for 352, 395
 navigation rule update 395
 overview 347
 ShowHistoryBean and 390, 534
 Project Manager, role of 317
 Project model object
 (ProjectTrack) 374, 379
 ProjectTrack case study
 adding a form to 295
 adding rollover effects to 301
 adding validator error
 messages to 304
 application layer classes 476
 application logic of 434

Approve a Project page. *See*
 Approve a Project page
 (ProjectTrack)
 backing bean construction 505
 backing beans for 357
 business (domain) model 283
 business layer 473
 business object retrieval 459
 conceptual model 281
 configuration file 290
 controlling access to pages 547
 converters 491
 custom authentication of 505
 default page for 290
 deployment descriptors 397
 directory structure 360
 enhancing pages 300
 environment segmentation 357
 error page 396
 formatting text with Cascading Style Sheets 303
 header page. *See* header page
 (ProjectTrack)
 Inbox page. *See* Inbox page
 (ProjectTrack)
 includes used by 379
 initial directory structure 289
 input form integration 379
 integration layer 476
 integration process 357
 internationalizing 398, 551
 listing projects 511
 localizing for Russian 402, 556, 561
 Login page. *See* Login page
 (ProjectTrack)
 main menu 370
 messages of 557
 model object properties
 exposure 357, 359
 model objects of 366, 374, 379, 390
 multiple layers of 459
 navigation headers 317
 object model 475

ProjectTrack case study
(continued)
object-based development approach 356
operation pages 379
organizing code into backing beans 500
page requirements 317
paging through history 534
parameterizing listeners for 516
password field for 291
project approval and rejection 511
Project Details page. *See* Project Details page (ProjectTrack)
project lists, views of 324
Project Manager, role of 317
project updating 522
projects, creating new 528
Reject a Project page. *See* Reject a Project page (ProjectTrack)
required functions 290
security for 360, 545
selecting language for 320
Show All page. *See* Show All page (ProjectTrack)
storage of constants 478
synchronization and 501
system entities 281
system requirements 278
toolbar used in 795
toString method of business objects 462
user interface 283
User object converter for 854
views, list of 359
Visit class 492
properties
 attributes compared to 610
 read-only 464
<property> element 959, 961
<property-class> element 115
<property-name> element 114
PropertyResolver class 938
prototypes (UI) 317

R

RAD. *See* Rapid Application Development (RAD)
radio buttons, creating 212
Rapid Application Development (RAD)
 building emphasis 145
 developers 288
 tools 4
RDF Site Summary. *See* RSS (Rich Site Summary)
read-only access 545
read-only properties 464
read-write access 545
<redirect> element 132
reference implementations, JSF. *See* JSF RI (reference implementations), Sun
referenced beans 98, 114
<referenced-bean> element 968
referrer 60
RegexpValidatorTag custom tag
 handler class
 regular expression validators and 842
 tag library entry for 851
 writing the 848
regular expressions
 characters for 841
 defined 841
 expressions, regular 841
 JSP tag handler for 842
 validator classes needed to build 842
RegularExpression validator 345
RegularExpression validator
 custom validator class
 core converter logic 847
 JSP integration of 847
 properties 843
 registration of 842, 847
 registration with UIInput component 852
 StateHolder interface and 842
 tag handler for 848
 using 852
writing the 842
Reject a Project page (ProjectTrack)
 access authorization 546
 beans used for 381
 integration with application 385
 navigation rule for 341, 385
 navigation rule update 385
 overview 338
 purpose of 379
release method 653, 660
render kits
 adding renderers to 971
 default 414
 defined 99
 methods for handling 447
 renderer types for standard 636
 renderers and 44, 148
Render Response phase 69, 421
Renderer class 641, 728
<renderer> element 972
renderer-neutral behavior 42
renderers
 adding to render kits 971
 application configuration files for 970
 attributes of 730
 attributes used by 443
 classes and elements required for custom 639
 components with rendering functionality compared to 729
 components, indirect associations with 636
 configuring 644
 decorating existing 750
 defaults for components 778
 defined 43
 description 39
 encoding and decoding components with 728
 IDEs and importing custom 647
 JSP integration 647
 overview 636

