CHAPTER 2

Elements of JSF

Now that we have taken a look at the JSF architecture in relation to other frameworks, it is time to dig into JSF itself. This chapter begins that process by giving an overview of essential JSF elements; in other words, those parts of the framework that are key to understanding how it works. We'll cover JSF UI components, Converters, Validators, Renderers, and much more. In doing so, we'll explore some of the sample applications provided with the JSF reference implementation. We'll continue with these applications in Chapter 3, "JSF Request-Processing Life Cycle," and Chapter 4, "JSF Configuration," as well while we finish covering the basics of JSF. In future chapters we'll present new sample applications as we cover more advanced topics. The hope here is that you become familiar with the JSF reference implementation while learning the basics of JSF.

The goal of this chapter is not a complete examination of each element, since that is the purpose of Chapters 5 to 10. Our goal here is simply to get a good feel for how JSF is composed and how those parts work together. At the end of this chapter, you will have a good, albeit basic, understanding of how the most important elements of JSF work.

Overview

JSF is primarily a user interface framework for developing Web-based applications on the Java platform. With the dizzying array of Java Web application frameworks already in existence, you may be wondering why we need another one. Here are a few of the more important motivations:

Standardization. JSF is an attempt to provide a standard user interface framework for the Web where no such standard currently exists. There are many benefits to having a multitude of frameworks to choose from, but such a choice does tend to fragment the Java development community. JSF provides a chance for Java developers to unite behind a standard, component-based user interface framework for the Web. JSF's ultimate inclusion under the J2EE umbrella will further this reality.

Tool Support. Having a standard Web application framework with a well-documented API makes the decision of which framework to support for all Java-based IDEs and other tools much easier. The design of JSF also lends itself well to graphical user interface building tools and will empower tool providers to create more complete and pleasurable development environments for developers at varying levels of experience.

Components. Of all the frameworks out there, very few of them incorporate the concept of reusable user interface components (beyond the concept of a Web page). JSF is just such an attempt, and it is intended to be for Web user interfaces what Swing is for more traditional user interfaces.

Web Development

Although JSF is designed to be independent of specific protocols and markup languages, most Java developers will use it in concert with Java servlets and JSPs to create HTML-based Web applications. These applications can communicate with Java application servers via the now ubiquitous and venerable HTTP. The authors of the JSF specification are aware of this demographic, so one of their primary goals is to resolve a number of issues related to using the stateless HTTP protocol in concert with HTML clients. The specification highlights a number of JSF features related to this goal:

UI Component State. JSF specifically addresses saving user interface component state between requests in a Web client session.

Component Rendering. HTML is just one of many markup languages, and each Web client's support of a particular markup language may vary. JSF provides a rendering mechanism for addressing this variety of target Web clients.

Form Processing. Most Web applications are form-based. JSF provides a number of convenient features for processing multipage and single-page form-based requests.

Form Validation. Along with form processing, validating form data is a critical need. JSF helps automate this process and provide the necessary error reporting.

Event Model. JSF provides a strongly typed component event model for responding to client-generated events with server-side handlers.

Type Conversion. Since Web client requests via HTTP provide form data as strings, a mechanism for converting these strings to and from the application model would be very useful. JSF provides a facility for enabling type conversion.

Error Handling. All applications must deal with application errors and exceptions. JSF provides a mechanism for handling error conditions and reporting them back to the user interface.

Internationalization. Multilanguage support is often a key requirement in Web applications, which are easily accessible from around the world. JSF provides native support of internationalization.

JSF also provides a standard tag library and rendering kits that support JSP development. We'll cover this JSP support in detail in Chapter 8, "JSP Integration in JSF."

UI Components

UI components are the centerpiece of JSF and are used as building blocks to create user interfaces that range from simple to complex. We covered the composable nature (via an implementation of the Composite Pattern) of these components in the previous chapter. Here, we will examine JSF UI components in more detail along with other component types that exist to support them. We'll delve into an even more comprehensive treatment of UI components in Chapter 5, "UI Components."

You are likely familiar with Swing user interface components as well as those from other languages or development environments. As Swing did for rich user interface clients, JSF provides a standard user interface component framework for the Web. This standardization promises more powerful and visual development environments for Web applications and libraries of rich user interface components like calendars, trees, and tables.

JSF also provides a number of other standard component types that play a supporting role to UI components. These include Converters, Validators,

Renderers, and others. Like UI components, these components are interchangeable and can be reused throughout the same application and a host of others. This opens up the possibility of additional libraries of components that cover these common supporting tasks.

So what makes a user interface component? Every user interface component in JSF implements the `javax.faces.component.UIComponent` interface. This comprehensive interface defines methods for navigating the component tree, interacting with backing data models, and managing supporting concerns such as component validation, data conversion, rendering, and a host of others. A convenient base class that implements this interface, `javax.faces.component.UIComponentBase`, is provided for creating new components. It provides default implementations for each method that component developers can then extend to customize the behavior of a component.

When creating new components in JSF, you essentially have three choices: create the component from scratch by directly implementing the `UIComponent` interface when no existing components meet your needs and you expect to override much of what `UIComponentBase` provides, subclass `UIComponentBase` to get default component behavior, or subclass an existing component to customize it for your needs. Subclassing an existing component will most likely be your choice for new individual components because most of the standard JSF components serve as good building blocks (input fields, labels, panels, and so on). Subclassing `UIComponentBase` will likely be your choice for building composite components. If you are creating new components for use with JSPs, you also need to either extend existing tag libraries or create your own so that your new component can be used in a JSP. We'll cover this process in great detail in Chapter 10, "Custom JSF Components."

The standard JSF components (and the standard JSF HTML RenderKit) provide a number of the basic components you'll need when building your Web applications. In most cases, you'll use one of these standard components or those from third-party libraries, so you typically won't need to worry about `UIComponent` or `UIComponentBase`. However, there will be times where you may wish to extend an existing component or have need of one that is not readily accessible. You may also wish to directly manipulate component trees on the server side (outside of JSPs) in a manner similar to that discussed in Chapter 1 with our coverage of the Composite Pattern.

Standard UI Components

Before looking at the UIComponent interface in more depth, let's briefly take a look at what the standard JSF components are. Each component is summarized in Table 2.1. A more detailed description of each component may be found in the *JavaServer Faces Specification*.

Table 2.1 Standard JSF UI Components

COMPONENT	DESCRIPTION
UICommand	UICommand represents UI components like buttons, hyperlinks, and menu items that result in application-specific actions or events. These components emit an ActionEvent when triggered that may be processed by a registered ActionListener.
UIForm	UIForm represents a user input form and is meant primarily to be a container of other components.
UIGraphic	UIGraphic displays an immutable image or picture.
UIInput	UIInput represents UI components like text input fields, numeric input fields, date input fields, memo fields, and so on. UIInput displays the current value of a field and accepts changes by the user. Each change triggers a ValueChangedEvent that may be processed with a registered ValueChangedListener. It is very common to see registered Validators and Converters attached to this type of component to ensure the validity of data entered by a user.
UIOutput	UIOutput represents UI components like labels, error messages, and any other textual data that is immutable.
UIPanel	UIPanel represents UI components that serve as containers for others without requiring form submissions. Examples of this type of component include a standard panel, tables, and lists. UIPanel will typically be used to manage the layout of its child components.
UIParameter	UIParameter represents information that requires no rendering. It is typically used to provide additional information to a parent component. Examples include declaring request parameters for the URL associated with a hyperlink (UICommand component) or input parameters necessary for displaying a registered message.
UISelectBoolean	UISelectBoolean represents a Boolean data field and is most often rendered as a check box. This component descends from UIInput, so it emits a ValueChangedEvent when a user checks or unchecks it.
UISelectItem	UISelectItem represents a single item in a selection list. It may be used to insert an item into a UISelectMany or a UISelectOne list.
UISelectItems	UISelectItems is very similar to UISelectItem with the exception that it allows the insertion of multiple items at once.
UISelectMany	UISelectMany represents UI components like combo boxes, list boxes, groups of check boxes, and so on. This component allows the selection of multiple items. Each item is specified by nesting one or more UISelectItem and UISelectItems components. This component descends from UIInput, so it emits a ValueChanged Event when a user modifies the list of selected items.
UISelectOne	UISelectOne is very similar to UISelectMany with the exception that it only allows the selection of one item.

A component hierarchy is provided in Figure 2.1. You'll notice how some components build upon and specialize others. This is not uncommon in user interface component libraries, and you can expect to see others extend this component hierarchy with a richer set of UI components.

