

Foreword by Ed Burns

SAMPLE CHAPTER

# JavaServer Faces IN ACTION



Kito Mann

 MANNING



*JavaServer Faces in Action*  
by Kito Mann  
**Sample Chapter 4**

Copyright 2004 Manning Publications

# *brief contents*

---

## **PART 1 EXPLORING JAVASERVER FACES ..... 1**

- 1 ■ Introducing JavaServer Faces 3
- 2 ■ JSF fundamentals 38
- 3 ■ Warming up: getting around JSF 88
- 4 ■ Getting started with the standard components 137
- 5 ■ Using the input and data table components 185
- 6 ■ Internationalization, validators, and converters 234

## **PART 2 BUILDING USER INTERFACES ..... 275**

- 7 ■ Introducing ProjectTrack 277
- 8 ■ Developing a user interface without Java code:  
the Login page 287
- 9 ■ Developing a user interface without Java code:  
the other pages 316
- 10 ■ Integrating application functionality 354

**PART 3 DEVELOPING APPLICATION LOGIC ..... 407**

- 11 ■ The JSF environment 409
- 12 ■ Building an application: design issues and foundation classes 456
- 13 ■ Building an application: backing beans, security, and internationalization 499
- 14 ■ Integrating JSF with Struts and existing applications 568

**PART 4 WRITING CUSTOM COMPONENTS, RENDERERS, VALIDATORS, AND CONVERTERS ..... 603**

- 15 ■ The JSF environment: a component developer's perspective 605

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

- 16 ■ UIInputDate: a simple input component 705
- 17 ■ RolloverButton renderer: a renderer with JavaScript support 727
- 18 ■ UIHeadlineViewer: a composite, data-aware component 756
- 19 ■ UINavigator: a model-driven toolbar component 794
- 20 ■ Validator and converter examples 839

**ONLINE EXTENSION**

The five chapters in part 5 (plus four additional appendixes) are not included in the print edition. They are available for download in PDF format from the book's web page to owners of this book. For free access to the online extension please go to [www.manning.com/mann](http://www.manning.com/mann).

# 4

## *Getting started with the standard components*

---

### ***This chapter covers***

- UI component basics
- Component development with IDEs
- The standard output components

Now that you understand what JavaServer Faces is and you're familiar with the fundamental concepts, it's time to learn how to use the standard amenities. In this chapter and the next, we cover the standard components that are included with all JSF implementations. They provide the basic set of functionality necessary for building HTML-based web applications. (You can also expect web container vendors, tool vendors, and third-party developers to provide additional components as well.)

In this chapter, we start with an overview of all the components, discuss JSP integration in more detail, and examine components that aren't involved with user input. In the next chapter, we'll focus on the standard input components, and the `HtmlDataTable` component, which displays or edits tabular data from a data source. Our goal with these two chapters is to help you use the components to develop user interfaces (UIs). If you're a front-end developer, these chapters will be indispensable, because you'll be working with these components most of the time. If you're a back-end developer, understanding how these components work will help you develop your own components, and help to ensure smooth integration with your code.

## **4.1 *It's all in the components***

---

The most central feature of JSF is its support for UI components—this is what sets it apart from many other web development frameworks. JSF ships with a standard set of components that are intended to provide support for typical HTML UIs. These include widgets for text display, text entry fields, drop-down lists, forms, panels, and so on. These are listed in table 4.1.

Next to each component name you'll notice a display name column. These are user-friendly names you may see inside IDEs. As you can see, the names may vary by IDE. We haven't listed display names for components that aren't usually shown in a component palette, or for which no examples were available when writing this book. Regardless of what display name an IDE uses, the code it generates will be standard (in other words, any generated JSP or Java code will look like the code in this book).

You may have noticed that many of these components closely resemble standard HTML controls. This is no accident—the intent was to provide enough widgets to build basic HTML UIs. All of the standard components that have a visual representation generate standard HTML 4.01 and integrate well with CSS and JavaScript. (If you were hoping the standard components would support older browsers, you're out of luck—look for third-party alternatives.)

**Table 4.1 JSF includes several standard UI components for building HTML views. Related components with similar functionality are organized into families.**

Family <sup>a</sup>	Component	Possible IDE Display Names	Description
Column	UIColumn	N/A	A table column. Used to configure template columns for parent <code>HtmlDataTable</code> component.
Command	<code>HtmlCommandButton</code>	Command – Button, Button	A form button that is an action source and can execute an action method.
	<code>HtmlCommandLink</code>	Command – Link, Link Action	A hyperlink that is an action source and can execute an action method.
Data	<code>HtmlDataTable</code>	Data Grid, Data Table	A data-aware table with customizable headers, footers, and other properties. Can connect to multiple types of data sources.
Form	<code>HtmlForm</code>	N/A	An input form; must enclose all input components.
Graphic	<code>HtmlGraphicImage</code>	Image	Displays an image based on its URL.
Input	<code>HtmlInputHidden</code>	Hidden Field	An input field of type “hidden”.
	<code>HtmlInputSecret</code>	Secret Field	An input field of type “password”.
	<code>HtmlInputText</code>	Text Field	An input field of type “text”.
	<code>HtmlInputTextarea</code>	Multi Line Text Area	A text area (multi-line input field).
Message	<code>HtmlMessage</code>	Display Error, Inline Message	Displays messages for a specific component.
Messages	<code>HtmlMessages</code>	Message List	Displays all messages (component-related and/or application-related).

*continued on next page*

**Table 4.1 JSF includes several standard UI components for building HTML views. Related components with similar functionality are organized into families.** *(continued)*

Family <sup>a</sup>	Component	Possible IDE Display Names	Description
Output	HtmlOutputFormat	Formatted Output	Outputs parameterized text.
	HtmlOutputLabel	Component Label	A text label for an input field.
	HtmlOutputLink	Hyperlink	A hyperlink that's not associated with a user command.
	HtmlOutputText	Output Text	Plain text, with optional CSS formatting.
	UIOutput	N/A	Plain text (no formatting). Useful for enclosing HTML markup or other custom tags.
Parameter	UIParameter	N/A	An invisible component used to configure other components.
Panel	HtmlPanelGrid	Grid Panel	A table with customizable headers, footers, and other properties.
	HtmlPanelGroup	Panel – Group Box, Group Box	Groups components together for use inside of other components, and to apply common styles or hide/display a group of components.
Select-Boolean	HtmlSelectBooleanCheckbox	Check Box, Checkbox	A single checkbox.
Select-Item	UISelectItem	N/A	Represents a single item or item group for use in SelectMany and SelectOne components.
Select-Items	UISelectItems	N/A	Represents a collection of items or item groups for use in SelectMany and SelectOne components.

*continued on next page*

**Table 4.1 JSF includes several standard UI components for building HTML views. Related components with similar functionality are organized into families.** (continued)

Family <sup>a</sup>	Component	Possible IDE Display Names	Description
Select-Many	HtmlSelectManyCheckbox	Check Box Group	A table with a list of checkboxes, grouped together.
	HtmlSelectManyListbox	Multi Select Listbox	A listbox that allows you to select multiple items.
	HtmlSelectManyMenu	N/A	A multi-select listbox that shows one available option at a time.
SelectOne	HtmlSelectOneRadio	Radio Button Group	A table of radio buttons, grouped together.
	HtmlSelectOneListbox	Listbox	A listbox that allows you to select a single item.
	HtmlSelectOneMenu	Combo Box, Dropdown List	A drop-down listbox that allows you to select a single item.
ViewRoot	UIViewRoot	N/A	Represents entire view; contains all components on the page.

<sup>a</sup> Technically, each of the component family names have the prefix “*javax.faces.*” So, the “Form” family is really called *javax.faces.Form*. Using the full component family name is useful when you’re working with renderers or components in Java code.

Table 4.1 also reveals another aspect of UI components: they’re organized into families. A *family* is a group of related UI components that have similar behavior. Families are primarily used behind the scenes to assist with rendering. However, because components in the same family have similar functionality, it’s useful to talk about components in the same family together, and that’s the approach we’ll use in this chapter and the next.

Most of these components have properties specific to HTML—for example, the `HtmlPanelGrid` component has a `cellpadding` property, because it maps to an HTML table. You can manipulate these properties if you’re working visually within an IDE, directly with JSP, or in Java code. However, these UI components are subclasses of more generic components that don’t have specific properties for the target client. If you’re writing Java code or developing custom components,

you may prefer to work with the superclasses so that you're not dependent on generating HTML; see parts 3 and 4 for more information.

All of this is fine and dandy, but what is it that makes UI components so important? We touched upon this in chapter 2—they provide stateful, packaged functionality for interacting with the user. For example, the `HtmlTextarea` component handles displaying an HTML `<textarea>` element to the user, updating its value with the user's response, and remembering that value between requests. As we've seen, that value can also be associated directly with a backing bean or model object. UI components also generate events that you can wire to server-side code. `HtmlTextarea` will generate a value-change event whenever the user enters a new value.

UI components also make great use of value-binding expressions—for the most part, *any* component property can be associated with a value-binding expression. This means you can specify all of your component's properties—everything from its `value` to other properties like its `size` and `title`—in some completely different data store. It doesn't really matter where it's stored (or if it's stored at all), as long as it's exposed through a bean.

**NOTE** In this chapter and the next, we cover the standard components from a front-end development perspective. Consequently, we will only list some properties that accessed exclusively with Java code, and we will not examine the component's methods. See part 3 for more details on writing Java code that interacts with UI components.

As we discussed in chapter 3, JSF is integrated with JSP through custom tags. Tags that are associated with UI components are called *component tags*. The ones that are specific to HTML are in the HTML tag library (usually with the prefix “h”), and the rest of them are in the core tag library (usually with the prefix “f”). The core tag library also has tags for validators, converters, and so on.

### 4.1.1 Using HTML attributes

All of the standard HTML components have properties that support basic HTML 4.01 attributes. These properties are passed directly through to the web browser, so they're called *pass-through* properties. They're available when you're manipulating a component in an event listener (as we demonstrated in chapter 1), and also in component tags.

In some cases, component tags map directly to an HTML element, so using pass-through properties seems quite logical. For example, this component tag:

```
<h:inputText value="hello" size="30" maxLength="40"  
             accesskey="T" tabindex="0" />
```

maps to this HTML:

```
<input type="text" name="_id1:_id2" value="hello" accesskey="T"  
       maxLength="40" size="30" tabindex="0" />
```

The properties marked in bold are passed through to the browser. Even though the `value` property looks as if it were passed through, it was actually checked for validity. This brings up an important point: pass-through properties are *not* checked for validity at all.

As you get a handle on how the component tags map to HTML elements, adding HTML pass-through attributes will become natural. They also make it easy to replace existing HTML tags with JSF component tags. We provide many examples of HTML pass-through attributes throughout the rest of parts 1 and 2.