- renderers (*continued*)

 purpose of 728

 registration 644

 relationship with converters 48

 render kits 44, 148

 RenderKit and 643, 936

 replacing default 639

 retrieving instances of current 615

 simple components and 615

 single 972

 types (list of) for the standard render kits 637

UI component families and corresponding 637

UI components compared to 615

when to write new 640

wrapping existing 750

writing new 639
- rendering

 defined 41

 delegated implementation model 43

 direct implementation model 43

rendering models 636, 707

RenderKit class 643, 936

<render-kit> element 971

Request Processing Lifecycle

 Apply Request Values phase 59, 64, 428, 622

 defined 57

 execution with Lifecycle class 936

 goal of 571

 Invoke Application phase 59, 66–68

 methods for component-based processing 616

 phase events 428

 possible event listener effects upon 59

 Process Validations phase 59, 65, 622, 642

 Render Response phase 59, 68–69

Render View stage 571

request and response faces of 571

Restore View phase 59

Update Model Values phase 66

request scope 501, 538

resource bundle keys 480

resource bundles

 components used with 241

 configuring application-wide 414

 creating in Java 238

 creating programmatically 241

 defined 238

 internationalization with 398, 400

 internationalizing text with 552

 localization with 402

 location of 241

 usage 236

 utility methods used with 414

resource paths, accessing 424

ResponseWriter class 613

Restore View phase 69

restoreAttachedState method 620

restoreState method 618, 717, 808

result sets, JDBC 541

RI, JSF. *See* JSF RI (reference implementations), Sun

rollover buttons 301, 728

rollover function, JavaScript 301

RolloverButton renderer

 classes and configuration files for 728

 using the 748

RolloverButtonDecorator-Renderer custom renderer class 750

RolloverButtonRenderer

 custom renderer class

 attributes for 729

 decoding 735

encoding 731

JSP integration and 737

registration 736

tag handler for 740

RSS (Really Simple Syndication) 757

RSS feeds

 challenges of 760

 channels 761

 Informa API and 759

 multiple versions of 760

 third-party libraries for 760

 UIHeadlineViewer and

 See UIHeadlineViewer component

runtime 21
-
- S**
- <s:form> tag 593, 599

saveAttachedState method 619

saveState method 618, 717

Scalable Vector Graphics (SVG) 43–44, 148

scoped variables

 changing names of 494

 implicit variables for 81

 looking up beans stored as 494

 overview 80

 using with JSP and JSTL 81

scripting languages 672

scrolling, control of 395

SDO (Service Data Objects) 895

Secure Sockets Layer (SSL) 546

security

 authentication 545

 authorization 545

 custom 547

 organizing pages for 360

 web container-based 546

SelectItem class 453, 465, 468

SelectItemGroup class 454

selectItems scoped variable

 (ProjectTrack)