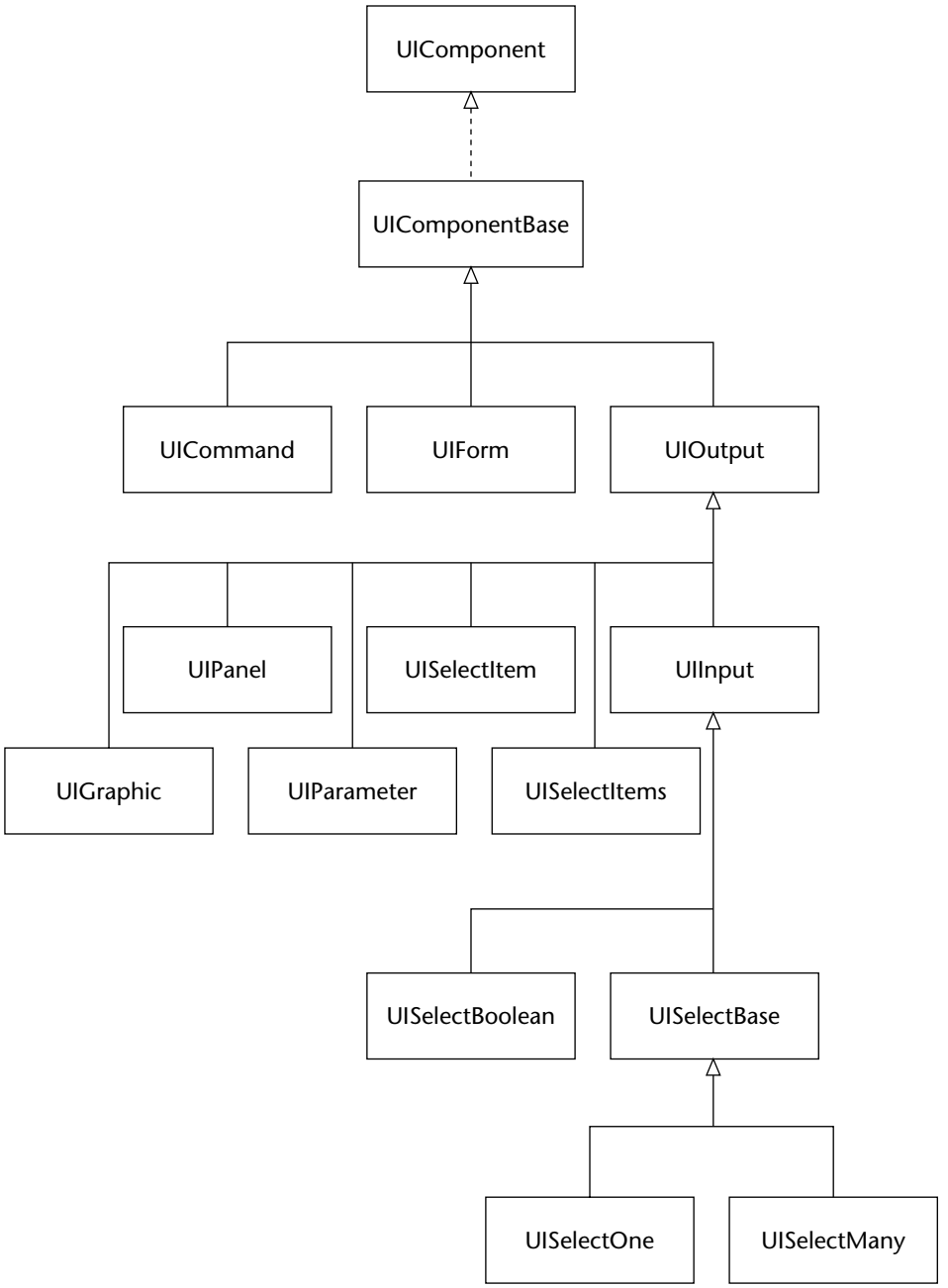

Figure 2.1 Standard JSF UI component hierarchy.

Each of the sample applications provided with the *JSF Reference Implementation* make use of some or all of these components. You will typically see them used via JSP tags that are defined as part of the default HTML RenderKit that every JSF implementation is required to provide. A summary list of these tags is provided in the "Rendering" section of this chapter.

You may have already noticed an important distinction with respect to JSF UI components. In JSF, function is separated from appearance. In other words, a component like UIInput actually represents a class of user interface components that you are used to dealing with. All of them are functionally similar, but they appear differently. For example, a simple text input field is functionally similar to a password entry field. They are represented by the same component type in JSF, but they have different Renderers (which produce a certain look and feel) associated with them. This is a simple but important concept to understand especially if you are considering developing your own UI components. Develop components according to function, then create Renderers to specify different look-and-feels as your user interface requires. The concept of delegation (which is actually a simple, yet fundamental software pattern) is used here to separate function from appearance. You'll see it again when we discuss Validators and Converters shortly.

Identifiers

Identifying components becomes important in complex user interfaces, and JSF addresses this with the concept of a *component identifier*. You set a component's identifier programmatically via its `setComponentId()` method, and you retrieve it via its `getComponentId()` method. However, most developers will set this identifier via a tag attribute in a JSP. Here is an example from the sample login application in Chapter 1.

```
<h:output_label for="userId">
    <h:output_text id="userIdLabel" value="User ID"/>
</h:output_label>
<h:input_text id="userId" valueRef="loginForm.userId"/>
```

Identifiers are optional, but in this example one is necessary for the label to associate itself with the input field. You should explicitly declare an identifier like this when other UI components or server-side components refer to it. Each identifier is required by the specification to meet the following requirements: should start with a letter; should contain only letters, whole numbers, dashes, and underscores; and should be as short as possible to limit the size of responses generated by JSF.

UI Component Trees

As we discussed in Chapter 1, user interface components in JSF are composable. In other words, you can nest one component within another to form more complex client interfaces. These compositions are referred to as *component trees* in JSF. A simple example of such a tree is the Guess Number sample application provided by the *JSF Reference Implementation*, is provided in Listing 2.1.

```
<html>
    <head> <title>Hello</title> </head>
    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
    <body bgcolor="white">
    <h2>
        Hi. My name is Duke. I'm thinking of a number from 0 to 10.
        Can you guess it?
    </h2>

    <f:use_faces>
    <h:form id="helloForm" formName="helloForm">
        <h:graphic_image id="wave_img" url="/wave.med.gif" />
        <h:input_number id="userNo" numberStyle="NUMBER"
                        valueRef="userNumberBean.userNumber">
            <f:validate_longrange minimum="0" maximum="10" />
        </h:input_number>

        <h:command_button id="submit" action="success" label="Submit"
                          commandName="submit" />
        <p>
        <h:output_errors id="errors1" for="userNo"/>
    </h:form>
    </f:use_faces>
</html>
```

Listing 2.1 JSP component tree.

You'll notice how the root component is a form component, and inside it we have an input component and a command button that submits the contents of the input component. You can of course imagine much more complicated interfaces that involve more input fields, labels, tables, and other components, but this at least provides a simple example of what a component tree looks like.

When the command button is pressed, a request that includes the content of the input field is submitted to the `FacesServlet`. As you'll see in Chapter 3, the servlet goes through a number of steps to process the request and return an appropriate response (in this case, another JSP). During that processing, a Java object version of the component tree is either created (for the initial request) or reconstituted. As a developer, you will work with this version of

the component tree in custom Validators, Converters, Renderers, and application components. Fortunately, JSF provides a convenient component API for manipulating these trees programmatically.

Tree Manipulation and Navigation

UI components have access to their parent and children via a number of methods defined by the `UIComponent` interface. These methods should be familiar to you from our discussion of the Composite Pattern in Chapter 1. Let's take another look at the Guess Number sample. A graphical representation of it is provided in Figure 2.2 in which some of those methods are reflected.

Most of these methods are self-explanatory. Invoking `getParent()` on either the input or submit component will return the parent form. Invoking `containsChild()` on the form will return `true`, while doing so on either of the child components will return `false`. You can use `getChildren()` on the form component to iterate over the list of direct children (in this case the input and submit components).

There are also methods for changing the list of child components in the form. Invoking `addChild()` with a new user interface component would add that component either to the end of the list of child components or at a position you specify. Likewise, invoking `removeChild()` either with a reference to the desired user interface component or its position in the list of children will remove it.

