Standard JSF Tags

Topics in This Chapter

Chapter 4

Development of compelling JSF applications requires a good grasp of the JSF tag libraries—core and HTML—that represent a combined total of 43 tags. Because of their prominence in the JSF framework, this chapter and the next—Data Tables—provide in-depth coverage of those tags, their attributes, and how you can best use them.

Even simple JSF pages use tags from both libraries. Many JSF pages have a structure similar to this:

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

<f:view>
    <h:form>
        ...
    </h:form>
</f:view>
```

To use the JSF tag libraries, you must import them with `taglib` directives, as in the preceding code fragment. You can choose any name you want for the prefixes. The convention is `f` and `h`, for the core and HTML libraries, respectively.

This chapter starts with a brief look at the core library. That library, with 18 tags, is smaller than its HTML sibling, which has 25. It's also considerably simpler than the HTML library. Because of that simplicity and because most of the core tags are discussed elsewhere in this book, the overwhelming majority of this chapter focuses on the HTML library.

We begin our exploration of the HTML library with a look at common attributes shared by most JSF HTML tags. Then we discuss each tag individually with attribute tables for reference and useful code examples that you can adapt to your own applications.

NOTE: The core library has 2.8 attributes per tag—the HTML library has 26.2.

An Overview of the JSF Core Tags

The core library is the poor stepchild of the HTML library—the former exists entirely to support the latter. The core tags are listed in Table 4–1.

Table 4–1 JSF Core Tags

Tag	Description
`view`	Creates the top-level view
`subview`	Creates a subview of a view
`facet`	Adds a facet to a component
`attribute`	Adds an attribute (key/value) to a component
`param`	Adds a parameter to a component
`actionListener`	Adds an action listener to a component
`valueChangeListener`	Adds a valuechange listener to a component
`convertDateTime`	Adds a datetime converter to a component
`convertNumber`	Adds a number converter to a component
`validator`	Adds a validator to a component
`validateDoubleRange`	Validates a `double` range for a component's value
`validateLength`	Validates the length of a component's value
`validateLongRange`	Validates a `long` range for a component's value
`loadBundle`	Loads a resource bundle, stores properties as a Map
`selectitems`	Specifies items for a select one or select many component
`selectitem`	Specifies an item for a select one or select many component
`verbatim`	Adds markup to a JSF page

Most of the core tags represent *objects you add to components*:

- Attributes
- Listeners
- Converters
- Validators
- Facets
- Parameters
- Select items

The core library also contains tags for defining views and subviews, loading resource bundles, and adding arbitrary text to a page.

All of the core tags are discussed at length elsewhere in this book.

NOTE: All tag attributes, except for `var` and `id`, accept value reference expressions. The majority of those expressions represent value bindings; a handful represent method bindings.

An Overview of the JSF HTML Tags

JSF HTML tags represent the following kinds of components:

- Inputs
- Outputs
- Commands
- Selection
- Others

The Others category includes forms, messages, and components that layout other components. Table 4–2 lists all the HTML tags.

Table 4–2 JSF HTML Tags

Tag	Description
`form`	HTML form
`inputText`	Single-line text input control
`inputTextarea`	Multiline text input control
`inputSecret`	Password input control

Table 4–2 JSF HTML Tags (cont.)

Tag	Description
`inputHidden`	Hidden field
`outputLabel`	Label for another component for accessibility
`outputLink`	HTML anchor
`outputFormat`	Like `outputText`, but formats compound messages
`outputText`	Single-line text output
`commandButton`	Button: submit, reset, or pushbutton
`commandLink`	Link that acts like a pushbutton
`message`	Displays the most recent message for a component
`messages`	Displays all messages
`graphicImage`	Displays an image
`selectOneListbox`	Single-select listbox
`selectOneMenu`	Single-select menu
`selectOneRadio`	Set of radio buttons
`selectBooleanCheckbox`	Checkbox
`selectManyCheckbox`	Set of checkboxes
`selectManyListbox`	Multiselect listbox
`selectManyMenu`	Multiselect menu
`panelGrid`	HTML table
`panelGroup`	Two or more components that are laid out as one
`dataTable`	A feature-rich table control
`column`	Column in a `dataTable`

We can group the HTML tags in the following categories:

- Inputs (`input...`)
- Outputs (`output...`)
- Commands (`commandButton` and `commandLink`)
- Selections (`checkbox`, `listbox`, `menu`, `radio`)
- Layouts (`panelGrid`)
- Data Table (`dataTable`); see Chapter 5
- Errors and messages (`message`, `messages`)

The JSF HTML tags share common attributes, HTML pass-through attributes, and attributes that support dynamic HTML.

NOTE: The HTML tags may seem overly verbose; for example, `selectManyListbox` could be more efficiently expressed as `multiList`. But those verbose names correspond to a component/renderer combination, so `selectManyListbox` represents a `selectMany` component paired with a `listbox` renderer. Knowing the type of component a tag represents is crucial if you want to access components programmatically.

NOTE: Both JSF and Struts developers implement web pages with JSP custom tags. But Struts tags generate HTML directly, whereas JSF tags represent a component and renderer that generate HTML. That key difference makes it easy to adapt JSF applications to alternative display technologies, as we'll see when we implement a wireless JSF application in Chapter 11.

Common Attributes

Three types of tag attributes are shared among multiple HTML component tags:

- Basic
- HTML 4.0
- DHTML events

Let's look at each type.

Basic Attributes

As you can see from Table 4–3, basic attributes are shared by the majority of JSF HTML tags.

Table 4–3 Basic HTML Tag Attributes[a]

Attribute	Component Types	Description
`id`	A (*25*)	Identifier for a component
`binding`	A (*25*)	Reference to the component that can be used in a backing bean
`rendered`	A (*25*)	A boolean; `false` suppresses rendering
`styleClass`	A (*23*)	Cascading stylesheet (CSS) class name
`value`	I, O, C (*19*)	A component's value, typically a value binding
`valueChangeListener`	I (*11*)	A method binding to a method that responds to value changes
`converter`	I,O (*15*)	Converter class name
`validator`	I (*11*)	Class name of a validator that's created and attached to a component
`required`	I (*11*)	A boolean; if `true`, requires a value to be entered in the associated field

a. A = all, I = input, O = output, C = commands, (*n*) = number of tags with attribute

The `id` and `binding` attributes, applicable to all HTML tags, reference a component—the former is used primarily by page authors, and the latter is used by Java developers.

The `value` and `converter` attributes let you specify a component value and a means to convert it from a string to an object, or vice versa.

The `validator`, `required`, and `valueChangeListener` attributes are available for input components so that you can validate values and react to changes to those values. See Chapter 6 for more information about validators and converters.

The ubiquitous `rendered` and `styleClass` attributes affect how a component is rendered.

Let's take a brief look at these important attributes.

IDs and Bindings

The versatile `id` attribute lets you do the following:

- Access JSF components from other JSF tags
- Obtain component references in Java code
- Access HTML elements with scripts

In this section we discuss the first two tasks listed above. See "Form Elements and JavaScript" on page 97 for more about the last task.

The `id` attribute lets page authors reference a component from another tag. For example, an error message for a component can be displayed like this:

```
<h:inputText id="name".../>
<h:message for="name"/>
```

You can also use component identifiers to get a component reference in your Java code. For example, you could access the `name` component in a listener, like this:

```
UIComponent component = event.getComponent().findComponent("name");
```

The preceding call to `findComponent` has a caveat: the component that generated the event and the `name` component must be in the same form (or data table). There is a better way to access a component in your Java code. Define the component as an instance field of a class. Provide property getters and setters for the component. Then use the `binding` attribute, which you specify in a JSF page like this:

```
<h:outputText binding="#{form.statePrompt}".../>
```

The `binding` attribute is specified with a value reference expression. That expression refers to a read-write bean property. See Chapter 2 for more information about the `binding` attribute. The JSF implementation sets the property to the component, so you can programatically manipulate components.

You can also *programmatically create a component that will be used in lieu of the component specified in the JSF page*. For example, the form bean's `statePrompt` property could be implemented like this:

```
    private UIComponent statePrompt = new UIOutput();
    public UIComponent getStatePrompt() { return statePrompt; }
    public void setStatePrompt(UIComponent statePrompt) {...}
```

When the `#{form.statePrompt}` value binding is first encountered, the JSF framework calls `Form.getStateOutput()`. If that method returns `null`—as is typically the case—the JSF implementation creates the component specified in the JSF page.

But *if that method returns a reference to a component*—as is the case in the preceding code fragment—*that component is used instead*.

Values, Converters, and Validators

Inputs, outputs, commands, and data tables all have values. Associated tags in the HTML library, such as `h:inputText` and `h:dataTable`, come with a `value` attribute. You can specify values with a string, like this:

```
<h:outputText value="William"/>
```

Most of the time you'll use a value binding, for example:

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

The `converter` attribute, shared by inputs and outputs, lets you attach a converter to a component. Input tags also have a `validator` attribute that you can use to attach a validator to a component. Converters and validators are discussed at length in Chapter 6.

Rendering and Styles

You can use CSS styles, either inline (`style`) or classes (`styleClass`), to influence how components are rendered. Most of the time you'll specify string constants instead of value bindings for the `style` and `styleClass` attributes; for example:

```
<h:outputText value="#{customer.name}" styleClass="emphasis"/>
<h:outputText value="#{customer.id}" style="border: thin solid blue"/>
```

Value bindings are useful when you need programmatic control over styles. You can also control whether components are rendered at all with the `rendered` attribute. That attribute comes in handy in all sorts of situations, for example, an optional table column.

TIP: Instead of using a hardwired style, it's better to use a stylesheet. Define a CSS style such as

```
.prompts {
    color:red;
}
```

Place it in a stylesheet, say, `styles.css`. Add a `link` element inside the head element in your JSF page:

```
<link href="styles.css" rel="stylesheet" type="text/css"/>
```

Then use the `styleClass` attribute:

```
<h:outputText value="#{msgs.namePrompt}" styleClass="prompts"/>
```

Now you can change the appearance of all prompts simply by updating the stylesheet.

TIP: Remember, you can use operators in value reference expressions. For example, you might have a view that acts as a tabbed pane by optionally rendering a panel depending on the selected tab. In that case, you could use `h:panelGrid` like this:

```
<h:panelGrid rendered='#{bean.selectedTab == "Movies"}'/>
```

The preceding code renders the movies panel when the `Movies` tab is selected.

HTML 4.0 Attributes

JSF HTML tags have appropriate HTML 4.0 pass-through attributes. Those attribute values are passed through to the generated HTML element. For example, `<h:inputText value="#{form.name.last}" size="25".../>` generates this HTML: `<input type="text" size="25".../>`. Notice that the `size` attribute is passed through to HTML.

The HTML 4.0 attributes are listed in Table 4–4.

Table 4–4 HTML 4.0 Pass-through Attributes[a]

Attribute	Description
`accesskey (14)`	A key, typically combined with a system-defined metakey, that gives focus to an element
`accept (1)`	Comma-separated list of content types for a form
`accept-charset (1)`	Comma- or space-separated list of character encodings for a form. The `accept-charset` attribute is specified with the JSF HTML attribute named `acceptcharset`.
`alt (4)`	Alternative text for nontextual elements such as images or applets
`border (4)`	Pixel value for an element's `border` width
`charset (3)`	Character encoding for a linked resource
`coords (2)`	Coordinates for an element whose shape is a rectangle, circle, or polygon
`dir (18)`	Direction for text. Valid values are `ltr` (left to right) and `rtl` (right to left).
`disabled (11)`	Disabled state of an input element or button
`hreflang (2)`	Base language of a resource specified with the `href` attribute; `hreflang` may only be used with `href`.

Table 4–4 HTML 4.0 Pass-through Attributes[a] (cont.)

Attribute	Description
`lang (20)`	Base language of an element's attributes and text
`maxlength (2)`	Maximum number of characters for text fields
`readonly (11)`	Read-only state of an input field; text can be selected in a read-only field but not edited
`rel (2)`	Relationship between the current document and a link specified with the `href` attribute
`rev (2)`	Reverse link from the anchor specified with `href` to the current document. The value of the attribute is a space-separated list of link types.
`rows (1)`	Number of visible rows in a text area. `h:dataTable` has a rows attribute, but it's not an HTML pass-through attribute.
`shape (2)`	Shape of a region. Valid values: `default`, `rect`, `circle`, `poly`. (`default` signifies the entire region)
`size (4)`	Size of an input field
`style (23)`	Inline style information
`tabindex (14)`	Numerical value specifying a tab index
`target (3)`	The name of a frame in which a document is opened
`title (22)`	A title, used for accessibility, that describes an element. Visual browsers typically create tooltips for the title's value
`type (4)`	Type of a link; for example, "stylesheet"
`width (3)`	Width of an element

a. (*n*) = number of tags with attribute

The attributes listed in Table 4–4 are defined in the HTML specification, which you can access on line at http://www.w3.org/TR/REC-html40. That web site is an excellent resource for deep digging into HTML.

DHTML Events

Client-side scripting is useful for all sorts of tasks, such as syntax validation or rollover images, and it is easy to use with JSF. HTML attributes that support scripting, such as `onclick` and `onchange` are referred to as dynamic HTML

(DHTML) event attributes. JSF supports DHTML event attributes for nearly all of the JSF HTML tags. Those attributes are listed in Table 4–5.

Table 4–5 DHTML Event Attributes[a]

Attribute	Description
`onblur (14)`	Element loses focus
`onchange (11)`	Element's value changes
`onclick (17)`	Mouse button is clicked over the element
`ondblclick (18)`	Mouse button is double-clicked over the element
`onfocus (14)`	Element receives focus
`onkeydown (18)`	Key is pressed
`onkeypress (18)`	Key is pressed and subsequently released
`onkeyup (18)`	Key is released
`onmousedown (18)`	Mouse button is pressed over the element
`onmousemove (18)`	Mouse moves over the element
`onmouseout (18)`	Mouse leaves the element's area
`onmouseover (18)`	Mouse moves onto an element
`onmouseup (18)`	Mouse button is released
onreset (1)	Form is reset
`onselect (11)`	Text is selected in an input field
onsubmit (1)	Form is submitted

a. (*n*) = number of tags with attribute

The DHTML event attributes listed in Table 4–5 let you associate client-side scripts with events. Typically, JavaScript is used as a scripting language, but you can use any scripting language you like. See the HTML specification for more details.

TIP: You'll probably add client-side scripts to your JSF pages soon after you start using JSF. One common use is to submit a request when an input's value is changed so that value change listeners are immediately notified of the change, like this: `<h:selectOneMenu onchange="submit()"...>`

Forms

Web applications run on form submissions, and JSF applications are no exception. Table 4–6 lists all h:form attributes.

Table 4–6 Attributes for h:form

Attributes	Description
binding, id, rendered, styleClass	Basic attributes[a]
accept, acceptcharset, dir, enctype, lang, style, target, title	HTML 4.0[b] attributes
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onreset, onsubmit	DHTML events[c]

a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.
c. See Table 4–5 on page 95 for information about DHTML event attributes.

Although the HTML form tag has method and action attributes, h:form does not. Because you can save state in the client—an option that is implemented as a hidden field—posting forms with the GET method is disallowed. The contents of that hidden field can be quite large and may overrun the buffer for request parameters, so all JSF form submissions are implemented with the POST method. Also, if you don't specify navigation, JSF form submissions post to the current page (through the Faces servlet), although the actual page that's loaded as a result of a form submit can (and typically does) change as a result of actions that are fired on behalf of command components. See Chapter 2 for more details about actions.

The h:form tag generates an HTML form element. For example, if, in a JSF page named /index.jsp, you use an h:form tag with no attributes, the Form renderer generates HTML like this:

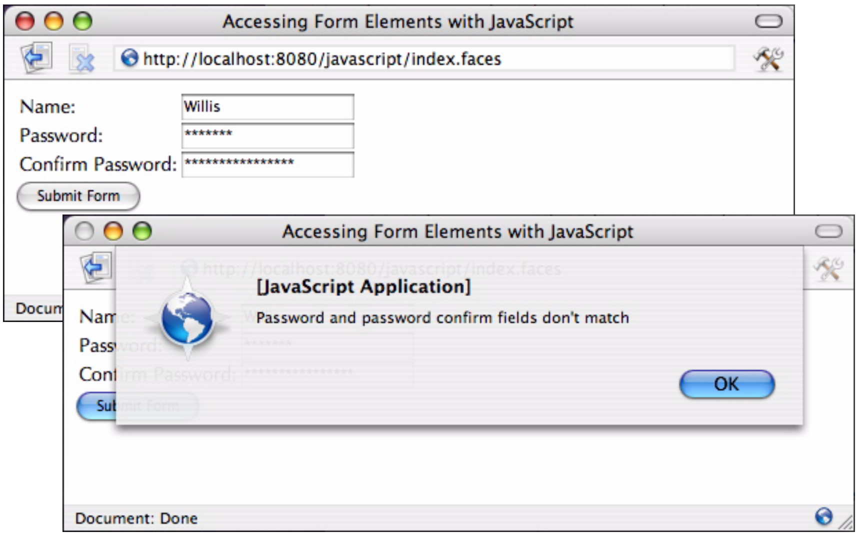
```
<form id="_id0" method="post" action="/forms/faces/index.jsp"
            enctype="application/x-www-form-urlencoded">
```

h:form comes with a full complement of DHTML event attributes. You can also specify the style or styleClass attributes for h:form. Those styles will then be applied to all output elements contained in the form.

Finally, the id attribute is passed through to the HTML form element. If you don't specify the id attribute explicitly, a value is generated by the JSF implementation, as is the case for all generated HTML elements. The id attribute is often explicitly specified for forms so that it can be referenced from stylesheets or scripts.

Form Elements and JavaScript

Java*Server* Faces is all about *server*-side components, but it's also designed to work with scripting languages, such as JavaScript. For example, the application shown in Figure 4–1 uses JavaScript to confirm that a password field matches a password confirm field. If the fields don't match, a JavaScript dialog is displayed. If they do match, the form is submitted.


Accessing Form Elements with JavaScript
http://localhost:8080/javascript/index.faces
Name:
Password:
Confirm Password:
Submit Form
Document: Done
OK

[JavaScript Application]

Password and password confirm fields don't match

Figure 4–1 Using JavaScript to Access Form Elements

We use the `id` attribute to assign names to the relevant HTML elements so that we can access them with JavaScript:

```
<h:form id="registerForm">
    ...
    <h:inputText id="password".../>
    <h:inputText id="passwordConfirm".../>
    ...
    <h:commandButton type="button"
                     onclick="checkPassword(this.form)"/>
    ...
</h:form>
```

When the button is clicked, a JavaScript function—`checkPassword`—is invoked. That function follows:

```
function checkPassword(form) {
    var password = form["registerForm:password"].value;
    var passwordConfirm = form["registerForm:passwordConfirm"].value;

    if (password == passwordConfirm)
        form.submit();
    else
        alert("Password and password confirm fields don't match");
}
```

Notice the syntax used to access form elements. You might think you could access the form elements with a simpler syntax, like this:

```
documents.forms.registerForm.password
```

But that won't work. Let's look at the HTML produced by the preceding code to find out why:

```
<form id="registerForm" method="post"
     action="/javascript/faces/index.jsp"
    enctype="application/x-www-form-urlencoded">
    ...
    <input id="registerForm:password"
         type="text" name="registerForm:password"/>
    ...
    <input type="button" name="registerForm:_id5"
        value="Submit Form" onclick="checkPassword(this.form)"/>
    ...
</form>
```

All form controls generated by JSF have names that conform to *formName:componentName*, where *formName* represents the name of the control's form and *componentName* represents the control's name. If you don't specify `id` attributes, the JSF framework creates identifiers for you, as you can see from the button in the preceding HTML fragment. Therefore, to access the password field in the preceding example, you must do this instead:

```
documents.forms.registerForm["registerForm:password"].value
```

The directory structure for the application shown in Figure 4–1 is shown in Figure 4–2. The JSF page is listed in Listing 4–1 and the English resource bundle is listed in Listing 4–2.

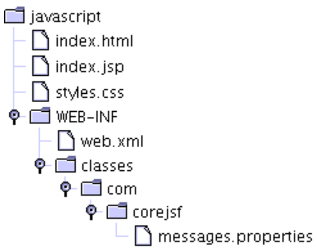


Figure 4–2 The JavaScript Example Directory Structure

Listing 4–1 javascript/index.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
      <head>
         <title>
            <h:outputText value="#{msgs.windowTitle}"/>
         </title>
      </head>
      <body>
         <h:form id="registerForm">
            <table>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.namePrompt}"/>
                  </td>
                  <td>
                     <h:inputText/>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.passwordPrompt}"/>
                  </td>
                  <td>
                     <h:inputSecret id="password"/>
                  </td>
               </tr>
               <tr>
```

Listing 4–1 javascript/index.jsp (cont.)

```
               <td>
                  <h:outputText value="#{msgs.confirmPasswordPrompt}"/>
               </td>
               <td>
                  <h:inputSecret id="passwordConfirm"/>
               </td>
            </tr>
         </table>
         <h:commandButton type="button" value="Submit Form"
            onclick="checkPassword(this.form)"/>
      </h:form>
   </body>
   <script type="text/javascript">
   <!--
      function checkPassword(form) {
         var password = form["registerForm:password"].value;
         var passwordConfirm = form["registerForm:passwordConfirm"].value;

         if(password == passwordConfirm)
            form.submit();
         else
            alert("Password and password confirm fields don't match");
      }
   -->
   </script>
</f:view>
</html>
```

Listing 4–2 valuechange/WEB-INF/classes/com/corejsf/messages.properties

```
windowTitle=Accessing Form Elements with JavaScript
namePrompt=Name:
passwordPrompt=Password:
confirmPasswordPrompt=Confirm Password:
```

Text Fields and Text Areas

Text inputs are the mainstay of most web applications. JSF supports three varieties represented by the following tags:

- `h:inputText`
- `h:inputSecret`
- `h:inputTextarea`

Since the three tags use similar attributes, Table 4–7 lists attributes for all three.

Table 4–7 Attributes for `h:inputText`, `h:inputSecret` and `h:inputTextarea`

Attributes	Description
`cols`	For `h:inputTextarea` only—number of columns
`immediate`	Process validation early in the life cycle
`redisplay`	For `h:inputSecret` only—when true, the input field's value is redisplayed when the web page is reloaded
`required`	Require input in the component when the form is submitted
`rows`	For `h:inputTextarea` only—number of rows
`valueChangeListener`	A specified listener that's notified of value changes
`binding, converter, id, rendered, required, styleClass, value, validator`	Basic attributes[a]
`accesskey, alt, dir, disabled, lang, maxlength, readonly, size, style, tabindex, title`	HTML 4.0 pass-through attributes[b]—`alt`, `maxlength`, and `size` do not apply to `h:inputTextarea`
`onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onselect`	DHTML events[c]

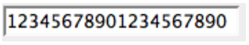
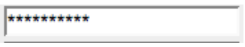
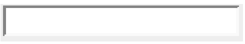
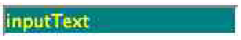
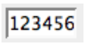
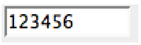
a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.
c. See Table 4–5 on page 95 for information about DHTML event attributes.

All three tags have `immediate`, `required`, `value`, and `valueChangeListener` attributes. The `immediate` attribute is used primarily for value changes that affect the user interface and is rarely used by these three tags. Instead, it is more commonly used by other input components such as menus and listboxes. See Chapter 7 for more information about the `immediate` attribute.

Three attributes in Table 4–7 are each applicable to only one tag: `cols`, `rows`, and `redisplay`. The `rows` and `cols` attributes are used with `h:inputTextarea` to specify the number of rows and columns, respectively, for the text area. The `redisplay` attribute, used with `h:inputSecret`, is a `boolean` that determines whether a secret field retains its value—and therefore redisplays it—when the field's form is resubmitted.

Table 4–8 shows sample uses of the `h:inputText` and `h:inputSecret` tags.

Table 4–8 `h:inputText` **and** `h:inputSecret` **Examples**

Example	Result
`<h:inputText value="#{form.testString}" readonly="true"/>`	 12345678901234567890
`<h:inputSecret value="#{form.passwd}" redisplay="true"/>`	 ********** (shown after an unsuccessful form submit)
`<h:inputSecret value="#{form.passwd}" redisplay="false"/>`	 (shown after an unsuccessful form submit)
`<h:inputText value="inputText" style="color: Yellow; background: Teal;"/>`	 inputText
`<h:inputText value="1234567" size="5"/>`	 123456
`<h:inputText value="1234567890" maxlength="6" size="10"/>`	 123456

The first example in Table 4–8 produces the following HTML:

```
<input type="text" name="_id0:_id4" value="12345678901234567890"
    readonly="readonly"/>
```

The input field is read-only, so our form bean only defines a getter method:

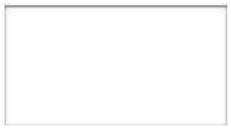
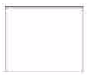
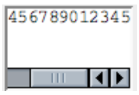
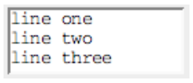
```
private String testString = "12345678901234567890";
public String getTestString() {
    return testString;
}
```

The `h:inputSecret` examples illustrate the use of the `redisplay` attribute. If that attribute is `true`, the text field stores its value between requests and therefore the value is redisplayed when the page reloads. If `redisplay` is `false`, the value is discarded and is not redisplayed.