 associated views 359

 description 358

 summary 380

- SelectMany family components
 200, 248, 380
 populating lookup lists for
 487
 value of instances 468
 working with 465
- SelectOne family components
 200, 212, 380
 populating lookup lists for
 487
 working with 465
- SelectProjectBean backing bean
 class (ProjectTrack)
 code analysis 511
 listing project information
 with 541
 managed bean configuration
 519
- selectProjectBean scoped
 variable (ProjectTrack)
 associated views 359
 description 358
- Service Data Objects (SDO) 895
- Servlet API 410, 422, 424, 547,
 572
- servlet classes 410
- servlet container. *See* web
 containers
- servlet filter 132
 for authorization 547
- servlet models 410
- ServletContext interface 410
- ServletContextListener
 interface 112
- ServletExec (New Atlanta) 89
- servlets
 defined 13
 entry points for primary 410
 lifecycles 289
 relationship with JSF 13
- session scopes 501
- sessions
 defined 13
 servlet 491
 states, visit objects for 491
- Setter Injection pattern 112
- setter methods (mutators) 14,
 463
- shared classes 92
- Show All page (ProjectTrack)
 324, 330, 378, 541
- showAllBean scoped variable
 (ProjectTrack) 370
- showHistoryBean scoped
 variable (ProjectTrack)
 associated views 359
 code analysis 534
 description 358
 managed bean configuration
 538
 reusing 538
 summary 390
- Simplica ECruiser 933
- singletons 476, 487
- SiteMesh (OpenSymphony) 317
- skin (alternate look and feel) 43
- Smile 667, 933
- social security numbers,
 expressions for 841
- SOFIA (Salmon) 19
- special characters 266
- SQL queries 542
- SQL Server 458
- SSL (Secure Sockets Layer) 546
- Standard Widget Toolkit
 (SWT) 7
- standard.jar file 92
- StateHolder interface 439, 617,
 717, 842, 856
- StateHolder methods 807
- stateless protocols 11
- StateManager class 938
- static content 11
- static includes 103
- static methods 480
- static text 288
- strings
 accessing from bundles 241
 conversion of 461
 converting objects into 252
 display 552
 locale specific text 238
 message 552
 storing as constants 478
 value-binding expressions
 and localized 244
- Struts (Apache Software
 Foundation)
 ActionForwards 129, 134
 as foundation framework 18
 controller architecture 599
 HTML and Bean tag
 equivalents 578
 JSF directory structure
 compared to 90
 JSF integration with 16, 500,
 564, 572
 JSF navigation compared to
 129
 JSF rule setting compared to
 134
 overlap with JSF 19
- Struts Actions, invoking from
 JSF event handlers 598
- struts implicit variable 581
- Struts in Action (Husted) 573
- struts-config.xml 90
- Struts-Faces example application
 adding proper libraries 575
 converting complicated pages
 585
 converting JSP Tiles pages
 593
 converting simple pages 582
 converting Struts JSPs to use
 JSF components 599
 importing the tag library 577
 invoking Struts Actions from
 JSF event handlers 598
 migrating Struts JSP pages
 577
 scenarios 573
 using JSF action methods 597
 using JSF managed beans 597
- version 5.0 575
- Struts-Faces integration library
 decorating functionality of
 943
 development of 572
 extending JSF with 944
 goal of 573
 tags of 577
 versions of 582
- submitted values 63, 621

suffix mapping 93–94
Sun
 Java Studio Creator. *See* Java Studio Creator (Sun)
 Java System Application Server 12
JSF RI. *See* JSF RI (reference implementation), Sun
 superclasses and HTML components 441
SVG (Scalable Vector Graphics) 43–44, 148
Swing
 defined 21
 event listener interface requirement 34
 JavaBean enabling of 14
 JSF compared to 7, 49
SWT (Standard Widget Toolkit) 7
 symbols, date format pattern 260
 synchronization, need for 501

T

tag classes 629
 tag handler classes 628
 tag handlers 628, 630, 718, 741, 822
 tag libraries 737, 745
 tag library definitions (TLDs) 627, 842
Tapestry (Apache Software Foundation) 19
 template languages 672
 template text, using 150
 terminology (JSF) 39, 666
 text
 externalizing into resource bundles 398
 internationalizing from back-end code 244
 internationalizing with resource bundles 552
 JSF applications and static 288
 text fields, creating 193
 thread safety 501
 threading conflicts 501

Tiles (Apache Software Foundation) 317, 594
 Tiles pages, JSP, integrating with JSF 593
 time zone codes 977
 time zone identifiers 977
 TLDs (tag library definitions) 627, 842
Tomcat (Apache Software Foundation) 8, 12, 89, 92
ToolbarRenderer class
 UINavigator and 811, 822
ToolbarRenderer custom
 renderer class
 attributes for 811
 component tag for 822
 decoding 820
 encoding 811
 registration of 821
 sample display of 795
 UINavigator and 795, 822
 toolbars, componentization of 795
toString method 461
<to-view-id> element 132
 two-way editors 145
 type conversion
 defined 235
 handling by converters 654
 JSF support for 461
 support for 450
 web framework feature 654