Since these methods are available for every user interface component in JSF, you are provided with a generic interface for navigating and manipulating components in arbitrarily complex tree structures. You just need to remember that the methods in this interface will behave differently based on whether a component actually has children or not. Methods like `getChildCount()` can help you determine what behavior to expect.

When dealing with a tree of components, you may wish to search for a particular component. You can do this by invoking the `findComponent()` method on a form, panel, or other container component. The expression argument will typically be the identifier of the component you are looking for. This particular method is convenient for avoiding lengthy traversals of more complex component trees.

Facets

The methods listed above allow you to manipulate and navigate composite components through a generic interface as described by the Composite Pattern. JSF provides an additional facility for defining the roles of subordinate components that may be independent of or orthogonal to the parent-child relationship. These roles are called *facets*.

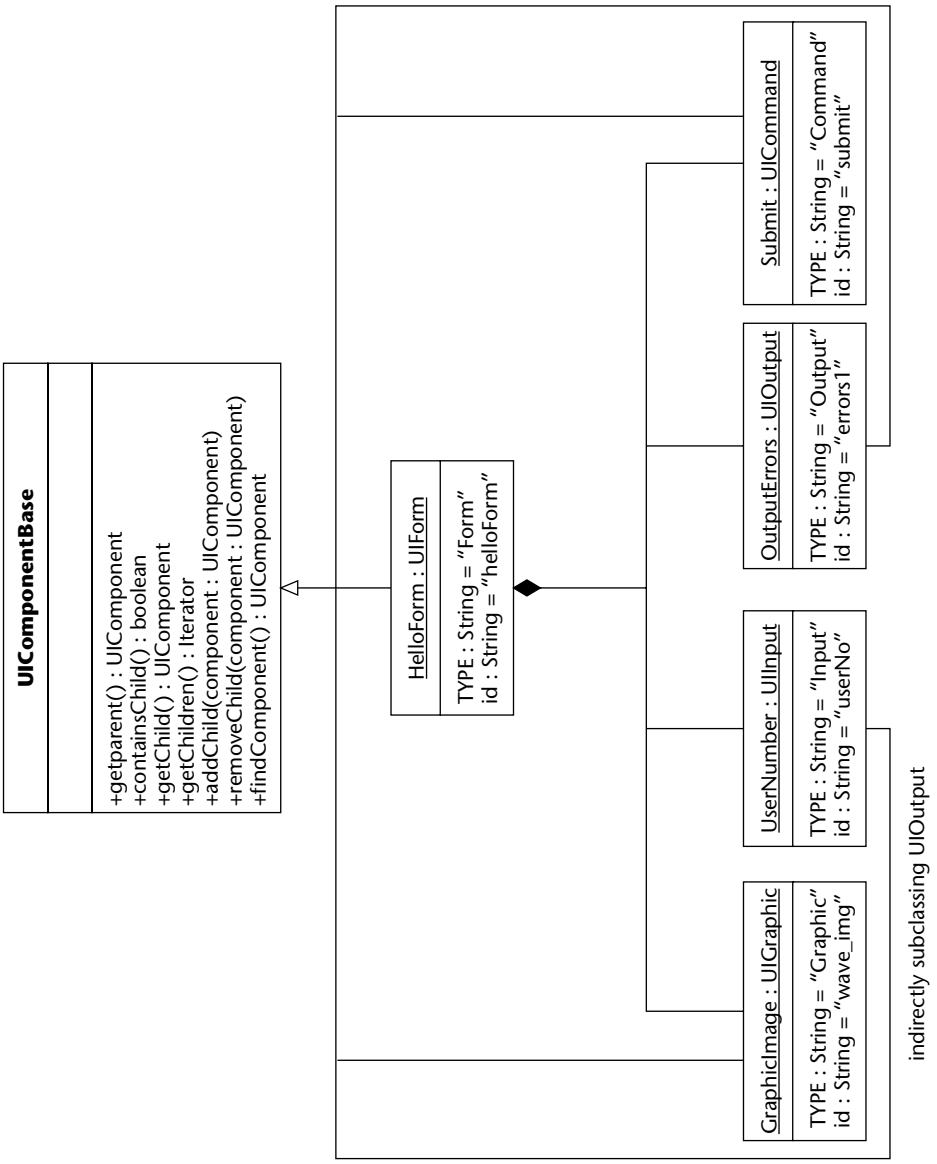

Figure 2.2 Component tree.

To understand how facets may be used, let's take a look at the Component sample application provided by the *JSF Reference Implementation*. An excerpt of the tabbed panes JSP is provided in Listing 2.2.

```
<h:form formName="tabbedForm" bundle="demoBundle">
<d:stylesheet path="/stylesheet.css"/>
Powered by Faces components:
<d:pane_tabbed id="tabcontrol"
        paneClass="tabbed-pane"
        contentClass="tabbed-content"
        selectedClass="tabbed-selected"
        unselectedClass="tabbed-unselected">

    <d:pane_tab id="first">
        <f:facet name="label">
            <d:pane_tablabel label="T a b 1" commandName="first" />
        </f:facet>
        ...
    </d:pane_tab>

    <d:pane_tab id="second" selected="true">
        <f:facet name="label">
            <d:pane_tablabel image="images/duke.gif"
commandName="second"/>
        </f:facet>
        ...
    </d:pane_tab>

    <d:pane_tab id="third">
        <f:facet name="label">
            <d:pane_tablabel label="T a b 3" commandName="third"/>
        </f:facet>
        ...
    </d:pane_tab>
</d:pane_tabbed>
</h:form>
```

Listing 2.2 Facets in tabbed pane JSP.

In this example, a facet called `label` is used to identify the label of each tab in the tabbed component. You can already see that the facet is being used here to logically identify a specific component that appears in each tab. To see how the facet is actually used, take a look at an excerpt from the `TabbedRenderer` class in Listing 2.3.

```
public void encodeEnd(FacesContext context, UIComponent component)
    throws IOException {

    // Ensure that exactly one of the child PaneComponents is selected
    Iterator kids = component.getChildren();

    ...

    // Render the labels for the tabs.
    String selectedClass =
        (String) component.getAttribute("selectedClass");
    String unselectedClass =
        (String) component.getAttribute("unselectedClass");
    ResponseWriter writer = context.getResponseWriter();

    ...

    kids = component.getChildren();
    while (kids.hasNext()) {
        UIComponent kid = (UIComponent) kids.next();
        if (!(kid instanceof PaneComponent)) {
            continue;
        }
        PaneComponent pane = (PaneComponent) kid;
        ...
        UIComponent facet = (UIComponent)pane.getFacet("label");
        if (facet != null) {
            if (pane.isSelected() && (selectedClass != null)) {
                facet.setAttribute("paneTabLabelClass", selectedClass);
            }
            else if (!pane.isSelected() && (unselectedClass != null)) {
                facet.setAttribute("paneTabLabelClass",
unselectedClass);
            }
            facet.encodeBegin(context);
        }
    ...
    }
    ...
}
```

Listing 2.3 Facets in tabbed pane Renderer.

This `Renderer` is used for showing the correct representation of a tabbed pane after the client has performed some action (such as changing the active tab). The `label` facet is used to easily locate the label component of each tab. Once located, the style of the label (or how it appears to the client) is set, based on whether or not the parent tab has been selected by the client. You can see the difference in the selected tab's label compared to those that are not selected in Figure 2.3.

If you select a different tab, not only does the content change, but the shading of the tabs themselves also changes. This is shown in Figure 2.4.