The size attribute specifies the number of visible characters in a text field. But because most fonts are variable width, the size attribute is not precise, as you can see from the fifth example in Table 4–8, which specifies a size of 5 but displays six characters. The maxlength attribute specifies the maximum number of characters a text field will display. That attribute is precise. Both size and maxlength are HTML pass-through attributes.

Table 4–9 shows examples of the h:inputTextarea tag.

Table 4–9 h:inputTextarea **Examples**

Example	Result
`<h:inputTextarea rows="5"/>`	
`<h:inputTextarea cols="5"/>`	
`<h:inputTextarea value="123456789012345"` `rows="3" cols="10"/>`	 456789012345
`<h:inputTextarea value="#{form.dataInRows}"` `rows="2" cols="15"/>`	 line one line two line three

The h:inputTextarea has cols and rows attributes to specify the number of columns and rows, respectively, in the text area. The cols attributes is analogous to the size attribute for h:inputText and is also imprecise.

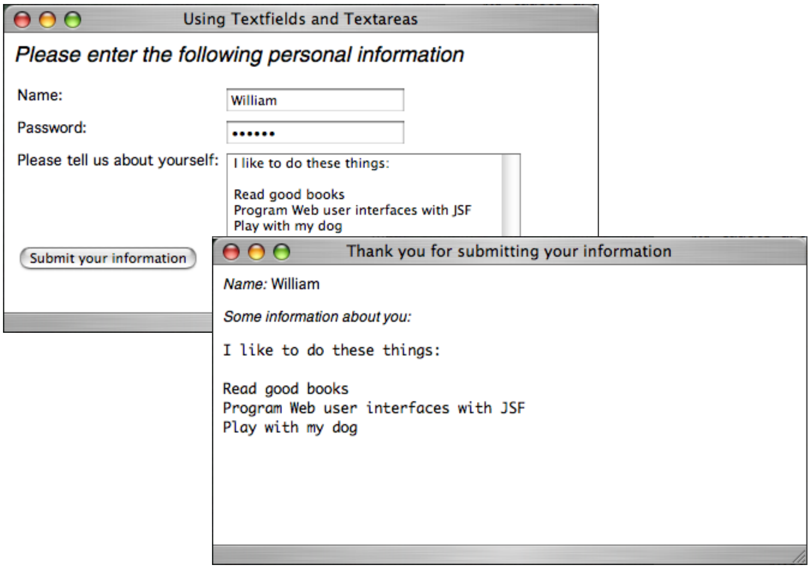
If you specify one long string for h:inputTextarea's value, the string will be placed in its entirety in one line, as you can see from the third example in Table 4–9. If you want to put data on separate lines, you can insert new line characters ('\n') to force a line break; for example, the last example in Table 4–9 accesses the dataInRows property of a backing bean. That property is implemented like this:

```
private String dataInRows = "line one\nline two\nline three";
public void setDataInRows(String newValue) {
    dataInRows = newValue;
}
public String getDataInRows() {
    return dataInRows;
}
```

Using Text Fields and Text Areas

Let's take a look at a complete example that uses text fields and text areas. The application shown in Figure 4–3 uses `h:inputText`, `h:inputSecret`, and `h:inputTextarea` to collect personal information from a user. Those components' values are wired to bean properties, which are accessed in a Thank You page that redisplays the information the user entered.

Three things are noteworthy about the following application. First, the JSF pages reference a user bean (`com.corejsf.UserBean`). Second, the `h:inputTextarea` tag transfers the text entered in a text area to the model (in this case, the user bean) as one string with embedded newlines (`'\n'`). We display that string by using the HTML `<pre>` element to preserve that formatting. Third, for illustration, we use the `style` attribute to format output. A more industrial-strength application would presumably use stylesheets exclusively to make global style changes easier to manage.


Using Textfields and Textareas
Submit your information
Thank you for submitting your information

Please enter the following personal information

Name: William

Password: ••••••

Please tell us about yourself:

```
I like to do these things:

Read good books
Program Web user interfaces with JSF
Play with my dog
```

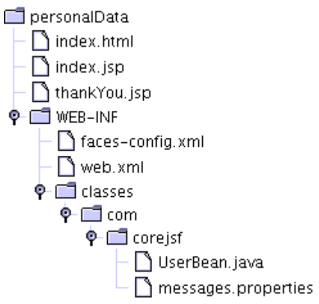
Name: William

Some information about you:

```
I like to do these things:

Read good books
Program Web user interfaces with JSF
Play with my dog
```

Figure 4–3 Using Text Fields and Text Areas

Figure 4–3 shows the directory structure for the application shown in Figure 4–3. Listing 4–3 through Listing 4–7 show the pertinent JSF pages, faces configuration file, and resource bundle.



```
personalData
    index.html
    index.jsp
    thankYou.jsp
    WEB-INF
        faces-config.xml
        web.xml
        classes
            com
                corejsf
                    UserBean.java
                    messages.properties
```

Figure 4–4 Directory Structure of the Text Fields and Text Areas Example

Listing 4–3 personalData/index.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
      <head>
         <title>
            <h:outputText value="#{msgs.indexWindowTitle}"/>
         </title>
      </head>
      <body>
         <h:outputText value="#{msgs.indexPageTitle}"
            style="font-style: italic; font-size: 1.5em"/>
         <h:form>
            <table>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.namePrompt}"/>
                  </td>
                  <td>
                     <h:inputText value="#{user.name}"/>
                  </td>
               </tr>
               <tr>
                  <td>
```

Listing 4–3 personalData/index.jsp (cont.)

```
                  <h:outputText value="#{msgs.passwordPrompt}"/>
               </td>
               <td>
                  <h:inputSecret value="#{user.password}"/>
               </td>
            </tr>
            <tr>
               <td>
                  <h:outputText value="#{msgs.tellUsPrompt}"/>
               </td>
               <td>
                  <h:inputTextarea value="#{user.aboutYourself}" rows="5"
                     cols="35"/>
               </td>
            </tr>
         </table>
         <h:commandButton value="#{msgs.submitPrompt}" action="thankYou"/>
      </h:form>
   </body>
</f:view>
</html>
```

Listing 4–4 personalData/thankYou.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
      <head>
         <title>
            <h:outputText value="#{msgs.thankYouWindowTitle}"/>
         </title>
      </head>
      <body>
         <h:outputText value="#{msgs.namePrompt}" style="font-style: italic"/>
         <h:outputText value="#{user.name}"/>
         <br/>
         <h:outputText value="#{msgs.aboutYourselfPrompt}"
            style="font-style: italic"/>
         <br/>
         <pre><h:outputText value="#{user.aboutYourself}"/></pre>
      </body>
   </f:view>
</html>
```

Listing 4–5 personalData/WEB-INF/classes/com/corejsf/UserBean.java

```java
package com.corejsf;

public class UserBean {
   private String name;
   private String password;
   private String aboutYourself;

   // PROPERTY: name
   public String getName() { return name; }
   public void setName(String newValue) { name = newValue; }

   // PROPERTY: password
   public String getPassword() { return password; }
   public void setPassword(String newValue) { password = newValue; }

   // PROPERTY: aboutYourself
   public String getAboutYourself() { return aboutYourself;}
   public void setAboutYourself(String newValue) { aboutYourself = newValue; }
}
```

Listing 4–6 personalData/WEB-INF/faces-config.xml

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

<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

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

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

</faces-config>
```

Listing 4–7 personalData/WEB-INF/classes/com/corejsf/messages.properties