U

UI component development
 adding event listeners 615
 application code
 development compared to 607
 attribute and property retrieval 610
 component configuration 624
 component tree management 617
 declaring output in a template 613
 decoding methods for 612

defining families and types 611
 elements of 609
 encoding methods for 612
 event handling methods 615
 event handling with method-bindings 623
 handling component values 621
 interpretation of client input 613
JSP integration 627
 overview 607
 registration 624
 renderers, setting default 615
 rendering the client identifier 613
 retrieval of current renderer 615
 skill set for 607
 state saving 617
 state saving helper methods 619
 subclassing base classes for 627
 UI development compared to 607
UIComponentBodyTag and 633
UIComponentTag and 627
 value-binding enabling properties 616
 when to write UI components 608
 writing properties 616
UI components
 accepted types for value Property 464
 accessing properties of 443
 action source 186
 application configuration files for 970
 associating Date objects with 256
 associating validators with 251
 association of converters with 252
 association with single converters 654
 base class for 440

UI components (*continued*)
behavior of standard 288
behaviors and property settings 145
binding a value to a backing bean 46
binding to backing beans 28, 46
binding to backing beans with JDeveloper 890
binding to backing beans with WebSphere Studio 902
binding to properties with Java Studio Creator 924
binding value properties to backing beans 469
classes 411–412, 439, 441
client identifiers. *See* client identifiers
component identifiers. *See* component identifiers
composite 759
conditionally displayed with JSTL tags 107
created by standard JSP component tags 438
creating and initializing in property accessor 470
data-aware 356
defined 21, 41
delegated rendering implementation of standard 636
development. *See* UI component development
development with ActionListener 433
disabled 614
dynamic building of graphs 471
dynamically creating and manipulating in views 469
encapsulating JavaScript functionality within 728
encoding and decoding methods for 636
encoding and decoding with renderers 728

families and renderers 612
families of 139
families, exposure of 611
finding on a page 445
handling JavaScript with JSF 302
hooking objects to 357
hooking up directly to data service objects 541
HTML subclasses 439
identification by client identifiers 421
IDEs and importing custom 627
importance of 142
integrating Struts applications with JSF 573
integrating with correct properties 379
integration with CSS 138, 143
integration with development tools 145
integration with JavaScript 138
interfaces 438, 441
javax.faces prefix and standard 612
list of standard 139
localized strings and 244
manipulating in a view 438
manipulating in code 34, 76
manipulation in a view 460
manipulation without JSP 667
model-driven 795
postbacks to 837
properties specific to HTML 141
property associations with value-binding expressions 142
purpose of 607
referencing on the client 75
registration 99, 970
relationship between classes and interfaces 441
renderer types and corresponding families of 637
renderer-independent attributes of 610
renderer-independent properties of 610
renderer-neutral behavior 42
renderers (default) for 778
renderers compared to 615
renderers compared to rendering functional 729
renderers, indirect associations with 636
replacing default 611
resemblance to standard HTML controls 138
resource bundles used with 241
setting parameters 151
state 42
state retrieval/change 438
states of, storage and retrieval 617
subclasses of 612
subclassing naming containers 623
support for 138
tags associated with 142
tree representation 149
trees of 617
type 443, 611
updating of local values 622
using value-binding expressions for 368
using with expressions 86
using with JavaScript 75
validators and custom 246
validators and third-party 246
value memory between requests 717
value validity 452
visibility indication 444
with visual representation 138
UI extensions (JSF)
adding configuration entries to configuration files 606
classes, subclassing the 606
configuration 959
converters 654
corresponding configuration entries 606