### Using Cascading Style Sheets

One subtle exception to the pass-through rule is the `class` attribute, which is used to associate Cascading Style Sheets (CSS) class names with an HTML element. Due to technical restrictions in JSP, the name “class” can’t be used. As a workaround, most UI components have a property called `styleClass`. When you declare a component, you can specify multiple CSS classes by placing a space in between them:

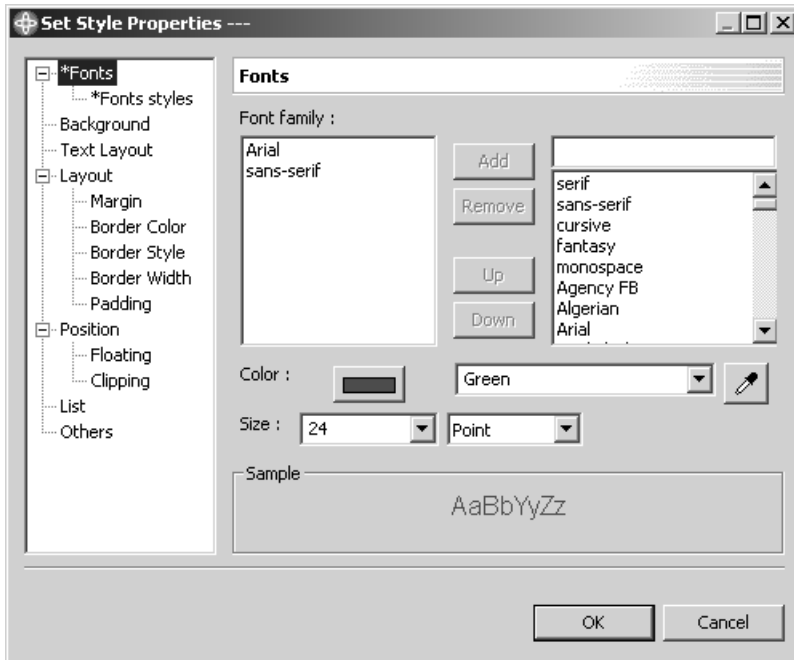
```
<h:myComponent styleClass="style1 style2 style3"/>
```

This specifies three CSS style classes for the fictional `myComponent`. Some IDEs will let you choose an existing class from style sheets that are currently in your project.

Most components also support the CSS `style` property, so you can specify their styles without a class. If you’re a CSS expert, you can integrate each component with your style sheets (or styles) manually by simply using the `style` and `styleClass` properties. Some IDEs simplify this process by including basic CSS editors that allow you to modify a component’s look and feel without knowing CSS, as shown in figure 4.1. Once you have selected the display properties (font color, background color, border, alignment, and so on), the IDE will create the proper CSS style for you.

#### 4.1.2 Understanding facets

So far, we’ve seen many examples of parent-child relationships, like an `HtmlInputText` component nested within an `HtmlForm`. JSF also supports named relationships, called *facets*. Facets aren’t exactly children, but they’re quite similar. They’re



**Figure 4.1** WebSphere Application Developer [IBM, WSAD] provides a convenient dialog box for creating new CSS styles that are used by some UI component properties.

used to specify subordinate elements like a header or footer, so they're common in more complicated components like an `HtmlPanelGrid` or an `HtmlDataTable`. Here's an example of the latter:

```
<h:dataTable value="#{defaultUser.favoriteSites}" var="site">
  <f:facet name="header">
    <h:outputText value="Table header"/>
  </f:facet>

  <h:column>
    <f:facet name="header">
      <h:outputText value="Column header"/>
    </f:facet>
    <h:outputText value="#{site}"/>
  </h:column>

  <f:facet name="footer">
    <h:panelGroup>
      <h:commandButton value="Next page" action="#{myBean.nextPage}"/>
      <h:commandButton value="Previous page"
        action="#{myBean.previousPage}"/>
    </h:panelGroup>
  </f:facet>
</h:dataTable>
```

```
</h:panelGroup>  
</f:facet>  
</h:dataTable>
```

This example has three facets: one for the table's header, one for the column header, and one for the table footer. As you can see, the `<f:facet>` tag defines the relationship, but the facet's child is the actual component that's displayed. Each facet can have only one child component, so if you want to include several components, they must be children of another component, like the `HtmlPanelGroup` used to enclose the components in the footer facet.

You'll see more facet examples when we explore components, like `HtmlPanelGrid`, that use them. In the meantime, let's take a look at how UI components integrate with development tools—one of JSF's primary goals.

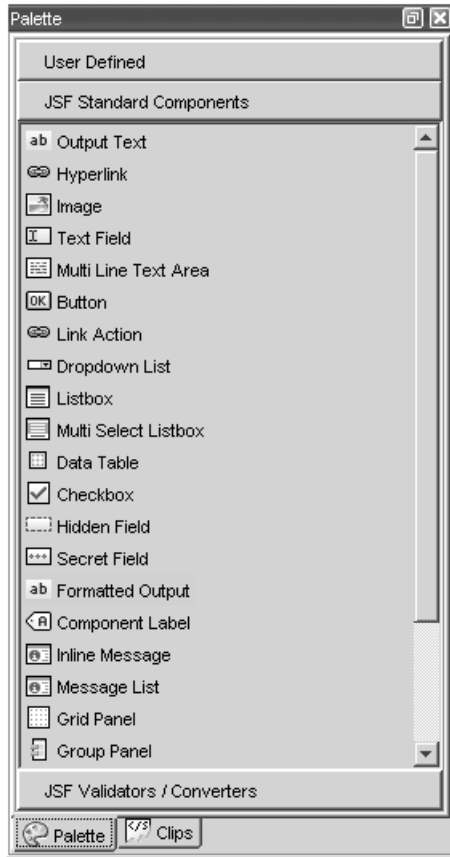
### 4.1.3 The power of tools

In chapter 1, we talked about the days of Rapid Application Development (RAD) and how the emphasis was on building UIs by dragging and dropping UI components from a palette. That type of functionality is one of the primary goals of JSF. When you use an IDE that has full-fledged support for JSF, there is usually a component palette that includes all of the standard JSF components, and often some nonstandard ones as well. Component palettes are usually small windows that list all of the components with an icon beside them, as shown in figure 4.2. Most tools will also offer proprietary components in addition to standard JSF components, as shown in figure 4.3.

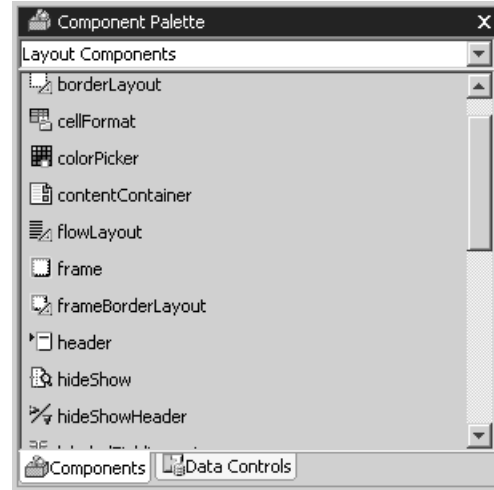
You build the UI by creating or opening a page, dropping the components from the palette into it, and then customizing the component's properties. Usually you'll modify things like the `value` (which is often associated with a backing bean property), the `rendered` property (which controls whether or not the component is visible), and HTML-specific properties like the CSS style. Figure 4.4 should give you an idea what this process looks like.

Because building a UI is largely dependent on the behavior of the UI components, we'll spend some time describing how each of these components behave with different property settings. Moreover, you'll see the raw JSP that these tools often generate (it's quite possible to visually design JSF views without JSP, but tools will initially support only this style of development with JSP).

Knowing how to use the component tags can be useful for cases where you're not using a tool at all, or you prefer to tweak the JSP by hand and immediately see the results while you're working inside an IDE. Most IDEs have *two-way editors*, so any changes you make in the designer will be propagated to the source, and vice versa.



**Figure 4.2** Java Studio Creator [Sun, Creator] has a dockable component palette. Note that the display names are different than the actual names of the components, as shown in table 4.1. The display names may vary between IDEs.



**Figure 4.3** Oracle provides a version of JDeveloper [Oracle, JDeveloper] that integrates its extensive palette of UIX components with JSF.

IDEs usually allow you to work with Java code as well—in fact, tools like WebSphere Application Developer [IBM, WSAD] and Oracle JDeveloper [Oracle, JDeveloper] allow you to build applications using a ton of other Java technologies in conjunction with, or instead of, JSF. For details about JSF support in some popular IDEs, see online extension appendix B.

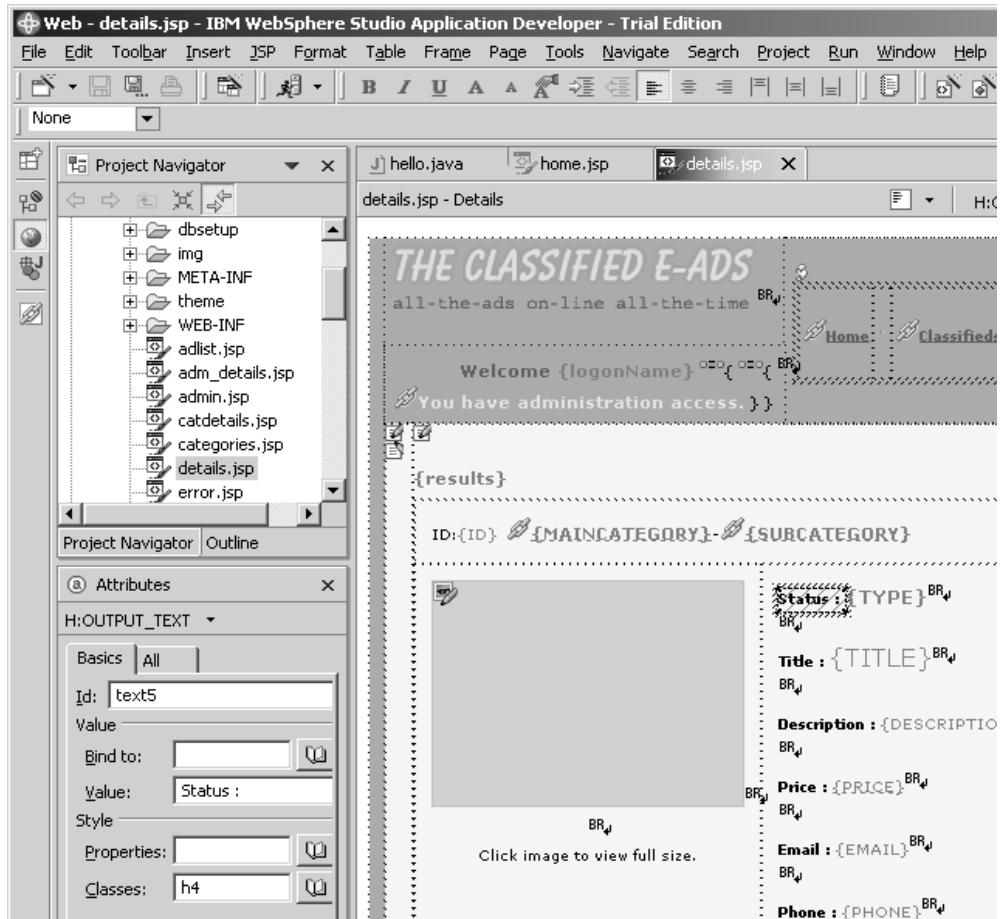


Figure 4.4 Building a JSF page inside of WebSphere Application Developer [IBM, WSAD].

**TIP** If you want your JSF application to be portable across implementations, stick to these standard components and third-party component suites. Don't rely on nonstandard components bundled with integrated development environments (IDEs) or JSF implementations unless you're sure they can be used in other environments, or vendor lock-in is not a concern.

Now that it's clear how tools are involved, let's take a step back and look at where all the HTML is actually generated.

#### 4.1.4 The render kit behind the scenes

As we said in chapter 2, renderers are responsible for displaying JSF components and setting a component's value based on the user's response. Render kits are logical groupings of renderers, and they usually output a specific type of markup or variants of the same markup. For example, you could have a render kit that generates Extensible Markup Language (XML), Wireless Markup Language (WML), Scalable Vector Graphics (SVG), HTML, or just about any other kind of output. You could also have another render kit that generates HTML but uses a lot of client-side script and browser-specific features.