```
indexWindowTitle=Using Textfields and Textareas
thankYouWindowTitle=Thank you for submitting your information
thankYouPageTitle=Thank you!
indexPageTitle=Please enter the following personal information
namePrompt=Name:
passwordPrompt=Password:
tellUsPrompt=Please tell us about yourself:
aboutYourselfPrompt=Some information about you:
submitPrompt=Submit your information
```

Displaying Text and Images

JSF applications use the following tags to display text and images:

- `h:outputText`
- `h:outputFormat`
- `h:graphicImage`

The `h:outputText` tag is one of JSF's simplest tags. With only a handful of attributes, it does not typically generate an HTML element. Instead, it generates mere text—with one exception: if you specify the `style` or `styleClass` attributes, `h:outputText` will generate an HTML `span` element. Also, `h:outputText` and `h:outputFormat` have one attribute that is unique among all JSF HTML tags: `escape`. By default, the `escape` attribute is `false`, but if you set it to `true`, the following characters `>` `<` `&` are converted to `&lt;` `&gt;` and `&amp;` respectively. Changing those characters helps prevent cross-site scripting attacks. See http://www.cert.org/advisories/CA-2000-02.html for more information about cross-site scripting attacks. Table 4–10 lists all `h:outputText` attributes.

Table 4–10 Attributes for `h:outputText`

Attributes	Description
`escape`	If set to `true`, escapes <, >, and & characters. Default value is `false`.
`binding, converter, id, rendered, styleClass, value`	Basic attributes[a]
`style, title`	HTML 4.0[b]

a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.

The h:outputFormat tag formats a compound message with parameters specified in the body of the tag; for example:

```
<h:outputFormat value="{0} is {1} years old">
    <f:param value="Bill"/>
    <f:param value="38"/>
</h:outputFormat>
```

In the preceding code fragment, the compound message is {0} is {1} years old and the parameters, specified with f:param tags, are Bill and 38. The output of the preceding code fragment is: Bill is 38 years old. The h:outputFormat tag uses a java.text.MessageFormat instance to format it's output.

Table 4–11 lists all attributes for h:outputFormat.

Table 4–11 Attributes for h:outputFormat

Attributes	Description
escape	If set to true, escapes <, >, and & characters. Default value is false.
binding, converter, id, rendered, styleClass, value	Basic attributes[a]
style, title	HTML 4.0[b]

a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.

The h:graphicImage tag lets you use a context-relative path—meaning relative to the web application's top-level directory—to display images. h:graphicImage generates an HTML img element.

Table 4–12 shows all the attributes for h:graphicImage.

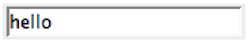
Table 4–12 Attributes for h:graphicImage

Attributes	Description
binding, id, rendered, styleClass, value	Basic attributes[a]
alt, dir, height, ismap, lang, longdesc, style, title, url, usemap, width	HTML 4.0[b]
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	DHTML events[c]

a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.
c. See Table 4–5 on page 95 for information about DHTML event attributes.

Table 4–13 shows some examples of using `h:outputText` and `h:graphicImage`.

Table 4–13 `h:outputText` **and** `h:graphicImage` **Examples**

Example	Result
`<h:outputText value="#{form.testString}"/>`	12345678901234567890
`<h:outputText value="Number #{form.number}"/>`	Number 1000
`<h:outputText` `value="<input type='text' value='hello'/>"/>`	 hello
`<h:outputText escape="true"` `value="<input type='text' value='hello'/>"/>`	`<input type="text" value="hello">`
`<h:graphicImage value="/tjefferson.jpg"/>`	
`<h:graphicImage value="/tjefferson.jpg"` `style="border: thin solid black"/>`	

The third and fourth examples in Table 4–13 illustrate use of the `escape` attribute. If the value for `h:outputText` is `<input type='text' value='hello'/>` and the `escape` attribute is `false`—as is the case for the third example in Table 4–13—the `h:outputText` tag generates an HTML `input` element. Unintentional generation of HTML elements is exactly the sort of mischief that enables miscreants to carry out cross-site scripting attacks. With the `escape` attribute set to `true`—as in the fourth example in Table 4–13—that output is transformed to harmless text, thereby thwarting a potential attack.

The final two examples in Table 4–13 show you how to use `h:graphicImage`.

Hidden Fields

JSF provides support for hidden fields with `h:inputHidden`. Table 4–14 lists all attributes for `h:inputHidden`.

Table 4–14 Attributes for `h:inputHidden`

Attributes	Attributes
`binding`, `converter`, `id`, `immediate`, `required`, `validator`, `value`, `valueChangeListener`	Basic attributes[a]

a. See Table 4–3 on page 90 for information about basic attributes.

Buttons and Links

Buttons and links are ubiquitous among web applications, and JSF provides the following tags to support them:

- `h:commandButton`
- `h:commandLink`
- `h:outputLink`

The `h:commandButton` and `h:commandLink` actions both represent JSF command components—the JSF framework fires action events and invokes actions when a button or link is activated. See Chapter 7 for more information about event handling for command components.

The `h:outputLink` tag generates an HTML `anchor` element that points to a resource such as an image or a web page. Clicking on the generated link takes you to the designated resource without further involving the JSF framework.

Table 4–15 lists the attributes shared by `h:commandButton` and `h:commandLink`.

Table 4–15 Attributes for `h:commandButton` and `h:commandLink`

Attribute	Description
`action`	*If specified as a string:* Directly specifies an outcome used by the navigation handler to determine the JSF page to load next as a result of activating the button or link *If specified as a method binding:* The method has this signature: `String methodName()`; the string represents the outcome
`actionListener`	A method binding that refers to a method with this signature: `void methodName(ActionEvent)`

Table 4–15 Attributes for `h:commandButton` and `h:commandLink` (cont.)

Attribute	Description
`charset`	For `h:commandLink` only—The character encoding of the linked reference
`image`	For `h:commandButton` only—A context-relative path to an image displayed in a button. If you specify this attribute, the HTML `input`'s type will be `image`.
`immediate`	A boolean. If `false` (the default), actions and action listeners are invoked at the end of the request life cycle; if `true`, actions and action listeners are invoked at the beginning of the life cycle. See Chapter 6 for more information about the `immediate` attribute.
`type`	*For h:commandButton:* The type of the generated `input` element: `button`, `submit`, or `reset`. The default, unless you specify the `image` attribute, is `submit`. *For h:commandLink:* The content type of the linked resource; for example, `text/html`, `image/gif`, or `audio/basic`
`value`	The label displayed by the button or link. You can specify a string or a value reference expression.
`accesskey, alt, binding, id, lang, rendered, styleClass, value`	Basic attributes[a]
`coords (h:commandLink only), dir, disabled, hreflang (h:commandLink only), lang, readonly, rel (h:commandLink only), rev (h:commandLink only), shape (h:commandLink only), style, tabindex, target (h:commandLink only), title, type`	HTML 4.0[b]
`onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect`	DHTML events[c]

a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.
c. See Table 4–5 on page 95 for information about DHTML event attributes.

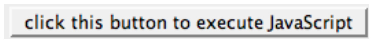
Using Command Buttons

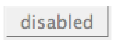
The `h:commandButton` tag generates an HTML `input` element whose type is `button`, `image`, `submit`, or `reset`, depending on the attributes you specify. Table 4–16 illustrates some uses of `h:commandButton`.

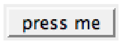
Table 4–16 `h:commandButton` **Examples**

Example	Result
`<h:commandButton value="submit" type="submit"/>`	

`<h:commandButton value="reset"`
`type="reset"/>`	

`<h:commandButton value="click this button..."`
`onclick="alert('button clicked')"`
`type="button"/>`
	

`<h:commandButton value="disabled"`
`disabled="#{not form.buttonEnabled}"/>`
	

`<h:commandButton value="#{form.buttonText}"`
`type="reset"/>`
	

The third example in Table 4–16 generates a push button—an HTML `input` element whose type is `button`—that does not result in a form submit. The only way to attach behavior to a push button is to specify a script for one of the DHTML event attributes, as we did for `onclick` in the example.

CAUTION: `h:graphicImage` and `h:commandButton` can both display images, but the way in which you specify the image is not consistent between the two tags. `h:commandButton` requires a context path, whereas the context path is added automatically by `h:graphicImage`. For example, for an application named `myApp`, here's how you specify the same image for each tag:

```
<h:commandButton image="/myApp/imageFile.jpg"/>
<h:graphicImage value="/imageFile.jpg"/>
```

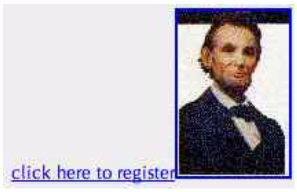
The `h:commandLink` tag generates an HTML `anchor` element that acts like a form submit button. Table 4–17 shows some `h:commandLink` examples.

Table 4–17 `h:commandLink` **Examples**

Example	Result

`<h:commandLink>`
   `<h:outputText value="register"/>`
`</h:commandLink>`
	register

`<h:commandLink style="font-style: italic">`
   `<h:outputText value="#{msgs.linkText}"/>`
`</h:commandLink>`
	click here to register

`<h:commandLink>`
   `<h:outputText value="#{msgs.linkText}"/>`
   `<h:graphicImage value="/registration.jpg"/>`
`</h:commandLink>`
	 click here to register

`<h:commandLink value="welcome"`
   `actionListener="#{form.useLinkValue}"`
   `action="#{form.followLink}">`
	welcome

`<h:commandLink>`
   `<h:outputText value="welcome"/>`
   `<f:param name="outcome" value="welcome"/>`
`</h:commandLink>`
	welcome

`h:commandLink` generates JavaScript to make links act like buttons. For example, here is the HTML generated by the first example in Table 4–17:

```
<a href="#"
onclick="document.forms['_id0']['_id0:_id2'].value='_id0:_id2';document.forms['_id0']
.submit()">register</a>
```

When the link is clicked, the `anchor` element's value is set to the `h:commandLink`'s client ID and the enclosing form is submitted. That submission sets the JSF life cycle in motion and, because the `href` attribute is `'#'`, the current page will be reloaded unless an action associated with the link returns a non-`null` outcome. See Chapter 3 for more information about JSF navigation.

You can place as many JSF HTML tags as you want in the body of an `h:commandLink` tag—each corresponding HTML element is part of the link. So, for example, if you click on either the text or image in the third example in Table 4–17, the link's form will be submitted.

The next-to-last example in Table 4–17 attaches an action listener, in addition to an action, to a link. The last example in Table 4–17 embeds an `f:param` tag in the body of the `h:commandLink` tag. When you click on the link, a request parameter with the name and value specified with the `f:param` tag is created by the link. You can use that request parameter any way you like. Chapter 7 tells you how to use request parameters to affect navigation outcomes.

Both `h:commandButton` and `h:commandLink` submit requests and subsequently invoke the JSF life cycle. Although those tags are useful, sometimes you just need a link that simply loads a resource without invoking the JSF life cycle. In that case, you can use the `h:outputLink` tag. Table 4–18 lists all attributes for `h:outputLink`.

Table 4–18 Attributes for `h:outputLink`

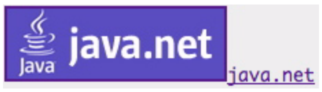
Attributes	Description
`accesskey, binding, converter, id, lang, rendered, styleClass, value`	Basic attributes[a]
`charset, coords, dir, hreflang, lang, rel, rev, shape, style, tabindex, target, title, type`	HTML 4.0[b]
`onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup`	DHTML events[c]

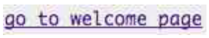
a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.
c. See Table 4–5 on page 95 for information about DHTML event attributes.

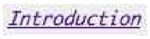
Like `h:commandLink`, `h:outputLink` generates an HTML `anchor` element. But unlike `h:commandLink`, `h:outputLink` does not generate JavaScript to make the link act like a submit button. The value of the `h:outputLink` `value` attribute is used for the anchor's `href` attribute, and the contents of the `h:outputLink` body are used to populate the body of the `anchor` element. Table 4–19 shows some `h:outputLink` examples.

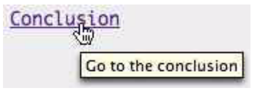
Table 4–19 `h:outputLink` **Examples**

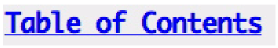
Example	Result

`<h:outputLink value="http://java.net">`
`   <h:graphicImage value="java-dot-net.jpg"/>`
`   <h:outputText value="java.net"/>`
`</h:outputLink>`
	

`<h:outputLink value="#{form.welcomeURL}">`
`   <h:outputText value="#{form.welcomeLinkText}"/>`
`</h:outputLink>`
	

`<h:outputLink value="#introduction">`
`   <h:outputText value="Introduction"`
`      style="font-style: italic"/>`
`</h:outputLink>`
	

`<h:outputLink value="#conclusion"`
`   title="Go to the conclusion">`
`   <h:outputText value="Conclusion"/>`
`</h:outputLink>`
	

`<h:outputLink value="#toc"`
`   title="Go to the table of contents">`
`   <f:verbatim>`
`      <h2>Table of Contents</h2>`
`   </f:verbatim>`
`</h:outputLink>`
	

The first example in Table 4–17 is a link to http://java.net. The second example uses properties stored in a bean for the link's URL and text. Those properties are implemented like this:

```
private String welcomeURL = "/outputLinks/faces/welcome.jsp";
   public String getWelcomeURL() {
      return welcomeURL;
   }
   private String welcomeLinkText = "go to welcome page";
   public String getWelcomeLinkText() {
      return welcomeLinkText;
   }
```

The last three examples in Table 4–17 are links to named anchors in the same JSF page. Those anchors look like this:

```
<a name="introduction">Introduction</a>
...
<a name="toc">Table of Contents</a>
...
<a name="conclusion">Conclusion</a>
...
```

Notice that the last example in Table 4–17 uses `f:verbatim`. You cannot simply place text inside the `h:outputLink` tag. For example, `<h:outputLink...>Introduction</h:outputLink>` would not work correctly—the text would appear outside the link. The remedy is to place the text inside a component, either with `h:outputText` or with `f:verbatim`. Generally, you use `h:outputLink` for text, `f:verbatim` for HTML—see the first and last examples in Table 4–17.

Using Command Links

Now that we've discussed the details of JSF tags for buttons and links, let's take a look at a complete example. Figure 4–5 shows the application discussed in "Using Text Fields and Text Areas" on page 104 with two links that let you select either English or German locales. When a link is activated, an action changes the view's locale and the JSF implementation reloads the current page.

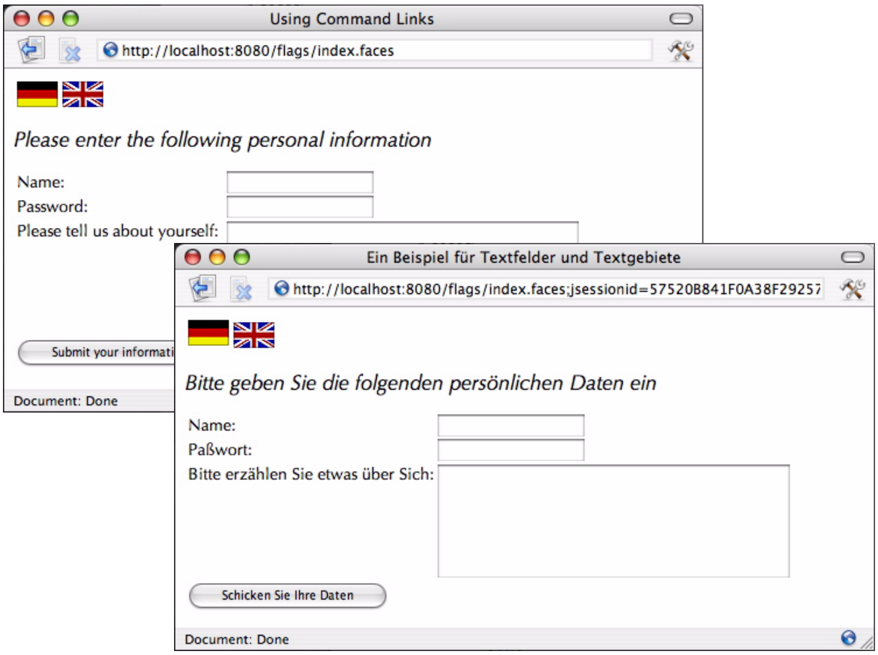

Using Command Links
http://localhost:8080/flags/index.faces
Please enter the following personal information
Name:
Password:
Please tell us about yourself:
Submit your informati
Document: Done
Ein Beispiel für Textfelder und Textgebiete
http://localhost:8080/flags/index.faces;jsessionid=57520B841F0A38F29257
Bitte geben Sie die folgenden persönlichen Daten ein
Name:
Paßwort:
Bitte erzählen Sie etwas über Sich:
Schicken Sie Ihre Daten
Document: Done

Figure 4–5 Using Command Links to Change Locales

The two links are implemented like this:

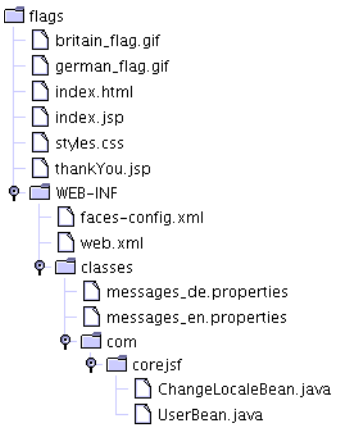
```
<h:form>
    ...
    <h:commandLink action="#{localeChanger.germanAction}">
        <h:graphicImage value="/german_flag.gif"
            style="border: 0px"/>
        <f:param name="locale" value="german"/>
    </h:commandLink>

    <h:commandLink action="#{localeChanger.englishAction}">
        <h:graphicImage value="/britain_flag.gif"
            style="border: 0px"/>
        <f:param name="locale" value="english"/>
    </h:commandLink>
    ...
</h:form>
```

Both links specify an image, request parameter, and an action method. Those methods look like this:

```
public class ChangeLocaleBean {
   public String germanAction() {
      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot.setLocale(Locale.GERMAN);
      return null;
   }
   public String englishAction() {
      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot().setLocale(Locale.ENGLISH);
      return null;
   }
}
```

Both actions set the locale of the view. And because we have not specified any navigation for this application, the JSF implementation will reload the current page after the form is submitted. When the page is reloaded, it is localized for English or German and the page redisplays accordingly.

Figure 4–6 shows the directory structure for the application and Listing 4–8 through Listing 4–10 show the associated JSF pages and the faces configuration file.