- UI extensions (JSF) (*continued*)**
- defined 606
 - development 96
 - directory structure 661
 - implementing interfaces 606
 - integrating classes with
 - display technology 607
 - integrating with JSP 636
 - internationalization 660
 - JAR files for 660
 - key steps to implementation 606
 - packaging 660
 - reference implementation
 - defining for default 606
 - renderers 636
 - UI components 607
 - validators 648
- UI frameworks 19, 224, 570
- UI layers 457
- UIColumn component** 139, 439
- UICommand component**
- adding rollover functionality to 301, 728
 - image rollover support with 728
- UIComponent class**
- description 439
 - instance container 412
 - JavaScript output with 840
 - overview 442
 - subclassed UIComponentBase compared to subclassing 611
 - UI component attribute and property retrieval through 610
 - ValueHolder interface
 - implementation with 449
 - working with model objects compared to 828
- UIComponentBase class** 439–440, 610, 636
- UIComponentBodyTag** 627, 633
- UIComponentTag** 627
- UIData component**
- DataModel object and 541
 - listing projects with 511
 - paging through data with 534
- UIHeadlineViewer custom component** usage compared to 758, 789
- using Informa API with 762
- UIHeadlineViewer custom component**
- adding styles to 790
 - configuration/registration of 780
 - displaying RSS feeds with 757
 - encapsulation of declared components in 764
 - goal of 768
 - HeadlineViewerTableTag and 781
 - implementation elements 759
 - JSP integration of 781
 - properties of 769
 - subclassed UIData component for 759
 - UIData component usage compared to 758, 789
 - UIHeadlineViewer class for 768
 - usage 789
 - using Informa API with 762
- UIHeadlineViewer custom component class** 768
- UIInput class** 708
- UIInput component** 852
- UIInputDate custom component**
- classes and configuration files for 707
 - configuration 718
 - decoding 715
 - encoding 709
 - InputDateTag 718
 - invoking 724
 - JSP custom tag for 718
 - JSP custom tag library 722
 - JSP integration 718
 - overview 706
 - registration 718
 - state management 717
 - tag handler for 718
 - UIInputDate class for 708
- UIInputDate custom component class** 708
- UINavigator custom component**
- benefits of 323
 - default CSS style for 819
 - elements for building 795
 - encoding methods for 813
 - for headers 321
 - JSP component tag for 795
 - layout of 817
 - model classes for 796
 - overview 795
 - parent UIForm for 820
 - purpose of 834
 - registration of 810
 - ToolbarRenderer class for 811
 - usage 834
- UINavigator custom component class**
- custom ActionListener for 809
 - methods for 803
 - overriding
 - UIComponentBase methods 806
 - state management for 807
 - writing the 801
- UINavigator-ToolbarRenderer pair**
- component tag for 822
 - sample display of 795
- UIOutput class** 440
- UIOutput component**
- description 140
 - embedding custom tags and markups 158
 - escaping large blocks of body text 157
 - summary 156
 - using with verbatim tag 155
- UIParameter class** 440
- UIParameter component**
- configuration of 517
 - description 140
 - setting component parameters with 151
 - summary 151
- using inside IDEs 152
- using with HtmlOutputLink 167

UISelectItem component
description 140
item list configuration with 453
list specification with 465
requirement of special type
for value-binding properties 485
single items, displaying with 200
summary 201

UISelectItems component
description 140
item list configuration with 453
list specification with 465
required association 453
requirement of special type
for value-binding properties 485
summary 204

UIViewRoot class 412, 446

UIViewRoot component
changing locale for 151
description 141
page control with 110, 149
summary 149

UIX (Oracle) 7, 19, 874–875

Unix command line 841

Update Model Values phase 69

updateProjectBean scoped variable (ProjectTrack)
associated views 359
code analysis 522
description 358
managed bean configuration 526
summary 381