JSF includes a standard HTML render kit that is responsible for providing HTML encoding and decoding behavior for all of the standard components. The render kit is used behind the scenes, so if you're working exclusively with the standard JSF tags or inside of a tool, you won't need to worry about renderers at all.

Render kits can be important when you're developing back-end Java code or writing custom components or renderers. For example, an event listener may need to change the render kit based on the client type or a user's preference. And if you're writing a new component, you may want to associate it with a specific renderer or write a new renderer altogether. We discuss building custom renderers and components in part 4. Now, it's time to learn about the standard UI components.

## 4.2 Common component properties

All of the components covered in this chapter share a set of common properties such as `id` and `value`. Rather than describing these properties each time we discuss a specific component, we have listed them all in table 4.2. During our discussion of each component, we'll tell you which of these properties it supports.

**Table 4.2 Common properties for UI components discussed in this chapter.**

Property	Type	Default Value	Required?	Description
<code>id</code>	String	None	Yes	Component identifier.
<code>value</code>	Object	None	Yes	The component's current local value. Can be literal text or a value-binding expression.
<code>rendered</code>	boolean	true	No	Controls whether or not the component is visible.
<code>converter</code>	Converter instance (value-binding expression or converter identifier)	None	No	Sets the converter used to convert the value to and from a string for display.

*continued on next page*

**Table 4.2** Common properties for UI components discussed in this chapter. (continued)

Property	Type	Default Value	Required?	Description
styleClass	String	None	No	Name of CSS style class; rendered as an HTML <code>class</code> attribute. Multiple classes can be specified with a space in between them.
binding	String	None	No	A value-binding expression that associates this component with a backing bean property. <sup>a</sup>

<sup>a</sup> Technically, this is a JSP component tag attribute, and not an actual UI component property. In other words, you can not set this property in Java code.

### 4.3 Controlling the page with UIViewRoot

It should be clear by now that all of the components on a page are represented in a tree, and that this tree is called a *view*. `UIViewRoot` is the component at the root of this tree. If there's no `UIViewRoot`, there is no view.

This component is unique because it doesn't display anything, and most tools will automatically add it to your JSP (you usually can't actually drag it from the component palette). Also, you can't bind it directly to a backing bean with the `binding` property.<sup>1</sup> Because it has nothing to do with HTML, it's in the core tag library. `UIViewRoot` is summarized in table 4.3.

**Table 4.3** `UIViewRoot` is the container for the entire view.

<b>Component</b>	<code>UIViewRoot</code>
<b>Family</b>	<code>javax.faces.ViewRoot</code>
<b>Possible IDE Display Names</b>	N/A
<b>Display Behavior</b>	Holds all of the child components for a view. Does not display anything.
<b>Tag Library</b>	Core
<b>JSP Tag</b>	<code>&lt;f:view&gt;</code>
<b>Pass-Through Properties</b>	None

continued on next page

<sup>1</sup> You can, however, always access `UIViewRoot` through `FacesContext` in Java code.

**Table 4.3** UIViewRoot is the container for the entire view. (continued)

Property	Type	Default Value	Required?	Description
locale	java.util.Locale	The user's current locale	No	The locale for the current view

So far, we've seen plenty examples of UIViewRoot in use—it's represented by the `<f:view>` tag, as shown in listing 4.1.

**Listing 4.1** UIViewRoot must enclose all other UI components on the same page

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<html>
<head>
  <title>UIViewRoot</title>
</head>
<body bgcolor="#ffffff">
<h1>UIViewRoot example</h1>

<f:view>
  <span style="background-color:grey">
    This is template text.
  </span>
  <h:panelGrid columns="2" border="1">
    <h:outputText value="This is an HtmlOutputText component."/>
    <h:graphicImage url="images/hello.gif"/>
  </h:panelGrid>
</f:view>

</body>
</html>
```

In the listing, UIViewRoot is used inside of template text (pure HTML). It has UI components nested within it (an HtmlPanelGrid with child HtmlOutputText and HtmlGraphicImage components). It also has raw HTML text nested within it. This is an important point: you can mix and match template text with JSF component tags *inside* the `<f:view>` tag. (If you want to use template text inside any other component, see the `<f:verbatim>` tag, covered in section 4.5.2.) You can't, however, use JSF component tags unless they're nested inside `<f:view>`.

Even though listing 4.1 shows a large amount of HTML in the page, it's certainly not required. In part 2, we'll look at developing views that contain hardly any HTML at all.

The only property exposed in `UIViewRoot`'s component tag is `locale`, which allows you to specify the language the current page supports. Suppose you wanted to ensure that a particular page always displays in Spanish:

```
<f:view locale="es">
  ...
</f:view>
```

The string “es” is the locale code for Spanish, so this view will always display in Spanish. You can have only one view per page, so the entire page must be in a single language.

Hard-coding the locale isn't terribly useful unless you're creating different pages for each locale; often you'll use a value-binding expression instead. You certainly don't have to set the locale manually—JSF usually sets it automatically. See chapter 6 for more information about internationalization and localization.

## 4.4 Setting component parameters with `UIParameter`

Most of the time, when a component displays itself, it knows exactly what to do. But sometimes it needs a little help—some extra information to complete the job. This is the purpose of the `UIParameter` component: to provide dynamic variables that can be used during the encoding and decoding processes.

`UIParameter` has no specific HTML equivalent; how it is displayed in a browser depends on the associated component. Three standard components use it: `HtmlOutputFormat`, `HtmlOutputLink`, and `HtmlCommandLink`.

This component's key properties are `name` and `value`. The `name` property is optional, and is only used in cases where the parent component requires a named parameter (for example, `HtmlOutputLink` requires `name`, and `HtmlOutputFormat` doesn't).

Because this component has no direct HTML counterpart, it is part of the core Faces tag library, not the tag library for the HTML render kit. `UIParameter` is summarized in table 4.4.

**Table 4.4** `UIParameter` is used to add parameters to another component.

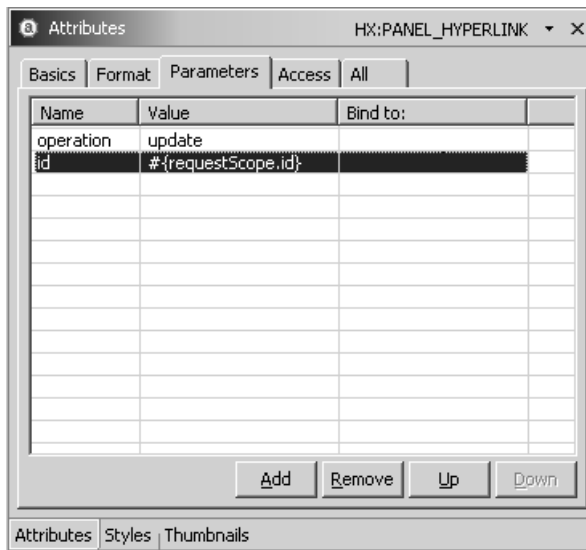
<b>Component</b>	<code>UIParameter</code>
<b>Family</b>	<code>javax.faces.Parameter</code>
<b>Possible IDE Display Names</b>	N/A
<b>Display Behavior</b>	None. Adds a parameter to its parent component.

*continued on next page*

**Table 4.4** `UIParameter` is used to add parameters to another component. (continued)

<b>Tag Library</b>	Core			
<b>JSP Tag</b>	<f:param>			
<b>Pass-Through Properties</b>	None			
<b>Common Properties</b>	id, value, binding (see table 4.2)			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
name	String	None	No	Name of the parameter. Optional for some components.

Unlike most JSF components, when you use `UIParameter` inside of an IDE, you usually don't drag and drop it off the component palette. Instead, you may be able to edit inside a property inspector. For example, IBM WebSphere Application Developer [IBM, WSAD] lets you add parameters through its attribute view, as shown in figure 4.5. The figure shows two named parameters: one named `operation`, with the value "update", and the other named `id`, with the value "`#{requestScope.id}`".



**Figure 4.5**  
**WebSphere Application Developer [IBM, WSAD] allows you to edit parameters with its property inspector.**

In JSP, you add a parameter to a component by nesting it within the parent component's tag:

```
<h:foobar>
  <f:param name="operation" value="update" />
</h:foobar>
```

This adds a `UIParameter` named `operation` with the value `"update"` to a fictional component represented by the component tag `<h:foobar>`. The exact behavior the `UIParameter` affects depends on the mysterious purpose of the `FooBar` component.

As shown in the figure, you can also associate a parameter with a value-binding expression. Using the `name` property is optional as well, as long as the component doesn't need a named parameter.

You'll see more examples of using `UIParameter` when we cover components that use it.

## 4.5 Displaying data with the Output components

---

A large portion of what web applications do is display data. That's the purpose of the Output components. You can specify explicitly the data that they display or have them display backing bean properties. These components will mind their own business—they are read-only, so they will never modify any associated objects.

JSP provides four components for simple output, all of which are part of the Output component family. The `HtmlOutputText` component is used for displaying plain text but also has properties for CSS styles. For displaying pure text with no formatting, you can use the `UIOutput` component. The `HtmlOutputLabel` component is used for attaching labels to input controls. And finally, the `HtmlOutputMessage` component is used for displaying flexible parameterized strings.

Now, let's take a look at each of these components, their behavior, and their properties. These components are indispensable, so you can find more examples throughout the rest of the book.

### 4.5.1 Displaying ordinary text with `HtmlOutputText`

So far you've seen `HtmlOutputText` plenty of times in previous examples; it just displays its value, with optional formatting. Table 4.5 describes this component and its properties.

Table 4.5 `HtmlOutputText` summary

<b>Component</b>	HtmlOutputText			
<b>Family</b>	javax.faces.Output			
<b>Possible IDE Display Names</b>	Output Text			
<b>Display Behavior</b>	Converts the value to a string and displays it with optional support CSS styles. (If the <code>id</code> or <code>style</code> property is set, encloses the text in a <code>&lt;span&gt;</code> element.)			
<b>Tag Library</b>	HTML			
<b>JSP Tag</b>	<code>&lt;h:outputText&gt;</code>			
<b>Pass-Through Properties</b>	style, title			
<b>Common Properties</b>	id, value, rendered, converter, styleClass, binding (see table 4.2)			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
escape	boolean	true	No	Controls whether or not HTML or XML characters are escaped (displayed literally in a browser).

By default, `HtmlOutputText` displays its value directly with no formatting. That's it. No additives—just the plain value.

Special characters are escaped using the appropriate HTML or XML entities by default, so if you embed any markup characters (HTML or XML), they will be displayed literally. An example is shown in table 4.6.

Table 4.6 `HtmlOutputText` example: Text is escaped by default.

<b>HTML</b>	What are <code>&amp;lt;i&gt;</code> you <code>&amp;lt;/i&gt;</code> looking at?
<b>Component Tag</b>	<code>&lt;h:outputText value="What are <code>&lt;i&gt;</code>you<code>&lt;/i&gt;</code> looking at?"/&gt;</code>
<b>Browser Display</b>	What are <code>&lt;i&gt;</code> you <code>&lt;/i&gt;</code> looking at?

As the table shows, by default any markup in `HtmlOutputText`'s value will be displayed literally in a browser. If you want to have the markup passed through directly, you can set the `escape` property to `false`, as shown in table 4.7.

**Table 4.7** `HtmlOutputText` example: Turning off escaping of text.

<b>HTML</b>	What are <i>&lt;i&gt;you&lt;/i&gt;</i> looking at?
<b>Component Tag</b>	<code>&lt;h:outputText value="What are <i>&lt;i&gt;you&lt;/i&gt;</i> looking at?" escape="false" /&gt;</code>
<b>Browser Display</b>	What are <i>you</i> looking at?

In this case, the component's value isn't escaped, so it is interpreted by the browser. Turning off escaping is useful whenever you want the component's value to pass through directly to the client, but if you have larger sets of text and you don't need to use style sheets or HTML formatting, you should use the `UIOutput` component, covered in the next section, instead.

**BY THE WAY**

If you're familiar with JSTL, you may wonder why anyone would use an `HtmlOutputText` component instead of JSTL's `<c:out>` tag. Using `UIOutput` allows you to take full control of the JSF component model. `HtmlOutputText` components are children of other components in the tree, and you can manipulate their behavior in server-side code. If you use a tag like `<c:out>` that isn't backed by a component, there's no server-side representation of the output—in effect, it's more like template text. It's worthwhile, however, to point out that `<c:out>` is more lightweight, because no server-side component is created.

`HtmlOutputText` is great for displaying simple values (with optional CSS formatting), but it doesn't display its body text (you *must* use the `value` property), and you can't embed custom tags in its body. These are features that the `<f:verbatim>` tag offers.

#### 4.5.2 Using `UIOutput` with the `<f:verbatim>` tag

When you want to display literal text without any formatting, or embed other JSP custom tags inside a view, you can use the `UIOutput` component with the `<f:verbatim>` tag. Table 4.8 describes the component and its properties.

**Table 4.8** `UIOutput` summary

<b>Component</b>	UIOutput			
<b>Family</b>	javax.faces.Output			
<b>Display Name</b>	N/A			
<b>Display Behavior</b>	Displays its body without formatting, processing any embedded JSP custom tags			
<b>Tag Library</b>	Core			
<b>JSP Tag</b>	<f:verbatim>			
<b>Pass-Through Properties</b>	None			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
escape	boolean	false	No	Controls whether or not HTML or XML characters are escaped (displayed literally in a browser).

**NOTE** `UIOutput` technically has additional properties; this section is limited to the `<f:verbatim>` tag's use of the component.

As you can see from the table, the `<f:verbatim>` tag doesn't expose a lot of the common UI component properties like `rendered`, `converter`, or `binding`. If you need to use those properties, you're better off with `HtmlOutputText`; `<f:verbatim>` is solely useful for displaying its body content. Unlike `HtmlOutputText`, the `escape` property defaults to `false`.

One of the best uses for `<f:verbatim>` is escaping and displaying a large block of static text. Suppose we wanted to display this snippet of a JSF configuration file in a page, as shown in table 4.9.

Here, we've placed all of the literal text inside the tag's body, which you can't do with `HtmlOutputText`. The `escape` property is also set to `true` to ensure that all of the XML elements are properly escaped. This example results in properly escaped HTML output without the need to manually type in all of those exciting HTML entities.

The `<f:verbatim>` tag is also useful for encasing markup or other tags inside components that are containers, like `HtmlPanelGrid` and `HtmlDataTable`. This is because these components require that all of their children be JSF components, so you can't nest arbitrary template text or other custom tags inside them. You can, however, use the `<f:verbatim>` tag.

**Table 4.9** UIOutput example: Escaping a large block of body text.

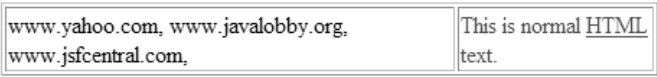
<b>HTML</b>	<pre>&lt;pre&gt;   &amp;lt;application&amp;gt;   &amp;lt;message-bundle&amp;gt;CustomMessages&amp;lt;/message-bundle&amp;gt;   &amp;lt;locale-config&amp;gt;   &amp;lt;default-locale&amp;gt;en&amp;lt;/default-locale&amp;gt;   &amp;lt;supported-locale&amp;gt;en&amp;lt;/supported-locale&amp;gt;   &amp;lt;supported-locale&amp;gt;es&amp;lt;/supported-locale&amp;gt;   &amp;lt;/locale-config&amp;gt;   &amp;lt;/application&amp;gt; &lt;/pre&gt;</pre>
<b>Component Tag</b>	<pre>&lt;pre&gt; &lt;f:verbatim escape="true"&gt;   &lt;application&gt;     &lt;message-bundle&gt;CustomMessages&lt;/message-bundle&gt;     &lt;locale-config&gt;       &lt;default-locale&gt;en&lt;/default-locale&gt;       &lt;supported-locale&gt;en&lt;/supported-locale&gt;       &lt;supported-locale&gt;es&lt;/supported-locale&gt;     &lt;/locale-config&gt;   &lt;/application&gt; &lt;/f:verbatim&gt; &lt;/pre&gt;</pre>
<b>Browser Display</b>	<pre>&lt;application&gt;   &lt;message-bundle&gt;CustomMessages&lt;/message-bundle&gt;   &lt;locale-config&gt;     &lt;default-locale&gt;en&lt;/default-locale&gt;     &lt;supported-locale&gt;en&lt;/supported-locale&gt;     &lt;supported-locale&gt;es&lt;/supported-locale&gt;   &lt;/locale-config&gt; &lt;/application&gt;</pre>

Suppose we wanted to output a simple two-column table, with the left column displaying a list from a backing bean and the right column displaying normal HTML text. There are only three items in the backing bean list: “www.yahoo.com”, “www.javalobby.org”, and “www.jsfcentral.com”.

You can create HTML tables with the `HtmlPanelGrid` component. We’ll cover that component in detail later in this chapter, but for now, let’s just use it to generate the preceding table. For the left column, we’ll use the JSTL `<c:forEach>` tag. For the right column, we’ll just embed literal HTML. The example is shown in table 4.10.

As you can see, the `<c:forEach>` tag and the literal HTML are both embedded in `<f:verbatim>` tags. The `<f:verbatim>` tag is nested inside the `<h:panelGrid>` tag. Only JSF component tags are allowed inside an `<h:panelGrid>`, so using

**Table 4.10** UIOutput example: Embedding custom tags and markup.

<b>HTML</b>	<pre> &lt;table border="1"&gt;   &lt;tbody&gt;     &lt;tr&gt;       &lt;td&gt;www.yahoo.com,         www.javalobby.org,         www.jsfcentral.com       &lt;/td&gt;       &lt;td&gt;         &lt;p&gt;           &lt;font color="red"&gt;This is normal &lt;u&gt;HTML&lt;/u&gt;             text.&lt;/font&gt;         &lt;/p&gt;       &lt;/td&gt;     &lt;/tr&gt;   &lt;/tbody&gt; &lt;/table&gt; </pre>
<b>Component Tag</b>	<pre> &lt;h:panelGrid columns="2" border="1"&gt;   &lt;f:verbatim&gt;     &lt;c:forEach items="\${user.favoriteSites}" var="site"&gt;       &lt;c:out value="\${site}, "/&gt;     &lt;/c:forEach&gt;   &lt;/f:verbatim&gt;   &lt;f:verbatim escape="false"&gt;     &lt;p&gt;&lt;font color="red"&gt;This is normal &lt;u&gt;HTML&lt;/u&gt;       text.&lt;/font&gt;&lt;/p&gt;   &lt;/f:verbatim&gt; &lt;/h:panelGrid&gt; </pre>
<b>Browser Display</b>	

`<f:verbatim>` allows us to treat the JSTL tag and the template text like JSF components. Also, note that the `escape` property is set to `false` for the HTML text because we'd like it to be processed by the browser.

`UIOutput` and `HtmlOutputText` are useful components for displaying simple text, but if you need to associate that text with an input control, you must embed them inside an `HtmlOutputLabel` component.

### 4.5.3 Creating input labels with `HtmlOutputLabel`

The `HtmlOutputLabel` component is used for associating labels with form elements—it maps directly to the HTML `<label>` element. This allows target devices to be smarter about how text is related with controls on the screen so that they

can do such things as highlight a text field when the user clicks on the label. It's also a requirement for creating accessible applications. The component is summarized in table 4.11.

**Table 4.11** `HtmlOutputLabel` summary

<b>Component</b>	HtmlOutputLabel			
<b>Family</b>	javax.faces.Output			
<b>Display Name</b>	Component Label			
<b>Display Behavior</b>	Displays a <label> element. Usually has an <code>HtmlOutputText</code> and/or other components as its children.			
<b>Tag Library</b>	Core			
<b>JSP Tag</b>	<h:outputLabel>			
<b>Pass-Through Properties</b>	HTML attributes for <label> element.			
<b>Common Properties</b>	id, value, rendered, converter, styleClass, binding (see table 4.2)			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
for	String	None	Yes	Component identifier of UI component this label is for.

An `HtmlOutputLabel` component's `for` property is required, and it must be associated with an input control's component identifier. The component doesn't actually display any text—it just renders a <label> element (which has no visual representation). To display anything inside that label, you must nest another UI component. (The child component is usually `HtmlOutputText`; some tools will add it automatically when you drag an `HtmlOutputLabel` component from the palette.)

An example of using `HtmlOutputLabel` for an `HtmlInputText` component is shown in table 4.12.

You can see that the <h:outputLabel> tag maps directly to the <label> tag, and the `accesskey` property is passed through. The `for` property references the `HtmlInputText` component by its component identifier, but the HTML output uses its client identifier (`HtmlOutputLabel` handles this translation for you automatically). This brings up an important point: whenever you're associating

**Table 4.12** `HtmlOutputLabel` example: Adding a label to an `HtmlInputLabel` component.

<b>HTML</b>	<pre>&lt;label for="myForm:userNameInput" accesskey="N"&gt;   Enter your name: &lt;/label&gt; &lt;input id="myForm:userNameInput" type="text" /&gt;</pre>
<b>Component Tag</b>	<pre>&lt;h:outputLabel for="userNameInput" accesskey="N"&gt;   &lt;h:outputText value="Enter your name: " /&gt; &lt;/h:outputLabel&gt; &lt;h:inputText id="userNameInput" /&gt;</pre>
<b>Browser Display</b>	<p>Enter your name: <input type="text"/></p>

`HtmlOutputLabel` with an input control, you have to specify the input control's identifier. Otherwise, you won't know what value to use as the `for` attribute.

In HTML, you can nest an input element within the label tag and leave out the `for` attribute. With JSE, you can certainly nest an input control one or more components inside of an `<h:outputLabel>` tag, but the `for` attribute is mandatory.

Now that we've examined components that display simple strings (or the result of value-binding expressions), let's look at a component that adds some formatting capabilities.

#### 4.5.4 Using `HtmlOutputFormat` for parameterized text

`HtmlOutputFormat` displays parameterized text. In other words, you can specify a special string, and the component will insert values (either hardcoded or defined by value-binding expressions) at specific places in the string. You can also use it to repeat variables in a single string of text without having to repeat the same value-binding expression. The component is summarized in table 4.13.

`HtmlOutputFormat` behaves quite similarly to `HtmlOutputText`, except for the fact that you can insert arbitrary parameters in the string that is displayed.

For example, suppose you wanted to display a message to current users that addressed them by their first name, displayed a hardcoded string, and also displayed their browser type. If the person's name was Joe and he was using Mozilla, the output would look something like this:

```
Hey Mark. This is a static value: hardcoded. Mark, you're using: Mozilla/
5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.2.1) Gecko/20021130.
```

**Table 4.13** `HtmlOutputFormat` summary

<b>Component</b>	HtmlOutputFormat			
<b>Family</b>	javax.faces.Output			
<b>Possible IDE Display Names</b>	Formatted Output			
<b>Display Behavior</b>	Converts its value to a string and displays it, replacing any parameter markers with values retrieved from child <code>UIParameter</code> components. Encloses the value in a <code>&lt;span&gt;</code> element if the <code>id</code> , <code>style</code> , or <code>styleClass</code> property is specified.			
<b>Tag Library</b>	HTML			
<b>JSP Tag</b>	<code>&lt;h:outputFormat&gt;</code>			
<b>Pass-Through Properties</b>	title, style			
<b>Common Properties</b>	id, value, rendered, converter, styleClass, binding (see table 4.2)			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
escape	boolean	false	No	Controls whether or not HTML or XML characters are escaped (displayed literally in a browser).

Assuming you had a `JavaBean` stored under the key `user`, you could generate this output with `HtmlOutputText`:

```
<h:outputText value="Hey #{user.firstName}. This is a static value: hard-coded. #{user.firstName}, you're using: #{header['User-Agent']}."/>
```

This is fine, but you have to use the expression `"#{user.firstName}"` each time you want to display the user's name. (By the way, `header` is an implicit EL variable; see chapter 3 for details.)

`HtmlOutputFormat` simplifies displaying strings with dynamic variables by using message format patterns. A *message format pattern* is a special string with markers that map to parameter values. A *message format element* (not to be confused with an XML or HTML element) is made up of a parameter number surrounded by curly braces (`{}`). The parameters are specified with nested `UIParameter` components. (`UIParameter` is covered in section 4.4.) Table 4.14 shows how to generate our output string with an `HtmlOutputFormat` component instead of an `HtmlOutputText` component.

**Table 4.14** `HtmlOutputFormat` example: Simple parameter substitution.

<b>HTML</b>	Hey Mark. This is a static value: hardcoded. Mark, you're using: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.2.1) Gecko/20021130.
<b>Component Tag</b>	<pre>&lt;h:outputFormat value="Hey {0}. This is a static value: {1}. {0}, you're using: {2}."&gt;   &lt;f:param value="{user.firstName}"/&gt;   &lt;f:param value="hardcoded"/&gt;   &lt;f:param value="{header['User-Agent']}"/&gt; &lt;/h:outputFormat&gt;</pre>
<b>Browser Display</b>	Hey Mark. This is a static value: hardcoded. Mark, you're using: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7b) Gecko/20040316.

Take a look at the `value` property of the `<h:outputFormat>` tag. This is a message format pattern. Like Java, the parameters start counting with zero: `{0}`, `{1}`, `{2}`, and so on. The first parameter value is substituted everywhere the message format element `{0}` exists. The second parameter substitutes for `{1}`, the third for `{2}`, and so on. If you specify extra parameters, the component ignores them. If you specify too few parameters, it will leave the literal text in the string (like `"{1}"`).

**NOTE** `HtmlOutputMessage` is often used to parameterize localized text. See chapter 6 for details.

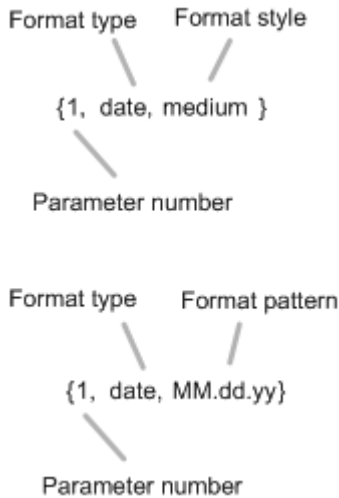
You may have noticed that the string used in the `<h:outputFormat>` tag requires two single quotes to produce the text “you’re” and the string in the `<h:outputText>` tag requires only a single quote. This is a specific requirement of the `HtmlOutputMessage`: you must use two single quotes to produce one. If you want to use curly braces inside a string you’re displaying, you have to enclose them with single quotes as well.

**BY THE WAY** If you’ve ever written Java code using the `MessageFormat` class, you may have noticed that `HtmlOutputFormat` uses the same syntax and abides by similar rules as that class. This is because it uses the `MessageFormat` class internally.

### **Message format elements**

In the last section, we described a message format element as a parameter number surrounded by curly braces—`{1}`, for example. Most of the time, this is all

you need to know. You can, however, format each parameter using a different syntax. After the parameter number, you specify the format type. After the format type, you specify a style or pattern. The parameter number, format type, and format style or pattern are all separated by a comma (,). There are two classes of format types: data type format types (dates, times, and numbers) and choice format types.



**Figure 4.6** Message format elements can specify additional format types. Each format type can have either a style or a pattern. Styles provide some control over formatting, and patterns provide complete control.

Two examples are shown in figure 4.6. The first specifies parameter number one, the date format type, and a format style of medium, which defaults to something like “May 10, 2003”. The second example is also a date format pattern for the first parameter, but has a date format pattern, which specifies the specific format for the date. In this case, it would be something like “05.10.03”.

Let’s say you wanted to display a string using value-binding references for both the name and the date. You could use a message format pattern to format the date property, as shown in table 4.15.

The first parameter uses the normal message element syntax; the second uses the extended syntax for formatting. As the table shows, the medium format element displays an abbreviated month with the day and year displayed as a number for U.S. English. For other languages, the actual result may vary.

**Table 4.15** `HtmlOutputFormat` example: Using the extended syntax for message format elements, with a specific format type.

<b>HTML</b>	Hey Mark, you were born on Apr 20, 2004.
<b>Component Tag</b>	<pre> &lt;h:outputFormat value="Hey {0}, you were born on {1, date, medium}."&gt;   &lt;f:param value="#{user.firstName}"/&gt;   &lt;f:param value="#{user.dateOfBirth}"/&gt; &lt;/h:outputFormat&gt;           </pre>
<b>Browser Display</b>	Hey Mark, you were born on Apr 20, 2004.

If we hadn't specified the date style, the displayed date would have been "04/20/04 06:04 PM", which is the locale's default format for the `Date` object. For the United States, this includes both the date and time; it may be different in other locales. You can leave out the date style in some cases, but usually it's better to have more control over the output. We cover the specific message format types and patterns as well as internationalization in chapter 6.

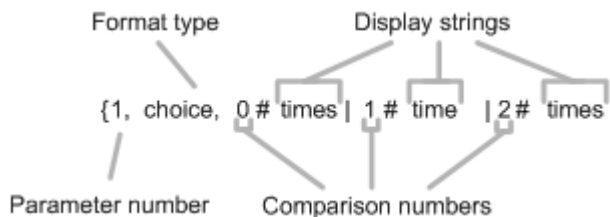
### **Dynamically displaying substrings with choice formats**

Sometimes you may want to display part of a string based on the value of a bean property. For example, let's say that you wanted to tell a user whether an account balance was positive without displaying the actual value. You can do this with a choice format. A *choice format* displays one of several values based on a number parameter. It's great when you want to display the plural form of a word instead of a number. Unlike date type formats, choice formats don't have styles; they only have patterns.

A choice format pattern is made up of a set of comparison value/display string pairs. Within each pair, the comparison value and display string are separated by a number sign (`#`). The comparison value/display string pairs are separated by a pipe separator (`|`), which stands for "or". This syntax is shown in figure 4.7.

The choice pattern works sort of like a case or switch statement. The display string whose comparison number is equal to the parameter's value will be displayed. If no comparison number equals the parameter's value, then the display value whose comparison number is the closest but less than the parameter value will be displayed. If the parameter value is less than the first comparison number, the first display string is chosen. If it's greater, the last display string is chosen.

Table 4.16 shows an example. If `user.numberOfTimes` is less than 1, the display string "times" is chosen. If it's equal to 1, the display string "time" is chosen. If it's greater than or equal to 2, the display string "times" is chosen.



**Figure 4.7**

A choice format allows you to select one display string depending on how its number compares to the parameter value. This example will choose the display string "times" if the parameter value is less than or equal to 0, or greater than 1. Otherwise (if the value equals 1), the string "time" will be displayed.

**Table 4.16** `HtmlOutputFormat` example: Using a choice format for a plural.

<b>HTML (parameter = 0)</b>	You have visited us 0 times.
<b>HTML (parameter = 1)</b>	You have visited us 1 time.
<b>HTML (parameter = 3)</b>	You have visited us 3 times.
<b>Component Tag</b>	<pre>&lt;h:outputFormat     value="You have visited us {0} {0, choice,         0#times 1#time 2#times}."&gt;     &lt;f:param value="#{user.numberOfVisits}"/&gt; &lt;/h:outputFormat&gt;</pre>
<b>Browser Display (parameter = 0)</b>	You have visited us 0 times.
<b>Browser Display (parameter = 1)</b>	You have visited us 1 time.
<b>Browser Display (parameter = 3)</b>	You have visited us 3 times.

Choice formats are a useful tool for displaying complex strings, and don't forget the power of value-binding expressions. You can use them in the value string (with choice or message format patterns), and for the value of any `HtmlOutputFormat` parameters.

#### 4.5.5 Displaying hyperlinks with `HtmlOutputLink`

The `HtmlOutputLink` component maps to an HTML hyperlink or anchor—an `<a>` element. It can be used to link to either an internal or an external URL. The component's `value` property represents the actual URL, and any nested components will be displayed within the hyperlink. If your web container is using URL rewriting to maintain the session, this component will automatically rewrite the URL for you. Table 4.17 summarizes the component's properties.

Most of the time, you'll want to nest an `HtmlOutputText` component inside an `HtmlOutputLink` component to provide the text inside the link. As a matter of fact, some tools, like Sun's Java Studio Creator [Sun, Creator], automatically do this for you. Some tools will also let you select existing pages from within your web application so you don't have to manually type in a relative URL (see figure 4.8).

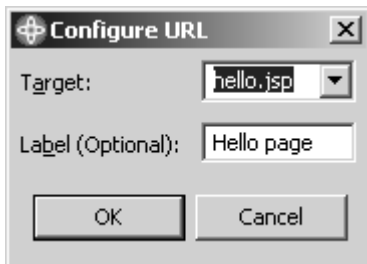
Let's start with a hyperlink to a relative image, shown in table 4.18.

**Table 4.17** `HtmlOutputLink` summary

<b>Component</b>	<code>HtmlOutputLink</code>
<b>Family</b>	<code>javax.faces.Output</code>
<b>Possible IDE Display Names</b>	Hyperlink
<b>Display Behavior</b>	Displays an HTML <code>&lt;a&gt;</code> element. The <code>value</code> property is rendered as the <code>href</code> attribute, and any child components are rendered within the <code>&lt;a&gt;</code> element.
<b>Tag Library</b>	HTML
<b>JSP Tag</b>	<code>&lt;h:outputLink&gt;</code>
<b>Pass-Through Properties</b>	HTML attributes for the <code>&lt;a&gt;</code> element
<b>Common Properties</b>	<code>id</code> , <code>value</code> , <code>rendered</code> , <code>converter</code> , <code>styleClass</code> , <code>binding</code> (see table 4.2)

**Table 4.18** `HtmlOutputLink` example: Simple link to a relative image.

<b>HTML</b>	<code>&lt;a href="images/hello.gif"&gt;Hello image&lt;/a&gt;</code>
<b>Component Tag</b>	<code>&lt;h:outputLink value="images/hello.gif"&gt;   &lt;h:outputText value="Hello image"/&gt; &lt;/h:outputLink&gt;</code>
<b>Browser Display</b>	<u><a href="#">Hello image</a></u>

**Figure 4.8** WebSphere Application Developer [IBM, WSAD] allows you to select a page in your application and set a label with a simple dialog box.

You can see that the component's value maps to the `href` value of the hyperlink, and the text within the hyperlink is from the nested `HtmlOutputText` component.

Although a single `HtmlOutputText` child component is a really common use case, you can also nest other components, like an `HtmlGraphicImage` component, and even `HtmlPanelGrids`. You can also nest more than one component.

In the real world, you often need to add parameters to the URL, as shown in table 4.19.

**Table 4.19** `HtmlOutputLink` example: Passing URL parameters.

<b>HTML</b>	<code>&lt;a href="http://groups.google.com/groups?group=comp&amp;hl=fr"&gt; Google Groups - Computers (in French) &lt;/a&gt;</code>
<b>Component Tag</b>	<code>&lt;h:outputLink value="http://groups.google.com/groups"&gt;   &lt;h:outputText value="Google Groups - Computers (in French)"/&gt;   &lt;f:param name="group" value="comp"/&gt;   &lt;f:param name="hl" value="fr"/&gt; &lt;/h:outputLink&gt;</code>
<b>Browser Display</b>	<u><a href="#">Google Groups - Computers (in French)</a></u>

As you can see, `HtmlOutputLink` will automatically append nested `UIParameter` instances to URL parameters. In this example, two `UIParameter` instances were used to output the parameters at the end of the `href` property. The first parameter tells Google to display all of the newsgroups that start with “comp”. The second tells it to use the French language. So, if you click on this link, you’ll see all of the `comp.*` newsgroups in French.

You can use `HtmlOutputLink` this way for external pages as well as other links within your application or site. And remember, `UIParameters`, like most JSF components, can also use value-binding expressions, so the parameters can come from beans in your application.

**BY THE WAY**

If you’re wondering why anyone would bother using this `HtmlOutputLink` instead of a normal HTML `<a>` element, there are two good reasons. First, you can’t lay out normal HTML elements within panels. So the minute you start using panels for grouping and layout, you must use JSF components. Second, using `UIParameters` allows you to dynamically configure the values being sent to the external URL, which means you can easily sync them up with backing beans.

`HtmlOutputLink` is the last of the Output family of components. Earlier, we used `HtmlGraphicImage` to display an image inside a hyperlink. Let’s take a closer look at this component.

## 4.6 Displaying images with *HtmlGraphicImage*

Most UIs have images somewhere—little icons that represent menu options, logos, or just something to spice things up. In JSF, images are handled by the

`HtmlGraphicImage` component, a read-only reference to a graphic. The component displays an `<img>` element whose `src` attribute is set to the current value of the component (represented by the `url` property), which should be a static string or value-binding expression for the image's URL. Table 4.20 summarizes this component.