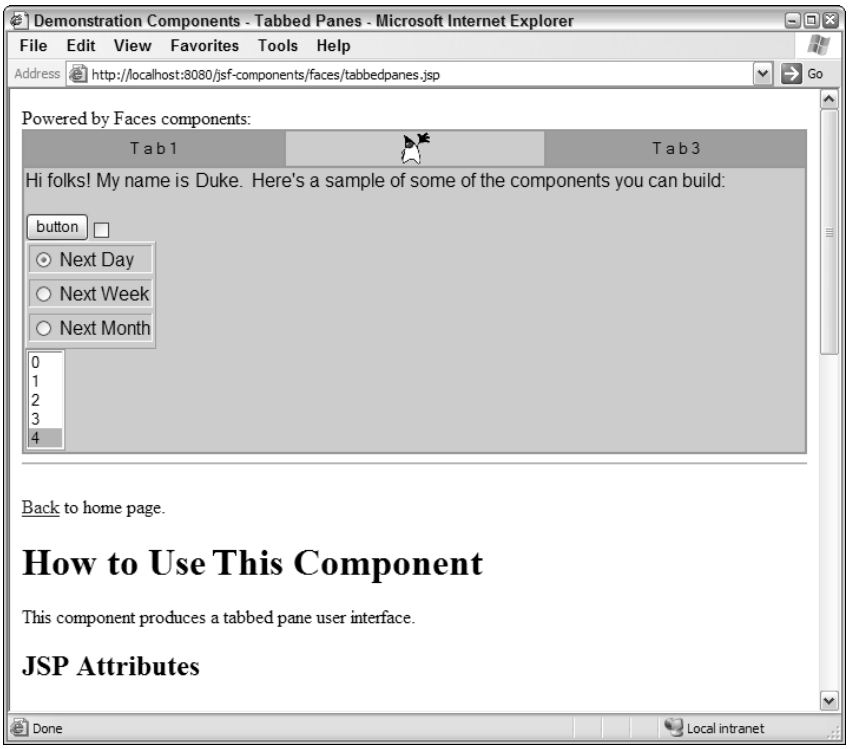

Figure 2.3 Initial tab selection.

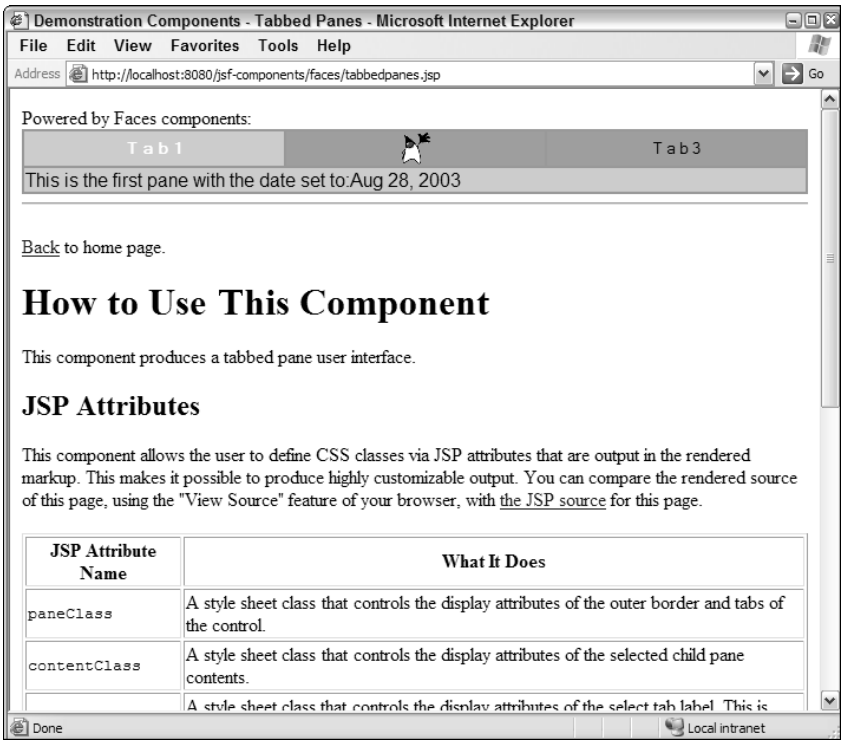

Figure 2.4 Different tab selection.

We'll talk more about Renderers later, but all you need to know for now is that facets can be used to easily classify and locate a component according to its function in your user interface. Instead of searching through a collection of components to find them, you simply retrieve them via their facet name or role.

It is worth noting that all components have methods for accessing and manipulating facets that have been associated with them. In our example, the `getFacet()` method was invoked along with the name of the facet we were looking for. There are other methods for adding and removing facets, as well as getting a list of facets. Only one component may be associated with a particular facet name at a time. Adding a facet that already exists has the effect of replacing the associated component.

Generic Attributes

JSF supports the concept of generic attributes, which allow for runtime component metadata that is separate from those properties defined for a component's implementation. Attributes are used frequently for Renderers, Validators, Converters, and events.

Every component has methods for accessing generic attributes. In our previous example, you've already seen the `getAttribute()` and `setAttribute()` methods in action for retrieving and changing the value of a specific attribute by name. The `getAttributeNames()` method may also be used to get a list of all attributes associated with a particular component. You may add attributes to a component at design time (in a JSP) or at run time (in a `Validator`, `Renderer`, and so on) to store pretty much anything.

When defining your own custom components, you may choose to represent properties of a component via the generic attribute mechanism or via standard JavaBean properties (with getters and setters on the component itself). You must weigh the type safety of actual JavaBean properties on the component itself versus the flexible but potentially unsafe use of generic attributes.

Render-Dependent Attributes

Each Renderer may have a set of attributes that is associated with a component. If you take another look at the tabbed pane sample we just covered, you'll notice a number of generic attributes set on the tabbed pane itself. The `contentClass`, `selectedClass`, and `unselectedClass` attributes are used by the associated `Renderer` to properly draw child tabs and their contents via style sheets. You'll also notice that an attribute called `selected` is used to mark the second tab (the one with our pal Duke) as the one that is currently selected. The code provided in Listing 2.3 (and the full method provided in the associated sample application) shows how the `Renderer` uses these attributes together to properly draw each tab in the tabbed pane.

Don't get too caught up in the specifics of this example, because we will cover the concept of component rendering in greater depth later in the book. For now, you should have a good idea of how attributes can be used with Renderers.

Configuration Parameters

Configuration parameters used with Converters, Validators, and event listeners may also be associated with components via generic attributes. In the Guess Number sample application we saw earlier in this chapter, you may remember the `Validator` that was associated with the input field.

```
<h:input_number id="userNo" numberStyle="NUMBER"
                valueRef="userNumberBean.userNumber">
    <f:validate_longrange minimum="0" maximum="10" />
</h:input_number>
```

A `Validator` is assigned to an input component that expects a number. The `Validator` has two attributes called `minimum` and `maximum` that when combined define the acceptable bounds of the user-provided number. In this case, the number must be between 1 and 10. When the `Validator` actually validates the value provided by the input component, it will use these two attributes to define the bounds.

This is just a simple example of using attributes to handle configuration parameters. We'll cover this topic in more detail later in the book.

Data Models

When working with JSF applications, you will usually deal with two different data models. The first model is associated with user interface components to manage their state. The second model is associated with the server-side application-level model. User interface component models are often populated with snapshots of data from the application model, while changes to component models are propagated back to the application model via application events.

Component Model

A component model object, which is often a JavaBean object, is usually associated with a component via JSF's managed bean facility. If you remember our discussion of managed beans in Chapter 1 while exploring the MVC pattern as applied to JSF, we used a managed bean to hold data from the login form JSP. The `LoginForm` JavaBean was registered in the associated JSF configuration file as a managed bean and made available to UI components through a unique identifier. To bind a UI component to this bean, we used the `valueRef` attribute.

```
<h:input_secret id="password" valueRef="loginForm.password"/>
```

This attribute references the managed bean through its `loginForm` identifier and then binds to its `password` property. We'll cover this topic in more detail in Chapter 4 and then again in Chapters 5 to 11.

Application Model

Application models represent business data. As such, there are a number of alternatives for representing and persisting such data:

- JavaBeans (where persistence is not required)
- JavaBeans with JDBC
- JavaBeans that use an object-relation framework
- Enterprise JavaBeans

You will often interface with these application model objects through managed beans and their associated `Action` objects. Our discussion of the MVC Pattern in Chapter 1 provided a good example of how a `LoginAction` interfaces with business objects. You'll see more examples of this in Chapters 5 to 11.

Validation

Performing correctness checks on user submitted data is an important aspect of all applications that collect information (not just Web applications), even if that information is only temporarily used to perform a calculation of some sort. Being able to catch bad data as it is provided is critical to ensuring the integrity of the information you collect and for your application to perform as expected. It is no surprise, then, that data validation is an integral part of the JSF framework.

In this section, we'll cover `Validator` components in JSF and how they are associated with UI components. We'll also look at how you register Validators with your applications, and before using your first `Validator` component, it will be good to know what standard `Validator` components JSF provides out of the box.

Validators

Validation in JSF comes in two forms: direct validation within a user interface component and delegated validation, which occurs outside of a user interface component. Figure 2.5 provides a graphical summary of this choice. Which method you choose is essentially determined by whether the validation you are performing is specific to a user interface component or should be reused among a number of different user interface components. It will not be uncommon for you to use both methods for components that have a nontrivial amount of data behind them, some of which may be unique to that component, while the rest may be common data types seen elsewhere.