URLs
encoding with ExternalContext 427
rewriting 12–13, 165, 168

user commands 432

user input
storage of 621
translating into custom types 488

user interface (UI)
adding validators to 304

binding a value to a backing bean 46
binding to a backing bean 46
building without Java code 288, 317
component identifier 42
components. *See UI components*
creating 145
declaring with display technology 416
declaring with templates 438
deployment descriptors and development of 290
developing separately, benefits of 288
event-firing by 623
extensions 970
family 141
first steps in creating a 288
helpers 840
integration of 356
interaction with 428
internationalizing 398
object display 252
prototypes in JSF 317
referencing on the client 75
renderer-neutral behavior 42
state 42
UI component behaviors and 145
UI component development compared to development of 607
using panels for layout 308
using with expressions 86
using with JavaScript 75
without application logic 288

user interface extension
registration 99

User model object (ProjectTrack) 366

user requests
access to current data 411
processing 410

UserConverter custom converter class browser view 854

JSP integration of 866
registration of 865
tag handler for 866
using 870
writing the 856

UserConverterTag class 866

users
authenticated 360
changing status of Projects 475
credential validation 545
disabling toolbar items for particular 369
resource access authorization 545
roles of 475

V

validation
behavior handling by input components 249
customizing error messages 307
defined 235, 648
importance of 245
in backing beans 648
integration with Struts 585
JSF support for 304
message displays 245
validation logic 840
Validator interface 648
validator methods 45, 85
backing beans and 245
compared to validator classes 840
support in EditableValueHolder interface 451
usage 245

<validator> element 973

validators
accessing 246
adding with Java Studio Creator 923
adding with JDeveloper 888
adding with WebSphere Studio 901

- validators (*continued*)
- application configuration files
 - for 970
 - associating components with 251
 - attributes used by 443
 - combining different 251
 - component validation 45
 - custom 247, 251
 - data entry errors and 501
 - description 39
 - developing 840
 - evaluating value-binding expressions for 850
 - external 45
 - input control acceptance of 248
 - JavaScript emitting 840
 - JSP integration of 652, 840
 - Length 345
 - order of execution 251
 - output and 840
 - overview 648
 - purpose of 28
 - registering 650
 - registering multiple for controls 251
 - registration 99
 - RegularExpression 345
 - SelectMany family
 - components and standard 248
 - setting properties of standard 248
 - stand-alone 840
 - standard JSF 245, 247, 840
 - states of 620
 - Struts 585, 840
 - third-party 251
 - usage 245
 - using 246
 - using with form input fields 345
 - validator methods. *See* validator methods
 - writing 648, 840
- ValidatorTag class 652
- value property 464, 622
- <value> element 114, 117, 120
- ValueBinding class
 - overview 417
 - storing objects in application scopes with 428
 - summary 413
- value-binding enabled
 - properties/attributes 444
- value-binding expressions
 - association of 444
 - components that accept 244
 - encapsulation of 415
 - evaluating for validators 850
 - object retrieval by 566
 - objects accessed through 462
 - property norms for 779
 - retrieval of 444
 - support of 444
 - value property support for 720
- value-change events 51
 - defined 50
 - firing after the Process Valiations phase 65
 - firing by EditableValueHolder interface 623
 - generation and firing of 186–187
 - handling 434
 - representation of component value changes 428
- value-change listener
 - method. *See* value-change listener methods
- value-change listener classes
 - combining with single value-change listener methods 187
 - declaring 187
 - registering multiple 187
- value-change listener methods
 - 50, 434
 - combining with value-change listener classes 187
 - comparing with value-change listener classes registering 187
 - writing 434
- ValueChangeEvent class 429, 434
- ValueChangeListener interface 429, 434
- ValueHolder interface 438, 449
- VariableResolver class 938
- VBScript (Microsoft) 9
- Velocity (Apache Software Foundation) 15, 410, 672
- <verbatim> tag 155
- view state 95
- ViewHandler class 938
- views
 - component organization in 446
 - component trees for 617
 - composition of 411
 - defined 42, 149, 666
 - forwarding to other 550
 - identifier 61
 - implementation of structure of 444
 - implicit variable 83
 - interaction with business layer 459
 - representation by component tree 421
 - state saving options 61
- Visit backing bean class (ProjectTrack) 492
- visit scoped variable (ProjectTrack)
 - associated views 359
 - description 357
 - header page and 365–366
 - summary 365
- Visual Basic (Microsoft) 4, 8, 37, 176
- visual editors 959
- Visual Studio.NET (Microsoft) 5, 918