**Table 4.20** `HtmlGraphicImage` summary

<b>Component</b>	<code>HtmlGraphicImage</code>			
<b>Family</b>	<code>javax.faces.Graphic</code>			
<b>Possible IDE Display Names</b>	Image			
<b>Display Behavior</b>	Displays an <code>&lt;img&gt;</code> element with the <code>src</code> attribute equal to the component's <code>url</code> property. Automatically encodes the URL to maintain the session, if necessary.			
<b>Tag Library</b>	HTML			
<b>JSP Tag</b>	<code>&lt;h:graphicImage&gt;</code>			
<b>Pass-Through Properties</b>	HTML attributes for the <code>&lt;img&gt;</code> element			
<b>Common Properties</b>	<code>id</code> , <code>value</code> , <code>rendered</code> , <code>styleClass</code> , <code>binding</code> (see table 4.2)			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
<code>url</code>	String	None	No	The URL of the image to be displayed. Can be literal text or a value-binding expression. (This is an alias for the <code>value</code> property.)

If your web container is using URL rewriting to maintain the session, `HtmlGraphicImage` will automatically rewrite the URL for you. Also, if your URL begins with a slash (`/`), it will be relative to the web application's context root. For example, see table 4.21.

As you can see, the `url` property was rendered as a `src` attribute, prefixed with the application's context root name (no session was active, so the URL was not rewritten). If we had used a relative URL like "images/logo.gif", the context root name would not have been added. All of the other properties were passed through.

Remember that `HtmlGraphicImage`, like most JSF components, supports value-binding expressions. This can be useful for situations where you maintain the URLs for graphics in a central location, like a database or an XML file.

**Table 4.21** `HtmlGraphicImage` example: URL relative to web application root.

<b>HTML</b>	<pre>&lt;img src="/jia-standard-components/images/logo.gif"       alt="Welcome to ProjectTrack" height="160"       title="Welcome to ProjectTrack" width="149" /&gt;</pre>
<b>Component Tag</b>	<pre>&lt;h:graphicImage url="/images/logo.gif"                 alt="Welcome to ProjectTrack"                 title="Welcome to ProjectTrack" width="149"                 height="160"/&gt;</pre>
<b>Browser Display</b>	

We present more examples of using `HtmlGraphicImage` in part 2. The topic of the next section, however, is displaying application messages.

## 4.7 Displaying component messages with `HtmlMessage`

In chapter 2, we touched on JSF's support for messages that report validation and conversion errors, as well as general-purpose information from the application itself. Zero or more messages can be generated when JSF processes a request. Every message has a severity level as well as summary and detailed information, and the information is usually localized for the user's current language. The severity levels are listed in table 4.22.

**Table 4.22** Messages have a severity, which is equal to one of these values.

Severity Level	Description
Info	Represents text you'd like to send back to the user that isn't an error.
Warn	Indicates that an error <i>may</i> have occurred.
Error	Indicates a definite error. Recommended for validation messages.
Fatal	Indicates a serious error.

You can probably see where all this is leading—`HtmlMessage` displays application messages. Actually, it displays a *single* message that's associated with a *specific* UI component. It's useful for displaying validation or conversion errors for an input

control, and usually the error is displayed next to the component (so the user knows where to correct the problem). If more than one message is registered for a component (which can happen if more than one validator is registered, or if a validator has problems with conversion), `HtmlMessage` displays only the first one. The component is summarized in table 4.23.

As the table shows, `HtmlMessage` lets you change the style based on the severity level of the message. This allows you to give the user better visual clues—for example, red text may mean a real problem, but blue text might just be an informational message. In addition, you can control whether or not you want the user to see the summary or detail of the message. Let's start with the example, shown in table 4.24.

**Table 4.23** `HtmlMessage` summary

<b>Component</b>	HtmlMessage			
<b>Family</b>	javax.faces.Message			
<b>Possible IDE Display Names</b>	Display Error, Inline Message			
<b>Display Behavior</b>	Displays the first messages registered for the component referenced by the <code>for</code> property. If the <code>id</code> , a tooltip, or any CSS styles are specified, the text will be wrapped in a <code>&lt;span&gt;</code> element.			
<b>Tag Library</b>	HTML			
<b>JSP Tag</b>	<code>&lt;h:message&gt;</code>			
<b>Pass-Through Properties</b>	style, title			
<b>Common Properties</b>	id, rendered, styleClass, binding (see table 4.2)			
Property	Type	Default Value	Required?	Description
<code>for</code>	String	None	Yes	The component identifier of the component for which messages should be displayed.
<code>showDetail</code>	boolean	false	No	Indicates whether or not to show the detail portion of the message.
<code>showSummary</code>	boolean	true	No	Indicates whether or not to show the summary portion of the message.

*continued on next page*

**Table 4.23** `HtmlMessage` summary (continued)

Property	Type	Default Value	Required?	Description
<code>errorClass</code>	<code>String</code>	None	No	CSS class for messages with Error severity.
<code>errorStyle</code>	<code>String</code>	None	No	CSS style for messages with Error severity.
<code>fatalClass</code>	<code>String</code>	None	No	CSS class for messages with Fatal severity.
<code>fatalStyle</code>	<code>String</code>	None	No	CSS style for messages with Fatal severity.
<code>infoClass</code>	<code>String</code>	None	No	CSS class for messages with Info severity.
<code>infoStyle</code>	<code>String</code>	None	No	CSS style for messages with Info severity.
<code>warnClass</code>	<code>String</code>	None	No	CSS class for messages with Warning severity.
<code>warnStyle</code>	<code>String</code>	None	No	CSS style for messages with Warning severity.
<code>tooltip</code>	<code>boolean</code>	<code>false</code>	No	Indicates whether or not the message detail should be displayed as a tooltip. Only valid if <code>showDetail</code> is <code>true</code> .

In this example, we’ve associated an `HtmlMessage` component with an `HtmlInputText` that has a `Length` validator and a `LongRange` validator. As we’ve seen so far, `HtmlInputText` simply collects input. The `Length` validator verifies the length of the input, and the `LongRange` validator checks to make sure the input is an integer in the proper range. (`HtmlInputText` is covered in chapter 5, and validators are covered in chapter 6.)

In this example, the input fails both validators, so two messages are generated. However, `HtmlMessage` displays only the first one.

We actually see the error message twice—once for the summary, and once for the detail. This is sort of a quirk with the RI—both the summary and the detail are the same. Other implementations and custom validators should have a distinctive detail message. For example, if the error was “Invalid credit card number”, the detail might be “The expiration date is invalid.”

This example also shows use of the style properties that format messages differently based on the severity. The rendered style is “color: red”, which is the

**Table 4.24** `HtmlMessage` example: Showing summary and detail and applying multiple styles.

<b>HTML (with two validation errors)</b>	<pre> Enter text: &lt;input id="_id0:myOtherInput" type="text"       name="_id0:myOtherInput" value="this is text" /&gt; &lt;span style="color: red"&gt;Validation Error: Value is greater than allowable maximum of '3'. Validation Error: Value is greater than allowable maximum of '3'. &lt;/span&gt; </pre>
<b>Component Tag</b>	<pre> &lt;h:outputLabel for="validatorInput"&gt;   &lt;h:outputText value="Enter text:" /&gt; &lt;/h:outputLabel&gt; &lt;h:inputText id="myOtherInput"&gt;   &lt;f:validateLength minimum="0" maximum="3" /&gt;   &lt;f:validateLongRange minimum="1" maximum="100" /&gt; &lt;/h:inputText&gt; &lt;h:message for="myOtherInput" showDetail="true"            showSummary="true" warnStyle="color: green"            infoStyle="color: blue" errorStyle="color: red" /&gt; </pre>
<b>Browser Display (with two validation errors)</b>	<p>Enter text: <input type="text" value="this is text"/> Validation Error: Value is greater than allowable maximum of '3'.</p> <p>Validation Error: Value is greater than allowable maximum of '3'.</p>

value of the `errorStyle` property. This means that the message's severity level was `Error`. (All of the standard validation messages have this severity level.) If the message's severity level had been `Warn`, the `warnStyle` property would have been used; if it had been `Info`, the `infoStyle` property would have been used.

`HtmlMessage` is designed to display messages for a specific component, and is useful whenever you need to inform the user of a mistake in a particular input control. It's typically used with input controls, but that's not a requirement. If you need to display multiple messages for a page, you should use `HtmlMessages` instead.

## 4.8 Displaying application messages with `HtmlMessages`

When JSF processes a request, multiple parts of the application—validators, converters, event listeners, and so on—can generate messages. (See the previous section for a quick overview of messages.) Because messages can be generated from so many sources, it's no surprise that you often end up with multiple messages that need to be displayed to the user. This is the job of the `HtmlMessages` component, which is summarized in table 4.25.

Like `HtmlMessage`, `HtmlMessages` lets you change the style based on the severity level of the message. So, if it's displaying several messages, each one can be in a different color. The first message might have the Info severity level and be displayed in blue, the second might have the Error severity level and be displayed in red, and so on.

Because `HtmlMessages` displays all messages, it doesn't have a `for` property that limits it to displaying messages for a specific component. You can, however, tell it to display only messages that aren't associated with components with the `globalOnly` property. This is useful if you want to display messages that were created in application code (like an event listener) as opposed to ones that were created by validators or converters.

**Table 4.25** `HtmlMessages` summary

<b>Component</b>	<code>HtmlMessages</code>			
<b>Family</b>	<code>javax.faces.Messages</code>			
<b>Possible IDE Display Names</b>	Message List			
<b>Display Behavior</b>	Displays all messages if the <code>globalOnly</code> property is <code>false</code> . If <code>globalOnly</code> is <code>true</code> , displays only messages that are not associated with a component (these are usually messages from event listeners). If the <code>layout</code> property is "table", a <code>&lt;table&gt;</code> element will be displayed, with each message on a single row. If the <code>id</code> and <code>tooltip</code> properties, or any CSS styles are specified, the text for each message will be wrapped in a <code>&lt;span&gt;</code> element.			
<b>Tag Library</b>	HTML			
<b>JSP Tag</b>	<code>&lt;h:messages&gt;</code>			
<b>Pass-Through Properties</b>	<code>style</code> , <code>title</code>			
<b>Common Properties</b>	<code>id</code> , <code>rendered</code> , <code>styleClass</code> , <code>binding</code> (see table 4.2)			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
<code>showDetail</code>	<code>boolean</code>	<code>false</code>	No	Indicates whether or not to show the detail portion of the message.

*continued on next page*

**Table 4.25** `HtmlMessages` summary (continued)

Property	Type	Default Value	Required?	Description
<code>showSummary</code>	boolean	true	No	Indicates whether or not to show the summary portion of the message.
<code>layout</code>	String	"list"	No	Specifies how to display the messages. Possible values are "list", which displays them one after another, and "table", which displays them in table columns.
<code>errorClass</code>	String	None	No	CSS class for messages with Error severity.
<code>errorStyle</code>	String	None	No	CSS style for messages with Error severity
<code>fatalClass</code>	String	None	No	CSS class for messages with Fatal severity.
<code>fatalStyle</code>	String	None	No	CSS style for messages with Fatal severity.
<code>infoClass</code>	String	None	No	CSS class for messages with Info severity.
<code>infoStyle</code>	String	None	No	CSS style for messages with Info severity.
<code>warnClass</code>	String	None	No	CSS class for messages with Warning severity.
<code>warnStyle</code>	String	None	No	CSS style for messages with Warning severity.
<code>tooltip</code>	boolean	false	No	Indicates whether or not the message detail should be displayed as a tooltip. Only valid if <code>showDetail</code> is true.
<code>globalOnly</code>	boolean	false	No	Controls whether or not the component only displays global messages (as opposed to both global messages and messages for a specific component).

Let's start with the simple example, shown in table 4.26. This table shows two sets of output depending on the type of messages available. First, there is the

output with two validation messages. These errors happen to be the same two messages generated by the example in table 4.25. The difference is that here, both of them are displayed.

**Table 4.26** *HtmlMessages* example: Simple usage with both validation and application messages.

<b>HTML (with two validation errors)</b>	<pre>&lt;span class="errors"&gt;Validation Error: Value is greater than allowable maximum of '3'. &lt;/span&gt; &lt;span class="errors"&gt;Validation Error: Value is not of the correct type. &lt;/span&gt;</pre>
<b>HTML (with two application messages)</b>	<pre>&lt;span class="errors"&gt;Your order has been processed successfully. &lt;/span&gt; &lt;span class="errors"&gt;Free shipping limit exceeded. &lt;/span&gt;</pre>
<b>Component Tag</b>	<pre>&lt;h:messages styleClass="errors"/&gt;</pre>
<b>Browser Display (with two validation errors)</b>	<pre>Validation Error: Value is greater than allowable maximum of '3'. Validation Error: Value is not of the correct type.</pre>
<b>Browser Display (with two application messages)</b>	<pre>Your order has been processed successfully. Free shipping limit exceeded.</pre>