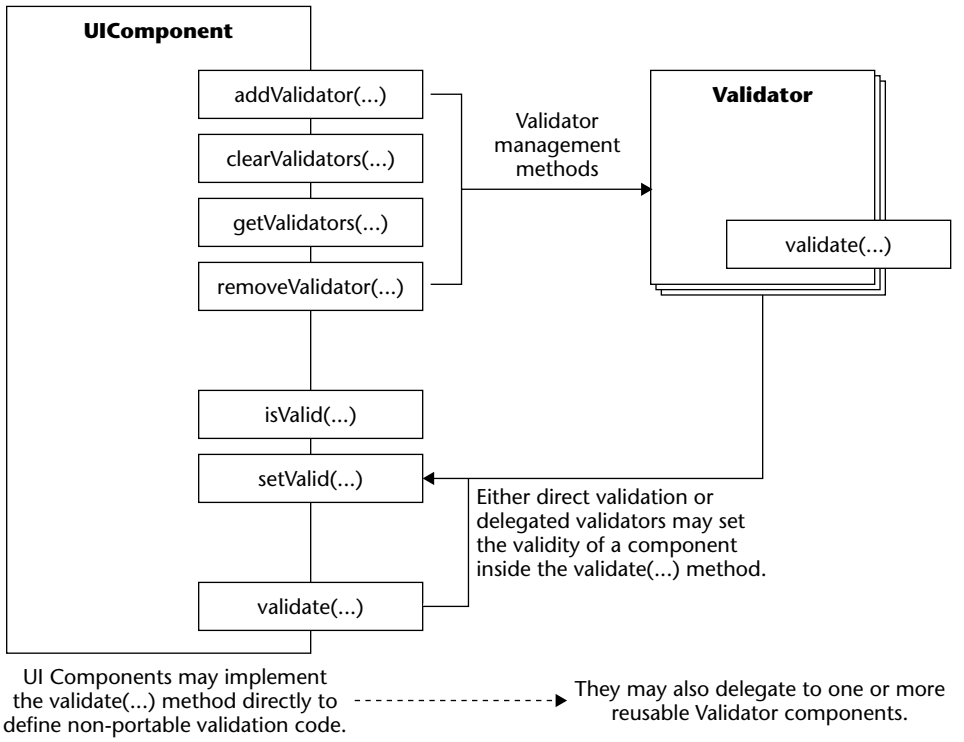

Figure 2.5 Direct validation versus delegated validation.

Direct Validation

If your validation is specific to a particular user interface component, then you may implement your validation code within the component by overriding the `validate()` method. At the appropriate point in the request processing life cycle, the JSF framework will invoke this method, thereby executing any code you have provided for correctness checking. This method of validation provides a quick and efficient means of validating your components; the downside is that your validation code is nonportable. An additional restriction is that you can only use this method of validation for your own components or you must subclass an existing component to implement the `validate()` method.

Delegated Validation

If your validation should be reused among different user interface component types or if you would like to attach a certain type of validation to an existing component, the JSF framework provides a mechanism for defining reusable Validator components that you can associate with a user interface component. Each Validator component must implement the `javax.faces.validator.Validator` interface. This interface contains the following method, which is very similar to what we covered for direct validation:

```
public void validate(FacesContext context, UIComponent component);
```

You implement your validation code within this method and operate on the generic component instance that is passed in.

We previously touched on the possibility of using generic attributes with Validators for the purpose of storing configuration information. For general-purpose Validators, this is certainly a flexible option. You may also define these configuration items via JavaBean properties and/or constructor arguments on the Validator component itself. Either way, you have the flexibility of defining configurable Validators when necessary.

Let's take a look at a simple, but complete Validator example. The JSP code in Listing 2.4 is an excerpt from the Car Demo sample application provided by the *JSF Reference Implementation*.

```
...
<h:form  formName="CustomerForm" >
    ...
    <h:input_text id="ccno" size="16" converter="creditcard" >
        <f:validate_required/>
        <cd:format_validator formatPatterns="9999999999999999|
                                             9999 9999 9999 9999|
                                             9999-9999-9999-9999"/>
    </h:input_text>
    ...
</h:form>
```

Listing 2.4 Assigning Validators to a UI component in a JSP.

You'll notice that two different Validators are being assigned to an input component that is used to collect a credit card number. Each `Validator` will later be called upon in the order in which it was defined to perform correctness checking. The first `Validator` ensures that something has been entered, while the second `Validator` ensures that the string is formatted properly according to the provided input masks. Let's now take a look at the source code for the second `Validator` in Listing 2.5.

```
public class FormatValidator implements Validator {

    public static final String FORMAT_INVALID_MESSAGE_ID =
        "cardemo.Format_Invalid";

    public FormatValidator() {
        super();
    }

    /**
     * <p>Construct a FormatValidator with the specified formatPatterns
     * String.</p>
     *
     * @param formatPatterns <code>|</code> separated String of format
     *                       patterns that this validator must match
     *                        against.
     */
    public FormatValidator(String formatPatterns) {
        super();
        this.formatPatterns = formatPatterns;
        parseFormatPatterns();
    }

    private String formatPatterns = null;

    public String getFormatPatterns() {
        return (this.formatPatterns);
    }

    public void setFormatPatterns(String formatPatterns) {
        this.formatPatterns = formatPatterns;
        parseFormatPatterns();
    }

    /**
     * Method from Validator interface
     */
    public void validate(FacesContext context, UIComponent component) {
        boolean valid = false;

        if ((context == null) || (component == null)) {
            throw new NullPointerException();
        }
        if (!(component instanceof UIOutput)) {
            return;
        }

        if ( formatPatternsList == null ) {
            // No patterns to match
```

Listing 2.5 Format Validator.

```
            component.setValid(true);
            return;
        }

        String value = (((UIOutput)component).getValue()).toString();
        // validate the value against the list of valid patterns.
        Iterator patternIt = formatPatternsList.iterator();
        while (patternIt.hasNext()) {
            valid = isFormatValid(((String)patternIt.next()), value);
            if (valid) {
                break;
            }
        }
        if ( valid ) {
            component.setValid(true);
        } else {
            component.setValid(false);
            Message errMsg = getMessageResources().getMessage(context,
                    FORMAT_INVALID_MESSAGE_ID,
                    (new Object[] {formatPatterns}));
             context.addMessage(component, errMsg);
        }
    }

    /**
     * Parses the <code>formatPatterns</code> into validPatterns
     * <code>ArrayList</code>. The delimiter must be "|".
     */
    public void parseFormatPatterns() {
        ...
    }

    /**
     * Returns true if the value matches one of the valid patterns
     */
    protected boolean isFormatValid(String pattern, String value) {
        ...
    }

    /**
     * This method will be called before calling
     * facesContext.addMessage, so message can be localized.
     * <p>Return the {@link MessageResources} instance for the message
     * resources defined by the JavaServer Faces Specification.
     */
    public synchronized MessageResources getMessageResources() {
        ...
    }
}
```

Listing 2.5 *(continued)*

One of the first things you'll notice here is that a constructor parameter is being used to set the acceptable format patterns, which, as we just mentioned, is an alternative to generic attributes. You'll also notice how the `setValid()` method is used inside the implemented `validate()` method to flag the target component as valid or invalid. In Chapter 3, we'll take a look at what happens when components are flagged as invalid.

Validator Registration

When using any of the standard JSF Validators, you simply nest the appropriate tag within the tag that represents a UI component. We saw an example of this being done with the Required `Validator` in Listing 2.4. Although you may assign Validators in JSPs via the standard JSF tag library, they may also be assigned and managed dynamically at run time via the component tree. Every user interface component has a number of methods for adding and removing Validators, as well as getting a list of all active Validators in the order they are assigned (as shown in Figure 2.5).

You may also create and assign your own `Validator` components. When doing so, you must make sure you register each custom `Validator` in your application's associated JSF configuration file (see Chapter 4 for a discussion of how to do this). You may then either create a JSP tag for your JSP or use the standard `<validator>` tag and specify the `Validator` by its type. When using this generic tag, the input field in Listing 2.4 could be rewritten as follows.

```
<h:input_text id="ccno" size="16" converter="creditcard" >
    <f:validate_required/>
    <f:validator type="cardemo.FormatValidator"/>
    <f:attribute name="formatPatterns"
                 value="9999999999999999|
                        9999 9999 9999 9999|
                        9999-9999-9999-9999"/>
</h:input_text>
```

You'll see that the `Validator` is identified by its unique type (which is provided as part of the registration process in the associated JSF configuration file), and its `formatPatterns` attribute is added to and accessible from the text input UI component. The advantage of creating an associated custom JSP tag is that it provides a more user-friendly notation for page authors, especially when the `Validator` requires the use of one or more attributes. For a more complete discussion of Validator components, see Chapter 7, "Validation and Conversion."