W

-
- WAR (web archive) 90
 - web application scopes 27, 113
 - web applications
 - directory structure of 289

web applications (*continued*)
specifying default pages for 290
start-up code 483
user information storage in servlet sessions 491
web archive (WAR) 90
web browsers, open source 670
web containers
Java System Application Server 12
Oracle Application Server 8, 89
ServletExec 89
Tomcat 8, 12, 89, 92
WebSphere Application Server 8, 89
web servers
services handled by 19
serving static content with HTTP 12
web sites, content/headline syndication of 757
WEB-INF directory 289
WEB-INF/classes directory 241

WEB-INF/faces-config.xml file 959
WEB-INF/lib directory 289
weblogs 757
WebMacro 15
WebObjects (Apple) 5
WebSphere Application Server (IBM) 8, 12, 89
WebSphere Studio (IBM)
adding parameters with 152
binding components to backing beans with 902
building JSF pages with 147
CSS style creation with 144
JSF and 895
mixing JSF and JSTL tags in 104
overview 896
ProjectTrack Login page, creating with 898
setting labels with 166
support for JSF applications 6
WebWork (OpenSymphony) 17–18
web.xml 289

Wireless Markup Language (WML) 43–44, 148

X

XML (eXtensible Markup Language) 15, 148, 670
XML configuration files 606, 959
XML dialects, custom 672
XML elements 98
XML processing frameworks 672
XML scripting engines 672
XML, XUL and 670
XSLT (eXtensible Style Sheet Language Transformations) 15
XUL display technology (Mozilla) 570, 670
XUL ViewHandler 948

JAVA SERVER FACES IN ACTION

Kito D. Mann • foreword by Ed Burns

ONLINE BONUS!

Exclusive to owners of this book: free online access to over 300 additional pages of substantial content. That's a total of over 1,000 pages of *JavaServer Faces in Action!*

JavaServer Faces helps streamline your web development through the use of UI components and events (instead of HTTP requests and responses). JSF components—buttons, text boxes, checkboxes, data grids, etc.—live between user requests, which eliminates the hassle of maintaining state. JSF also synchronizes user input with application objects, automating another tedious aspect of web development.

JavaServer Faces in Action is an introduction, a tutorial, and a handy reference. With the help of many examples, the book explains what JSF is, how it works, and how it relates to other frameworks and technologies like Struts, Servlets, Portlets, JSP, and JSTL. It provides detailed coverage of standard components, renderers, converters, and validators, and how to use them to create solid applications. This book will help you start building JSF solutions today.

What's Inside

How to

- Use JSF widgets
- Integrate with Struts and existing apps
- Benefit from JSF tools by Oracle, IBM, and Sun
- Build custom components and renderers
- Build converters and validators
- Put it all together in a JSF application

An independent enterprise architect and developer, **Kito D. Mann** runs the JSFCentral.com community site and is a member of the JSF 1.2 and JSP 2.1 Expert Groups. He lives in Stamford, Connecticut with his wife, two parrots, and four cats.

“Can't wait to make it available to the people I teach.”

—Sang Shin, Java Technology Evangelist,
Sun Microsystems Inc.

“This book unlocks the full power of JSF... It's a necessity.”

—Jonas Jacobi
Senior Product Manager, Oracle

“... explains advanced topics in detail. Well-written and a quick read.”

—Matthew Schmidt, Director,
Advanced Technology, Javalobby

“... by a programmer who knows what programmers need.”

—Alex Kolundzija, Columbia House

“A great reference and tutorial!”

—Mike Nash, JSF Expert Group Member,
Author, *Explorer's Guide to Java Open Source Tools*



Ask the Author



Ebook edition

www.manning.com/mann



8 791932 394122

ISBN 1-932394-12-5



\$49.95 US/\$74.95 Canada