The second set of output is for application messages rather than validation messages. The component isn't displaying both types of messages because validation errors usually prevent event listeners from being called. So, in this case, the event listener won't generate any messages unless all of the input is valid.

One subtle point with the last example is that even though the two application messages have different severity levels, they are both displayed with the same CSS class. Because *HtmlMessages* can display several different messages (each of which can have different severity levels), it's generally a good idea to apply different styles so that users can differentiate them.

**TIP** It's often helpful to place an *HtmlMessages* component at the top of your page during development. This will allow you to see validation and conversion errors that may pop up while you're building the application.

If your views have a lot of forms, you'll be using `HtmlMessage` and `HtmlMessages` quite often. They're key for keeping the user informed about input errors, and they greatly simplify the process of reporting such errors. These components are also handy for displaying messages generated by your application.

## 4.9 Grouping and layout with the Panel components

---

If you've worked with Swing, or tools such as Visual Basic or Delphi, then you're probably familiar with the concept of a panel. Panels are often used to group related components. Once you've placed some components inside a panel, you can manipulate them as a single unit simply by interacting with the panel. You can also format the panel with something like a border and hide or display it (and all of its child controls) depending on the application's state.

Panels in JSF are similar to these panels, but they're a little different, too. Their primary goal is to group components together, but sometimes they handle layout as well. As a matter of fact, panels are the only standard way to handle layout within a single view. Using a combination of different panels, you can achieve complex organization of controls on a page.

JSF ships with two panel components: `HtmlPanelGroup` and `HtmlPanelGrid`. `HtmlPanelGroup` simply groups all child components together, and optionally applies CSS styles. `HtmlPanelGrid` can be used for very configurable layouts, sort of like a `GridLayout` in Swing; it renders an HTML `<table>` element.

In the following sections, we examine these components in more detail.

### 4.9.1 Grouping components with `HtmlPanelGroup`

The `HtmlPanelGroup` component groups a set of components together so that they can be treated as a single entity. It doesn't map directly to an HTML element. As a matter of fact, the only time it outputs anything is when you specify an identifier or a style, in which case the child components will be enclosed in a `<span>` element. It does, however, display all of its child components without modification. The component is summarized in table 4.27.

Take a look at the simple example shown in table 4.28. In this case, `HtmlPanelGroup` doesn't display anything—the child components are just displayed as is. You'll often use it this way to group together components within a facet; this is quite common, for example, when you're defining the header and footers within an `HtmlPanelGrid` or `HtmlDataTable` (see those sections for more examples).

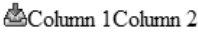
**Table 4.27** `HtmlPanelGroup` summary

<b>Component</b>	<code>HtmlPanelGroup</code>
<b>Family</b>	<code>javax.faces.Panel</code>
<b>Possible IDE Display Names</b>	Panel – Group Box, Group Box
<b>Display Behavior</b>	All child components are displayed as is. If the <code>id</code> , <code>style</code> , or <code>styleClass</code> properties are specified, encloses all child components in a <code>&lt;span&gt;</code> element. Used to group child components together.
<b>Tag Library</b>	HTML
<b>JSP Tag</b>	<code>&lt;h:panelGroup&gt;</code>
<b>Pass-Through Properties</b>	<code>style</code>
<b>Common Properties</b>	<code>id</code> , <code>rendered</code> , <code>styleClass</code> , <code>binding</code> (see table 4.2)

**TIP** `HtmlPanelGroup` can be useful as a placeholder. Use it when you want to create a blank cell inside a table rendered by `HtmlPanelGrid` or `HtmlDataTable`.

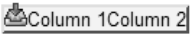
`HtmlPanelGroup` can also be used to add simple formatting to a group of components, as shown in table 4.29.

**Table 4.28** `HtmlPanelGroup` example: Grouping three components with no style.

<b>HTML</b>	<code>&lt;img src="images/inbox.gif" alt="" /&gt;Column 1Column 2</code>
<b>Component Tag</b>	<pre>&lt;h:panelGroup&gt;   &lt;h:graphicImage url="images/inbox.gif"/&gt;   &lt;h:outputText value="Column 1"/&gt;   &lt;h:outputText value="Column 2"/&gt; &lt;/h:panelGroup&gt;</pre>
<b>Browser Display</b>	

In this example, the component outputs a `<span>` element with its client identifier and the specified CSS class, providing a nice background and border for the child components. This type of formatting is nice for simple cases, but for laying out components, you should use `HtmlPanelGrid` instead.

**Table 4.29** `HtmlPanelGroup` example: Grouping three components with a style.

<b>HTML</b>	<pre>&lt;span id="_id1:myGroup" class="table-background"&gt;   &lt;img src="images/inbox.gif" alt="" /&gt;Column 1Column 2 &lt;/span&gt;</pre>
<b>Component Tag</b>	<pre>&lt;h:panelGroup id="myGroup" styleClass="table-background"&gt;   &lt;h:graphicImage url="images/inbox.gif" /&gt;   &lt;h:outputText value="Column 1" /&gt;   &lt;h:outputText value="Column 2" /&gt; &lt;/h:panelGroup&gt;</pre>
<b>Browser Display</b>	

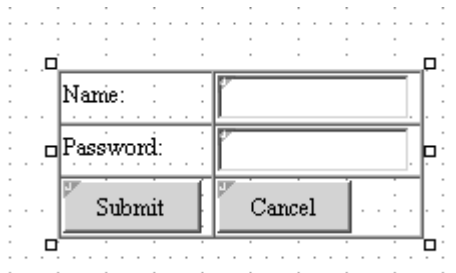
### 4.9.2 Creating tables with `HtmlPanelGrid`

`HtmlPanelGrid` is useful for creating arbitrary, static component layouts (it maps to the `<table>` element). You can also configure header and footer with facets that map to the `<thead>` and `<tfoot>` table subelements, respectively. Table 4.30 summarizes this component.

You can expect tools to render a table for you in real time, as you drag and drop controls into an `HtmlPanelGrid` from a component palette (see figure 4.9). As you can see, a common use of this component is to lay out forms, like a login form.

We'll look at the JSP for more complicated views like a login form in part 2. For now, let's start with a simple three-column, two-row table where each cell contains a single `HtmlOutputText` component. This is shown in table 4.31.

As you can see, child components are organized according to the specified number of columns. Because we specified three columns, the first three components formed the first row (one per column), the next three formed the second row, and so on. The `width`, `border`, and `cellpadding` properties were passed through. Note that unlike with an HTML table, you don't have to explicitly denote columns and rows—you just embed the child components inside the panel, and it will do the rest.



**Figure 4.9** Most JSF IDEs, like Java Studio Creator [Sun, Creator], allow you to drag and drop components into an `HtmlPanelGrid` and change the table's layout when you modify the component's properties.

Table 4.30 HtmlPanelGrid summary

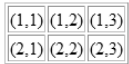
<b>Component</b>	HtmlPanelGrid			
<b>Family</b>	javax.faces.Panel			
<b>Possible IDE Display Names</b>	Grid Panel			
<b>Display Behavior</b>	Displays an HTML <table> element with the specified number of columns. Lays out one child component per cell, starting a new row after displaying columns components. If the header facet is specified, displays a <thead> element with the contents of the header. If the footer facet is specified, displays a <tfoot> element with the contents of the footer.			
<b>Tag Library</b>	HTML			
<b>JSP Tag</b>	<h:panelGrid>			
<b>Pass-Through Properties</b>	HTML attributes for <table>			
<b>Common Properties</b>	id, rendered, styleClass, binding (see table 4.2)			
<b>Property</b>	<b>Type</b>	<b>Default Value</b>	<b>Required?</b>	<b>Description</b>
columns	int	None	No	Number of columns to display.
headerClass	String	None	No	Name of CSS style class for the header facet.
footerClass	String	None	No	Name of CSS style class for the footer facet.
rowClasses	String	None	No	Comma-delimited list of CSS style classes for the rows. You can specify multiple styles for a row by separating them with a space. After each style has been applied, they repeat. For example, if there are two style classes (style1 and style2), the first row will be style1, the second style2, the third style1, the fourth style2, so on.

continued on next page

**Table 4.30** `HtmlPanelGrid` summary (continued)

Property	Type	Default Value	Required?	Description
<code>columnClasses</code>	String	None	No	Comma-delimited list of CSS style classes for the columns. You can specify multiple styles for a column by separating them with a space. After each style has been applied, they repeat. For example, if there are two style classes ( <code>style1</code> and <code>style2</code> ), the first column will be <code>style1</code> , the second <code>style2</code> , the third <code>style1</code> , the fourth <code>style2</code> , so on.
Facet	Description			
<code>header</code>	Child components displayed as the table's header.			
<code>footer</code>	Child components displayed as the table's footer.			

**Table 4.31** `HtmlPanelGrid` example: A simple three-column, two-row table.

HTML	<pre>&lt;table border="1" cellpadding="1" width="40%"&gt;   &lt;tbody&gt;     &lt;tr&gt;       &lt;td&gt; (1,1) &lt;/td&gt;       &lt;td&gt; (1,2) &lt;/td&gt;       &lt;td&gt; (1,3) &lt;/td&gt;     &lt;/tr&gt;     &lt;tr&gt;       &lt;td&gt; (2,1) &lt;/td&gt;       &lt;td&gt; (2,2) &lt;/td&gt;       &lt;td&gt; (2,3) &lt;/td&gt;     &lt;/tr&gt;   &lt;/tbody&gt; &lt;/table&gt;</pre>
Component Tag	<pre>&lt;h:panelGrid columns="3" cellpadding="1" border="1" width="40%"&gt;   &lt;h:outputText value=" (1,1) " /&gt;   &lt;h:outputText value=" (1,2) " /&gt;   &lt;h:outputText value=" (1,3) " /&gt;   &lt;h:outputText value=" (2,1) " /&gt;   &lt;h:outputText value=" (2,2) " /&gt;   &lt;h:outputText value=" (2,3) " /&gt; &lt;/h:panelGrid&gt;</pre>
Browser Display	

**TIP** Currently, there is no guaranteed default number of columns (the RI defaults to one). So, if you want to ensure the same behavior across different implementations, always specify the number of columns.

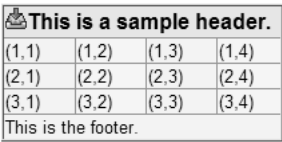
It's also quite easy to add styles to different columns or rows, and to add a header and footer. Table 4.32 shows how to do this.

**Table 4.32** `HtmlPanelGrid` example: A table with a header, a footer, and alternating styles for the columns.

<b>HTML</b>	<pre> &lt;table class="table-background" border="1" cellpadding="1" width="40%"&gt;   &lt;thead&gt;     &lt;tr&gt;       &lt;th class="page-header" colspan="4" scope="colgroup"&gt;         &lt;img src="images/inbox.gif" alt="" /&gt;This is a sample header. &lt;/th&gt;     &lt;/tr&gt;   &lt;/thead&gt;   &lt;tfoot&gt;     &lt;tr&gt;       &lt;td class="table-footer" colspan="4"&gt;This is the footer.     &lt;/td&gt;     &lt;/tr&gt;   &lt;/tfoot&gt;   &lt;tbody&gt;     &lt;tr&gt;       &lt;td class="table-odd-column"&gt;(1,1) &lt;/td&gt;       &lt;td class="table-even-column"&gt;(1,2) &lt;/td&gt;       &lt;td class="table-odd-column"&gt;(1,3) &lt;/td&gt;       &lt;td class="table-even-column"&gt;(1,4) &lt;/td&gt;     &lt;/tr&gt;     &lt;tr&gt;       &lt;td class="table-odd-column"&gt;(2,1) &lt;/td&gt;       &lt;td class="table-even-column"&gt;(2,2) &lt;/td&gt;       &lt;td class="table-odd-column"&gt;(2,3) &lt;/td&gt;       &lt;td class="table-even-column"&gt;(2,4) &lt;/td&gt;     &lt;/tr&gt;     &lt;tr&gt;       &lt;td class="table-odd-column"&gt;(3,1) &lt;/td&gt;       &lt;td class="table-even-column"&gt;(3,2) &lt;/td&gt;       &lt;td class="table-odd-column"&gt;(3,3) &lt;/td&gt;       &lt;td class="table-even-column"&gt;(3,4) &lt;/td&gt;     &lt;/tr&gt;   &lt;/tbody&gt; &lt;/table&gt; </pre>
-------------	--

*continued on next page*

**Table 4.32** `HtmlPanelGrid` example: A table with a header, a footer, and alternating styles for the columns. (continued)

<b>Component Tag</b>	<pre> &lt;h:panelGrid columns="4" styleClass="table-background"              headerClass="page-header"              columnClasses="table-odd-column, table-even-column"              footerClass="table-footer"              cellpadding="1" border="1" width="40%"&gt;  &lt;f:facet name="header"&gt;   &lt;h:panelGroup&gt;     &lt;h:graphicImage url="images/inbox.gif"/&gt;     &lt;h:outputText value="This is a sample header."/&gt;   &lt;/h:panelGroup&gt; &lt;/f:facet&gt; &lt;h:outputText value="(1,1)"/&gt; &lt;h:outputText value="(1,2)"/&gt; &lt;h:outputText value="(1,3)"/&gt; &lt;h:outputText value="(1,4)"/&gt; &lt;h:outputText value="(2,1)"/&gt; &lt;h:outputText value="(2,2)"/&gt; &lt;h:outputText value="(2,3)"/&gt; &lt;h:outputText value="(2,4)"/&gt; &lt;h:outputText value="(3,1)"/&gt; &lt;h:outputText value="(3,2)"/&gt; &lt;h:outputText value="(3,3)"/&gt; &lt;h:outputText value="(3,4)"/&gt; &lt;f:facet name="footer"&gt;   &lt;h:outputText value="This is the footer."/&gt; &lt;/f:facet&gt; &lt;/h:panelGrid&gt; </pre>
<b>Browser Display</b>	 <p>The browser display shows a table with a header row containing an image icon and the text "This is a sample header.". Below the header is a 3x4 grid of cells. The first row contains cells with values (1,1), (1,2), (1,3), and (1,4). The second row contains cells with values (2,1), (2,2), (2,3), and (2,4). The third row contains cells with values (3,1), (3,2), (3,3), and (3,4). Below the grid is a footer row containing the text "This is the footer.".</p>

**NOTE** You can also style the rows with the `rowClasses` attribute—it works just like `columnClasses`.

You can see that the `styleClass` property specifies the CSS class for the entire table. The class attributes of the table columns alternate between the two values specified in the `columnClasses` properties. There is no limit to the number of classes you can specify. The header is displayed as a single row spanning all columns, and is styled with the CSS class specified by the `headerClass` attribute. The

footer also spans all columns, but it is styled with the CSS class specified by the `footerClass` attribute.

Note that the header facet uses an `HtmlPanelGroup` component; this is required if you'd like to include several components inside of a facet (you could also use another container, like an additional `HtmlPanelGrid`). The footer facet has only one child component, so no nested `HtmlPanelGroup` is necessary.

In this example, we alternated styles for the columns. It's worthwhile to note that you can have more than two different styles. Also, just like `styleClass`, all of the other class properties, including the ones for columns and rows, can support more than one style.