Standard Validators

The *JSF Specification* requires every JSF implementation to provide some basic but useful Validators, like the first `Validator` in the previous example. A summary of each standard `Validator` is provided in Table 2.2.

The associated JSP tags are available via the core JSF tag library, which is declared as follows in your JSP page.

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

You can expect a number of custom `Validator` components for all sorts of correctness checking from third parties. The JSF framework makes this very easy to do. In Chapter 7, we'll cover the process of creating your own custom Validators, which are similar to the format `Validator` we just examined.

Table 2.2 Standard JSF Validators

VALIDATOR	JSP TAG	DESCRIPTION
DoubleRangeValidator	validate_doublerange	Validates that an input field provides a String that may be converted to a double and that it is within the supplied maximum and minimum values.
LengthValidator	validate_length	Validates that an input field provides a String (or a value that may be converted to a String) and that it's length is within the supplied maximum and minimum values.
LongRangeValidator	validate_longrange	Validates that an input field provides a String that may be converted to a long and that it is within the supplied maximum and minimum values.
RequiredValidator	validate_required	Validates that an input field has a value that is not null. If the value is set to a String, the length of that String must be one character or more.
StringRangeValidator	validate_stringrange	Validates that an input field provides a String (or something that may be converted to a String) and that it is within the supplied maximum and minimum values.

Conversion

Web applications capture user data as request parameters over the HTTP protocol. Each of these parameters is in the form of a String, while the backing data for the application on the server side is in the form of Java objects. An example of these two views of the same data is a date. To a Web client, the date may be shown as a String in any number of formats from numbers (03/05/2003) to natural language (March 5, 2003). It may also have some sort of calendar control that allows users to graphically select a date. On the server side, a date is probably represented with a `java.util.Date` object.

The challenge for you as a developer is to convert this data back and forth between both representations. This can be mind-numbing, error-prone work, but JSF fortunately provides support for reusable data conversion via Converters.

Converters

Every user interface component may have an optional Converter associated with it. Each Converter must implement the `javax.faces.convert.Converter` interface. A custom Converter from the Car Demo sample application is provided in Listing 2.6.

```
public class CreditCardConverter implements Converter {

    /**
     * Parses the CreditCardNumber and strips any blanks or
     * <code>"-"</code> characters from it.
     */
    public Object getAsObject(FacesContext context,
        UIComponent component, String newValue)
        throws ConverterException {

        String convertedValue = null;
        if ( newValue == null ) {
            return newValue;
        }
        // Since this is only a String to String conversion, this
        // conversion does not throw ConverterException.
        convertedValue = newValue.trim();
        if ( ((convertedValue.indexOf("-")) != -1) ||
            ((convertedValue.indexOf(" ")) != -1)) {
            char[] input = convertedValue.toCharArray();
            StringBuffer buffer = new StringBuffer(50);
            for ( int i = 0; i < input.length; ++i ) {
                if ( input[i] == '-' || input[i] == ' '  ) {
```

Listing 2.6 Credit card custom Converter.

```
                    continue;
                } else {
                    buffer.append(input[i]);
                }
            }
            convertedValue = buffer.toString();
        }
        return convertedValue;
    }

    /**
     * Formats the value by inserting a space after every four
* characters for better readability if one doesn't already
* exist. In the process converts any <oode>"-"</code>
* characters into blanks for consistency.
     */
    public String getAsString(FacesContext context,
        UIComponent component, Object value) throws ConverterException {

        String inputVal = null;
        if ( value == null ) {
            return null;
        }
        // Value must be of the type that can be cast to a String
        try {
            inputVal = (String)value;
        } catch (ClassCastException ce) {
            throw new ConverterException(Util.getExceptionMessage(
                        Util.CONVERSION_ERROR_MESSAGE_ID));
        }

        // Insert spaces after every four characters for better
        // readability if it doesn't already exist.
        char[] input = inputVal.toCharArray();
        StringBuffer buffer = new StringBuffer(50);
        for ( int i = 0; i < input.length; ++i ) {
            if ( (i % 4) == 0 && i != 0) {
                if (input[i] != ' ' || input[i] != '-'){
                    buffer.append(" ");
                  // If there any "-"'s convert them to blanks.
                } else if (input[i] == '-') {
                    buffer.append(" ");
                }
            }
            buffer.append(input[i]);
        }
        String convertedValue = buffer.toString();
        return convertedValue;
    }
}
```

Listing 2.6 *(continued)*

This Converter implements the `getAsObject()` and `getAsString()` methods as required by the `Converter` interface. Converters are expected to be symmetric in their processing of data. In other words, a `Converter` should typically implement both the `getAsObject` and `getAsString` methods such that a piece of data will be equivalent when it is converted back and forth via these methods.

Converter instances are typically shared among components via a factory mechanism. Because of this, it is important that they be programmed in a thread-safe manner. In some cases, a `Converter` may require configuration values to operate correctly. The example we provided earlier involved the expected format of a date, which may be stored as a generic attribute on an associated user interface component.

Every user interface component has methods for getting and setting a Converter. These methods are `getConverter()` and `setConverter()`, respectively, and they may be used to dynamically assign a `Converter` at any time. Converters are associated via a String-based *converter identifier*, which we will cover in the following "Converter Registration" section.

Although you may associate a Converter with a user interface component by invoking its `setConverter()` method, you will typically set a Converter via an associated JSP tag attribute. If you take another look at Listing 2.4, you'll see the custom credit card Converter being assigned to the credit card number input component via the `converter` attribute. This attribute is provided as part of the standard JSF HTML tag library. Behind the scenes, the `<input_text>` tag just invokes the `setConverter()` method for you with the converter identifier provided.

Converter Registration

When using your own custom Converter components, you must first register them with your application in its associated JSF configuration file. This process is covered in Chapter 4 for the credit card Converter we have been using as an example here. As part of the registration process, you provide a unique identifier for the Converter, and this identifier is what you use to associate a Converter with a UI component via its optional `converter` attribute.

Standard Converters

The *JSF Specification* requires every JSF implementation to provide a basic set of Converters. A summary of each standard `Converter` is provided in Table 2.3.

Table 2.3 Standard JSF Converters

CONVERTER	ATTRIBUTES	DESCRIPTION
Date	dateStyle timezone	Converts a String or Number to a Date with attributes for controlling the date style and time zone considerations
DateFormat	formatPattern timezone	Converts a String or Number to a Date with attributes for controlling the format of the date and time zone considerations
DateTime	dateStyle timeStyle timezone	Converts a String or Number to a Date with attributes for controlling the date style, time style, and time zone considerations
Number	numberStyle	Converts a String or Number to a Number with an attribute for controlling the number style (currency, percent, integer, and so on)
NumberFormat	formatPattern	Converts a String or Number to a Number with an attribute for controlling the number format

These Converters basically give you a few different ways to convert input fields into dates and numbers with various styles and formats. As with standard Validators, you'll need to declare the core JSF tag library in the JSP page you wish to use the Converters. An example usage of one of these standard Converters is provided below.

```
<h:input_text id="total" valueRef="Invoice.total" converter="Number">
    <f:attribute name="numberStyle" value="currency"/>
</h:input_text>
```

This input field represents the total amount of an invoice, and the associated Converter will transfer the `String` entered by users into a `Number` that represents money.

As with Validators, you can expect a number of custom Converter components for all sorts of `String` and `Number` conversions (among others) from third parties. The JSF framework makes this very easy to do. In Chapter 7, we'll cover the process of creating your own custom Converters, which are similar to the credit card Converter we just examined.

Events and Listeners

Events provide an important mechanism for user interface components to propagate user actions to other components (including server-side components). These interested components register as listeners to events they are interested in (the pressing of a button or perhaps a change to the value in a text input field). Events were introduced in Chapter 1 in terms of the Observer Pattern, so the concept of events should be very familiar to you. We also discovered that JSF takes an approach to events and event handling that is similar to Swing's. Both of them are based on the *JavaBean Specification* event model. We'll review JSF events here in a bit more detail.