```
flags
    britain_flag.gif
    german_flag.gif
    index.html
    index.jsp
    styles.css
    thankYou.jsp
    WEB-INF
        faces-config.xml
        web.xml
        classes
            messages_de.properties
            messages_en.properties
            com
                corejsf
                    ChangeLocaleBean.java
                    UserBean.java
```

Figure 4–6 Directory Structure of the Text Fields and Text Areas Example

Listing 4–8 flags/index.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <f:loadBundle basename="messages" var="msgs"/>
      <head>
         <link href="styles.css" rel="stylesheet" type="text/css"/>
         <title>
            <h:outputText value="#{msgs.indexWindowTitle}"/>
         </title>
      </head>
      <body>
         <h:form>
            <table>
               <tr>
                  <td>
                     <h:commandLink immediate="true"
                        action="#{localeChanger.germanAction}">
                        <h:graphicImage value="/german_flag.gif"
                           style="border: 0px"/>
                     </h:commandLink>
                  </td
                  <td>
                     <h:commandLink immediate="true"
                        action="#{localeChanger.englishAction}">
```

Listing 4–8 flags/index.jsp (cont.)

```
                        <h:graphicImage value="/britain_flag.gif"
                           style="border: 0px"/>
                     </h:commandLink>
                  </td>
               </tr>
            </table>
            <p>
               <h:outputText value="#{msgs.indexPageTitle}"
                  style="font-style: italic; font-size: 1.3em"/>
            </p>
            <table>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.namePrompt}"/>
                  </td>
                  <td>
                     <h:inputText value="#{user.name}"/>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.passwordPrompt}"/>
                  </td>
                  <td>
                     <h:inputSecret value="#{user.password}"/>
                  </td>
               </tr>
               <tr>
                  <td style="vertical-align: top">
                     <h:outputText value="#{msgs.tellUsPrompt}"/>
                  </td>
                  <td>
                     <h:inputTextarea value="#{user.aboutYourself}" rows="5"
                        cols="35"/>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:commandButton value="#{msgs.submitPrompt}"
                        action="thankYou"/>
                  </td>
               </tr>
            </table>
         </h:form>
      </body>
   </f:view>
</html>
```

Listing 4–9 flags/WEB-INF/classes/com/corejsf/UserBean.java

```
package com.corejsf;

public class UserBean {
   private String name;
   private String password;
   private String aboutYourself;

   // PROPERTY: name
   public String getName() { return name; }
   public void setName(String newValue) { name = newValue; }

   // PROPERTY: password
   public String getPassword() { return password; }
   public void setPassword(String newValue) { password = newValue; }

   // PROPERTY: aboutYourself
   public String getAboutYourself() { return aboutYourself;}
   public void setAboutYourself(String newValue) { aboutYourself = newValue; }
}
```

Listing 4–10 flags/WEB-INF/faces-config.xml

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

<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

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

   <managed-bean>
      <managed-bean-name>localeChanger</managed-bean-name>
      <managed-bean-class>com.corejsf.ChangeLocaleBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>

```

Listing 4–10 flags/WEB-INF/faces-config.xml (cont.)

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

</faces-config>
```

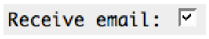
Selection Tags

JavaServer Faces has seven tags for making selections:

- `h:selectBooleanCheckbox`
- `h:selectManyCheckbox`
- `h:selectOneRadio`
- `h:selectOneListbox`
- `h:selectManyListbox`
- `h:selectOneMenu`
- `h:selectManyMenu`

Table 4–20 shows examples of each tag listed above.

Table 4–20 Selection Tag Examples

Tag	Generated HTML	Examples
`h:selectBooleanCheckbox`	`<input type="checkbox">`	 Receive email: ✓

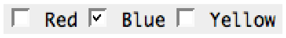
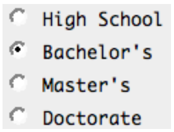
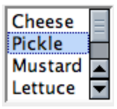
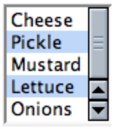
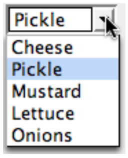
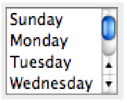
`h:selectManyCheckbox`	`<table>`
	`   ...`
	`   <label>`
	`      <input type="checkbox"/>`
	`   </label>`
	`   ...`
	`</table>`
		 ☐ Red ✓ Blue ☐ Yellow

Table 4–20 Selection Tag Examples (cont.)

Tag	Generated HTML	Examples
`h:selectOneRadio`	`<table> ... <label> <input type="radio"/> </label> ... </table>`	
`h:selectOneListbox`	`<select> <option value="Cheese"> Cheese </option> ... </select>`	
`h:selectManyListbox`	`<select multiple> <option value="Cheese"> Cheese </option> ... </select>`	
`h:selectOneMenu`	`<select size="1"> <option value="Cheese"> Cheese </option> ... </select>`	
`h:selectManyMenu`	`<select multiple size="1"> <option value="Sunday"> Sunday </option> ... </select>`	

The h:selectBooleanCheckbox is the simplest selection tag—it renders a checkbox you can wire to a boolean bean property. You can also render a set of checkboxes with h:selectManyCheckbox.

Tags whose names begin with selectOne let you select one item from a collection. The selectOne tags render sets of radio buttons, single-select menus, or listboxes. The selectMany tags render sets of checkboxes, multiselect menus, or listboxes.

All selection tags share an almost identical set of attributes, listed in Table 4–21.

Table 4–21 Attributes for h:selectBooleanCheckbox, h:selectManyCheckbox, h:selectOneRadio, h:selectOneListbox, h:selectManyListbox, h:selectOneMenu, h:selectManyMenu

Attributes	Description
disabledClass	CSS class for disabled elements—for h:selectOneRadio and h:selectManyCheckbox only
enabledClass	CSS class for enabled elements—for h:selectOneRadio and h:selectManyCheckbox only
layout	Specification for how elements are laid out: lineDirection (horizontal) or pageDirection (vertical)—for h:selectOneRadio and h:selectManyCheckbox only
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Basic attributes[a]
accesskey, border, dir, disabled, lang, readonly, style, size, tabindex, title	HTML 4.0[b]—border is applicable to h:selectOneRadio and h:selectManyCheckbox only. size is applicable to h:selectOneListbox and h:selectManyListbox only.
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	DHTML events[c]

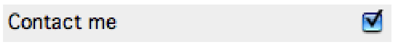
a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.
c. See Table 4–5 on page 95 for information about DHTML event attributes.

Checkboxes and Radio Buttons

Two JSF tags represent checkboxes:

- `h:selectBooleanCheckbox`
- `h:selectManyCheckbox`

`h:selectBooleanCheckbox` represents a single checkbox that you can wire to a `boolean` bean property. Here is an example.



In your JSF page, you do this:

```
<h:selectBooleanCheckbox value="#{form.contactMe}"/>
```

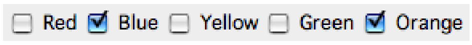
In your backing bean, provide a read-write property:

```
private boolean contactMe;
public void setContactMe(boolean newValue) {
    contactMe = newValue;
}
public boolean getContactMe() {
    return contactMe;
}
```

The generated HTML looks something like this:

```
<input type="checkbox" name="_id2:_id7"/>
```

You can create a group of checkboxes with `h:selectManyCheckbox`. As the tag name implies, you can select one or more of the checkboxes in the group. You specify that group within the body of `h:selectManyCheckbox`, either with one or more `f:selectItem` tags or one `f:selectItems` tag. See "Items" on page 130 for more information about those core tags. For example, here's a group of checkboxes for selecting colors.