```
<h:panelGrid columns="4"
             headerClass="page-header extra-border"
             columnClasses="table-odd-column extra-border,
                           table-even-column,
                           table-even-column extra-border,
                           table-even-column"
             footerClass="table-footer"
             cellpadding="1" border="1" width="40%">
...
</h:panelGrid>
```

There are two things to note in this example. First, the `headerClass` has two classes—`page-header` and `extra-border`. Both will be applied to the header facet. In addition, the `columnClasses` property has classes specified for four rows. The first column has an extra style applied to it, so it will display with both styles combined. The other three columns use a different style, and the second-to-last one has an extra style applied to it as well. If there were more than four columns, these styles would repeat.

You can use the `rowClasses` property just like the `columnClasses` property, and you can use them at the same time. This can lead to some interesting style combinations, which can be a good thing or a bad thing, depending on how your style classes are set up. (If you're not careful, you can have conflicts.)

These simple examples should get you started with `HtmlPanelGrid`. You can accomplish complex layouts by nesting several panels, just as you can with HTML tables. We show more complex examples in part 2. If you need to lay out tabular data retrieved from a data source, `HtmlDataTable` is your component, and it's covered in chapter 5.

## 4.10 Summary

---

The core promise of JSF is support for UI components—reusable units of code, like text boxes and data tables, that handle a specific type of interaction with the user. Like other UI component frameworks, JSF components have properties and events, and they’re specifically designed to be used inside tools. These tools let you drag and drop UI components from a palette and customize their properties.

JSF includes several standard components that are guaranteed to be available no matter which IDE or implementation you’re using. These components are tailored for displaying HTML output, and they’re backed by an HTML render kit. Components can also support facets, which are named subordinate elements, like a header or footer.

In this chapter, we covered the nonvisual and read-only standard components. First, we looked at `UIViewRoot`, which contains all of the other components on the page. Next, we examined `UIParameter`, which is used to pass parameters to other components. For example, `HtmlOutputLink` uses it to specify parameters for a hyperlink.

We then moved on to the Output family of components: `HtmlOutputText`, which displays ordinary text; the `<f:verbatim>` tag, which encapsulates JSP tags and template text; `HtmlOutputLabel`, which provides a label for an input field; `HtmlOutputFormat`, which outputs parameterized text; and `HtmlOutputLink`, which displays a hyperlink.

Images in JSF are displayed with the `HtmlGraphicImage` component, and application, validation, and conversion messages are the domain of `HtmlMessage` and `HtmlMessages`. Finally, the Panel components group together components as a single entity. `HtmlPanelGroup` is generally used to group together components, and `HtmlPanelGrid` lays out components in a table, with optional header and footer facets.

In the next chapter we look at the other side of the coin—the standard input controls and JSF’s data grid component, `HtmlDataTable`.

# JavaServer Faces IN ACTION

Kito Mann

JavaServer Faces is the new big thing in Java web development. It improves your power and reduces your workload 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

- A gentle introduction
- JSF under the hood
- Using JSF widgets
- How to:
  - integrate with Struts and existing apps
  - benefit from JSF tools from Oracle, IBM, and Sun
  - build custom components (lots of examples)
  - build renderers, converters, validators
  - put it all together in a JSF application

A developer for 16 years, **Kito D. Mann** is an enterprise architect who has consulted for several Fortune 500 companies. He runs the JSFCentral.com community site. Kito lives in Stamford, Connecticut with his wife, two parrots, and four cats.

“I 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

“Gets right into using JSF and explains the advanced topics in detail. Well-written and a quick read.”

—Matthew Schmidt  
Director, Advanced Technology  
JavaLobby

“A book written by a programmer who knows what programmers need.”

—Alex Kolundzija  
Front-End Developer  
Columbia House

“... an excellent job showing that JSF can be used with other technologies. A great reference and tutorial!”

—Mike Nash, JSF Expert Group Member  
Author, *Explorer’s Guide to Java Open Source Tools*