UI Events

Each user interface component may emit any number of events and have any number of listeners registered to have these events broadcast to them. These listeners may be other user interface components or application components. Each supported event must extend the `javax.faces.event.FacesEvent` base class.

The constructor of each event accepts a reference to the user interface component that is responsible for propagating it, while the `getComponent()` method provides access to the originating component. Events may also have useful information (usually involving component state) that they wish to communicate to registered listeners. For this reason, an Event may provide any number of properties and constructors to initialize them. The JavaBean specification recommends that the name of each event class end with the word *Event*.

The JSF specification defines two standard user interface component events. The `javax.faces.event.ActionEvent` is broadcast from the standard `UICommand` component (typically rendered as a push button, a menu item, or a hyperlink) when activated by a user, whereas the `javax.faces.event.ValueChangedEvent` is broadcast from the standard `UIInput` component or subclasses when its value has changed and passed validation.

Listeners

Each event type has a corresponding listener interface that must extend the `javax.faces.event.FacesListener` interface. Application components or other user interface components may implement any number of listener interfaces, as long as they provide the appropriate event handler method for each one. Listener implementations have access to the corresponding event via a parameter that is passed in to the event handler method. The JavaBean specification recommends that each listener interface name be based on the event class it is associated with and end with *Listener*.

The JSF specification defines two standard listener interfaces corresponding to the two standard event types. Listeners of the `ActionEvent` must implement `javax.faces.event.ActionListener` and receive notification via invocation of the `processAction()` method, whereas listeners of the `ValueChangedEvent` must implement `javax.faces.event.ValueChangedListener` and receive notification via invocation of the `processValueChanged()` method they are required to implement.

An example of a listener is provided in Listing 2.7 from the Car Demo sample. This particular listener implementation responds to a user's clicking an image map by forwarding control to the appropriate JSP page.

```
public class ImageMapEventHandler implements ActionListener {

    Hashtable localeTable = new Hashtable();

    public ImageMapEventHandler ( ) {
        localeTable.put("NAmericas", Locale.ENGLISH);
        localeTable.put("SAmericas", new Locale("es","es"));
        localeTable.put("Germany", Locale.GERMAN);
        localeTable.put("France", Locale.FRENCH);
    }

    public PhaseId getPhaseId() {
        return PhaseId.ANY_PHASE;
    }

    // Processes the event queued on the specified component
    public void processAction(ActionEvent event) {

        UIMap map = (UIMap)event.getSource();
        String value = (String) map.getAttribute("currentArea");
        Locale curLocale = (Locale) localeTable.get(value);
        if ( curLocale != null) {
            FacesContext context = FacesContext.getCurrentInstance();
            context.setLocale(curLocale);
            String treeId = "/Storefront.jsp";
            TreeFactory treeFactory = (TreeFactory)
            FactoryFinder.getFactory(FactoryFinder.TREE_FACTORY);
            Assert.assert_it(null != treeFactory);
            context.setTree(treeFactory.getTree(context,treeId));
        }
    }
}
```

Listing 2.7 Custom event listener.

Now that we've covered what is required of event listeners, you may be wondering how we register them to receive event notifications. The JavaBean specification requires a component that emits a particular event to define a pair of methods for registering and unregistering listeners. We'll use the standard `ActionListener` as an example of what these methods should look like:

```
public void addActionListener(ActionListener listener);
public void removeActionListener(ActionListener listener);
```

Any component that wishes to register for a certain event, must simply call the appropriate `add()` method at any time on the user interface component it wishes to observe. Likewise, if a component no longer wishes to receive event notifications, it must simply call the appropriate `remove()` method on the component it is observing. However, you will typically register listeners in JSPs, as shown for our image map example in Listing 2.8.

```
...
<h:form  formName="imageMapForm" >
    ...
    <h:graphic_image id="mapImage" url="/world.jpg" usemap="#worldMap"
/>
    <d:map id="worldMap" currentArea="NAmericas" >
        <f:action_listener type="cardemo.ImageMapEventHandler"/>
        <d:area id="NAmericas" modelReference="NA"
                 onmouseover="/cardemo/world_namer.jpg"
                 onmouseout="/cardemo/world.jpg" />
        <d:area id="SAmericas" modelReference="SA"
                onmouseover="/cardemo/world_samer.jpg"
                onmouseout="/cardemo/world.jpg" />
        <d:area id="Germany" modelReference="gerA"
                onmouseover="/cardemo/world_germany.jpg"
                onmouseout="/cardemo/world.jpg" />
        <d:area id="France" modelReference="fraA"
                onmouseover="/cardemo/world_france.jpg"
                onmouseout="/cardemo/world.jpg" />
    </d:map>
</h:form>
...
```

Listing 2.8 Custom event listener assigned in JSP.

This registration is done via the standard JSF HTML tag library with the `<action_listener>` tag, as shown below.

```
<f:action_listener type="cardemo.ImageMapEventHandler"/>
```

The appropriate listener class is associated via the `type` attribute. Behind the scenes, the `<action_listener>` tag calls the `addActionListener()` method for you on the image map component.

Phase Identifiers

All listener implementations must implement the `getPhaseId()` method from the `FacesEvent` interface to specify at which stage of the request-processing life cycle they wish to receive event notifications. This method returns an instance of `javax.faces.event.PhaseId`, which is an enumerated type that defines each stage at the end of which events may be broadcast to registered listeners. An additional type of `PhaseId.ANY_PHASE` is provided for those listeners who wish to be notified every time a particular event is broadcast. The event listener in Listing 2.8 implements this method.

We'll cover the request-processing life cycle in greater detail in Chapter 3, including the overall process for handling events.

Event Queuing and Broadcasting

During the request-processing life cycle, events may be created in response to user actions, and all of them are queued in the `FacesContext` in the order in which they are received. At the end of each phase where events may be handled, any events in the queue are broadcast to registered listeners that define the appropriate `PhaseId`. As we discussed earlier, this action results in the appropriate event-processing method being invoked on each registered listener.

Application Events

JSF also provides `Action` objects that may respond to `ActionEvents` emitted by `UICommand` components. We covered the use of `Action` objects in our discussion of the MVC Pattern in Chapter 1. You attach an `Action` object to a UICommand component by setting its optional `actionRef` attribute. As a review of what we covered, here is the relevant piece of code in the login JSP.

```
<h:command_button commandName="login" actionRef="loginForm.login"/>
```

The `loginForm` identifier refers to a managed bean that holds the input form data and that has been registered in the application's JSF configuration file. It is not uncommon to place an `Action` inside of the component's associated data bean so that it has direct access to the input form data. Actions are used in JSF much as they are in Struts—to interface with business objects. We'll touch upon `Action` objects again in greater detail in Chapter 6, "Navigation, Actions, and Listeners," and again in Chapter 9, "Building JSF Applications."

For now, just remember that JSF `Action` objects are what you will typically use to interact with your business object layer in response to user interface commands.

Rendering

One of the most important aspects of user interfaces is how they look and feel to users. JSF provides a flexible mechanism for rendering responses in Web applications, and it comes in two forms: direct rendering within a user interface component and delegated rendering via RenderKits that occur outside of a user interface component. Figure 2.6 provides a graphical summary of this choice. As with Validators, the method you choose is dependent upon how specific a rendering is to a particular user interface component.

Direct Rendering

With direct rendering, a user interface component must encode and decode itself by overriding one or more of the rendering methods defined by `UIComponentBase`.

The `decode()` method is invoked on a component after a request is received and is expected to convert request parameters into a user interface component with its current state. This conversion process is aided by a Converter if one has been assigned to the component. The set of encode methods, `encodeBegin()`, `encodeChildren()`, and `encodeEnd()` are invoked when the JSF implementation is preparing a response to a client request. If the component you are rendering has no children, then you only need to implement the `encodeEnd()` method. As with the `decode()` method, a Converter may be used if assigned. When performing direct rendering, you must also override the `getRendersSelf()` method to return `true`.

Direct rendering coupled with the direct validation we discussed earlier allows component authors to build self-contained custom components in single classes. If used correctly, this option can be compact and efficient. On the other hand, it does limit reuse of common rendering and validation among multiple components, which could have a negative impact on maintainability.

Delegated Rendering

Delegating user interface component encoding and decoding to external components allows you to quickly change the look and feel of components and to render appropriate responses to different client types. In essence, the rendering of a user interface component is separated out and becomes pluggable with other possible rendering.

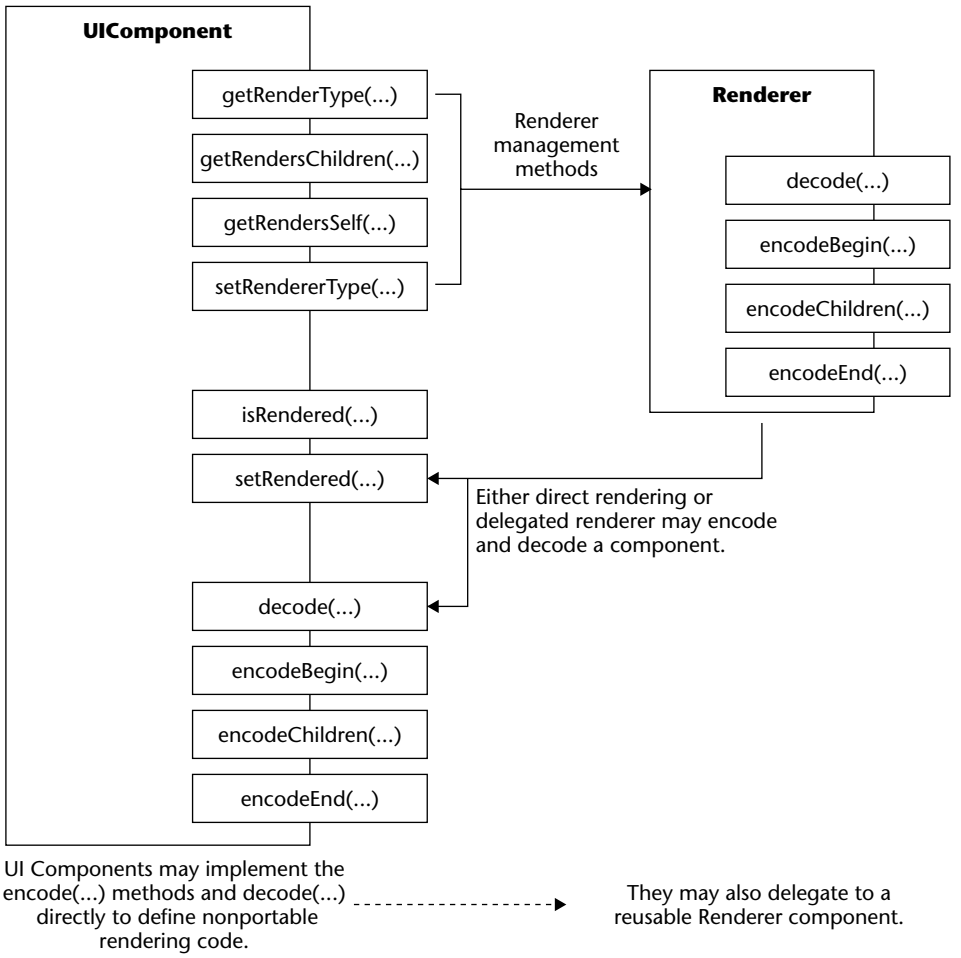

Figure 2.6 Direct rendering versus delegated rendering.

A `Renderer` is a subclass of the abstract class `javax.faces.render.Renderer` and provides the same encode and decode methods that exist on a user interface component for direct rendering. We provided an example of a custom `Renderer` in Listing 2.3. We only showed the `encodeEnd()` method, but if you look at the complete example provided with the JSF reference implementation, you'll see the other methods as well.

You are probably wondering at this point how to associate a `Renderer` with a UI component. This is done by implementing the `getRendererType()` method and returning the appropriate render type (see the "Registering Renderers" section for more information on render types). At run time, your chosen JSF implementation will call this method when encoding and decoding the UI component to determine which `Renderer`, if any, should be used. This

property would return a value of `null` if you choose direct over delegated rendering. When using standard JSF UI components, you don't have to worry about setting a render type, because each of these components already defaults to one in the standard HTML RenderKit. You only need to worry about setting a render type when you are creating a new component or customizing the look of an existing one. A good example of handling component rendering for a custom UI component is provided in Chapter 10.

Each `Renderer` may also recognize certain generic attributes that are used to properly encode and decode an associated user interface component. You'll recall our earlier example of a tabbed pane that uses the `selected` attribute to determine which tab is selected. This tab is then rendered with a different appearance than the others.

Render Kits

A RenderKit is a subclass of the abstract class `javax.faces.render.RenderKit` and represents a collection of Renderers that typically specialize in rendering user interface components in an application based on some combination of client device type, markup language, and/or user locale. Render kits are conceptually similar to Swing looks and feels in that they are pluggable and often render user interfaces based on a common theme.

At some point you may wish to customize the Renderers of an existing RenderKit or even create your own RenderKit. You will typically create Renderers for your custom components and register them with existing RenderKits. We'll take a look at that registration process next, and we'll explore custom user interface component rendering in much greater detail in Chapter 10.

Registering Renderers

Before using the tabbed `Renderer` we have seen more than once so far in this chapter, it must be registered in the associated application's JSF configuration file.

```
<render-kit>
  ...
  <renderer>
    <renderer-type>Tabbed</renderer-type>
    <renderer-class>components.renderkit.TabbedRenderer</renderer-class>
  </renderer>
  ...
</render-kit>
```

The configuration information here registers the `Renderer` with the default HTML RenderKit (which is the default behavior of not specifying a particular

RenderKit). As we discussed earlier in this chapter, your JSF implementation will check each UI component's `getRendererType()` method to see if it should delegate rendering to a `Renderer`; otherwise, it will expect the UI component to render itself. We'll explore Renderers more in Chapter 10.

Standard RenderKits

The JSF specification defines a standard RenderKit and set of associated Renderers for generating HTML compatible markup. Each JSF implementation is required to support this RenderKit. A summary of the available Renderers is provided in Table 2.4.

Table 2.4 Standard JSF HTML Renderers

RENDERER / TAGS	UICOMPONENTS	DESCRIPTION
Button / <command_button>	UICommand	Represents your typical command button
Hyperlink / <command_hyperlink>	UICommand	Represents a Web link
Image / <graphic_image>	UIGraphic	Represents an icon or image.
Date / <input_date> <output_date>	UIInput / UIOutput	Represents a date String in an input field or a label
DateTime / <input_datetime> <output_datetime>	UIInput / UIOutput	Represents a date/time String in an input field or a label
Number / <input_number> <output_number>	UIInput / UIOutput	Represents a Number in an input field or a label
Text / <input_text> <output_text>	UIInput / UIOutput	Represents plain text in an input field or a label
Time / <input_time> <output_time>	UIInput / UIOutput	Represents a time String in an input field or a label
Hidden / <input_hidden>	UIInput	Represents an invisible field that is useful for a page author
Secret / <input_secret>	UIInput	Represents a password input field or one in which the characters are masked

(continued)

Table 2.4 *(continued)*

RENDERER / TAGS	UICOMPONENTS	DESCRIPTION
TextArea / <input_textarea>	UIInput	Represents a multiline text input or memo field
Errors / <output_errors>	UIOutput	Displays error messages for input fields (or an entire page as well)
Label / <output_label>	UIOutput	Represents a label for an input field
Message / <output_message>	UIOutput	Displays a localized message
Data / <panel_data>	UIPanel	Represents a set of rows from a collection of data
Grid / <panel_grid>	UIPanel	Represents a table
Group / <panel_group>	UIPanel	Visually represents a group of related components
List / <panel_list>	UIPanel	Represents a list
Checkbox / <selectboolean_ checkbox>	UISelectBoolean	Represents a check box
CheckboxList /	UISeletMany	Represents a list of check boxes
Listbox / <selectmany_listbox> <selectone_listbox>	UISelectMany / UISelectOne	Represents a list of items from which one or more may be selected
Menu / <selectmany_menu> <selectone_menu>	UISelectMany / UISelectOne	Represents a menu
Radio / <selectone_radio>	UISelectOne	Represents a set of radio buttons from which one choice may be made

The determination of which `Renderer` you use will be automatically handled based on the tag you use in your JSPs. You have already seen some of these tags in action in our examples, and you'll see more of them throughout the rest of the book. You will no doubt also see more Renderers (and associated tags) from third-party vendors and likely the vendor of your particular JSF implementation.

Summary

Now that we've covered the most important elements of JSF and how many of them work together in JSF applications, it's time to pull everything together with a discussion of the JSF request-processing life cycle. We'll do this in Chapter 3 before rolling up our sleeves and digging into more details.