The `h:selectManyCheckbox` tag looks like this:

```
<h:selectManyCheckbox value="#{form.colors}">
    <f:selectItem itemValue="Red"/>
    <f:selectItem itemValue="Blue"/>
    <f:selectItem itemValue="Yellow"/>
    <f:selectItem itemValue="Green"/>
    <f:selectItem itemValue="Orange"/>
</h:selectManyCheckbox>
```

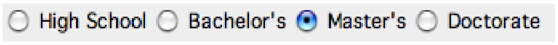
The checkboxes are specified with `f:selectItem` tags. See "f:selectItem" on page 130 for more information on that tag.

h:selectManyCheckbox generates an HTML table element; for example, here's the generated HTML for our color example:

```
<table>
    <tr>
        <td>
            <label for="_id2:_id14">
                <input name="_id2:_id14" value="Red" type="checkbox">Red</input>
            </label>
        </td>
    </tr>
    ...
</table>
```

Each color is an input element, wrapped in a label for accessibility purposes. That label is placed in a td element.

Radio buttons are implemented with h:selectOneRadio. Here is an example.



The h:selectOneRadio value attribute specifies the currently selected item. Once again, we use multiple f:selectItem tags to populate the radio buttons:

```
<h:selectOneRadio value="#{form.education}">
    <f:selectItem itemValue="High School"/>
    <f:selectItem itemValue="Bachelor's"/>
    <f:selectItem itemValue="Master's"/>
    <f:selectItem itemValue="Doctorate"/>
</h:selectOneRadio>
```

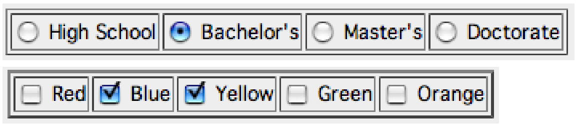
Like h:selectManyCheckbox, h:selectOneRadio generates an HTML table. Here's the table generated by the preceding tag:

```
<table>
    <tr>
        <td>
            <label for="_id2:_id14">
                <input name="_id2:_id14" value="High School" type="radio">
                    High School
                </input>
            </label>
        </td>
    </tr>
    ...
</table>
```

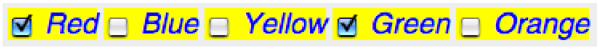
Besides generating HTML tables, `h:selectOneRadio` and `h:selectManyCheckbox` have something else in common—a handful of attributes unique to those two tags.

- `border`
- `enabledClass`
- `disabledClass`
- `layout`

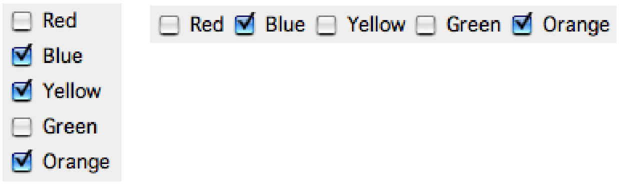
The `border` attribute specifies the width of the border. For example, here are radio buttons and checkboxes with borders of 1 and 2, respectively.



`enabledClass` and `disabledClass` specify CSS classes used when the checkboxes or radio buttons are enabled or disabled, respectively. For example, the following picture shows an enabled class with an italic font style, blue color, and yellow background.



The `layout` attribute can be either `lineDirection` (horizontal) or `pageDirection` (vertical). For example, the following checkboxes on the left have a `pageDirection` layout and the checkboxes on the right are `lineDirection`.



The `pageDirection` value for the `layout` attribute is case insensitive, so for example, you could use `PAGEDIRECTION` if you prefer. `LINEDIRECTION` is the default.

NOTE: You might wonder why `layout` attribute values aren't `horizontal` and `vertical`, instead of `lineDirection` and `pageDirection`, respectively. Although `lineDirection` and `pageDirection` are indeed horizontal and vertical for Latin-based languages, that's not always the case for other languages. For example, a Chinese browser that displays text top to bottom could regard `lineDirection` as vertical and `pageDirection` as horizontal.

Menus and Listboxes

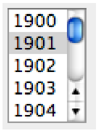
Menus and listboxes are represented by the following tags:

- `h:selectOneListbox`
- `h:selectManyListbox`
- `h:selectOneMenu`
- `h:selectManyMenu`

The attributes for the preceding tags are listed in Table 4–21 on page 124, so that discussion is not repeated here.

Menu and listbox tags generate HTML `select` elements. The menu tags add a `size="1"` attribute to the `select` element. That size designation is all that separates menus and listboxes.

Here's a single-select listbox.



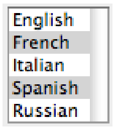
The corresponding listbox tag looks like this:

```
<h:selectOneListbox value="#{form.year}" size="5">
    <f:selectItem value="1900"/>
    <f:selectItem value="1901"/>
    ...
</h:selectOneListbox>
```

Notice that we've used the `size` attribute to specify the number of visible items. The generated HTML looks like this:

```
<select name="_id2:_id11" size="5">
    <option value="1900">1900</option>
    <option value="1901">1901</option>
    ...
</select>
```

Use `h:selectManyListbox` for multiselect listboxes like this one.



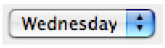
The listbox tag looks like this:

```
<h:selectManyListbox value="#{form.languages}">
    <f:selectItem itemValue="English"/>
    <f:selectItem itemValue="French"/>
    <f:selectItem itemValue="Italian"/>
    <f:selectItem itemValue="Spanish"/>
    <f:selectItem itemValue="Russian"/>
</h:selectManyListbox>
```

This time we don't specify the size attribute, so the listbox grows to accommodate all its items. The generated HTML looks like this:

```
<select name="_id2:_id11" multiple>
    <option value="English">English</option>
    <option value="French">French</option>
    ...
</select>
```

Use h:selectOneMenu and h:selectManyMenu for menus. A single-select menu looks like this.



h:selectOneMenu created the preceding menu:

```
<h:selectOneMenu value="#{form.day}">
    <f:selectItem itemValue="Sunday"/>
    <f:selectItem itemValue="Monday"/>
    <f:selectItem itemValue="Tuesday"/>
    <f:selectItem itemValue="Wednesday"/>
    <f:selectItem itemValue="Thursday"/>
    <f:selectItem itemValue="Friday"/>
    <f:selectItem itemValue="Saturday"/>
</h:selectOneMenu>
```

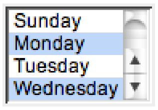
Here's the generated HTML:

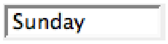
```
<select name="_id2:_id17" size="1">
    <option value="Sunday">Sunday</option>
    ...
</select>
```

h:selectManyMenu is used for multiselect menus. That tag generates HTML that looks like this:

```
<select name="_id2:_id17" multiple size="1">
    <option value="Sunday">Sunday</option>
    ...
</select>
```

That HTML does not yield consistent results among browsers. For example, here's `h:selectManyMenu` on Internet Explorer (left) and Netscape (right).





NOTE: In HTML, the distinction between menus and listboxes is artificial. Menus and listboxes are both HTML `select` elements. The only distinction: menus always have a `size="1"` attribute.

Browsers consistently render single-select menus as drop-down lists, as expected. But they do not consistently render multiple select menus, specified with `size="1"` and `multiple` attributes. Instead of rendering a drop-down list with multiple selection, as you might expect, browsers render absurdities such as tiny scrollbars that are nearly impossible to manipulate (Windows IE)—as shown above—or no scroll-bar at all, leaving you to navigate with arrow keys (Mozilla).

Starting with "Selection Tags" on page 122, we've consistently used multiple `f:selectItem` to populate `select` components. Now that we're familiar with the fundamentals of selection tags, let's take a closer look at `f:selectItem` and the related `f:selectItems` tags.

Items

All selection tags except `h:selectBooleanCheckbox` use `f:selectItem` or `f:selectItems` to specify their items. Let's look at those core tags, starting with `f:selectItem`.

f:selectItem

You use `f:selectItem` to specify single selection items, like this:

```
<h:selectOneMenu value="#{form.condiments}">
    <f:selectItem itemValue="Cheese"/>
    <f:selectItem itemValue="Pickle"/>
    <f:selectItem itemValue="Mustard"/>
    <f:selectItem itemValue="Lettuce"/>
    <f:selectItem itemValue="Onions"/>
</h:selectOneRadio>
```

The values—`Cheese`, `Pickle`, etc.—are transmitted as request parameter values when a selection is made from the menu and the menu's form is subsequently submitted. Those values are also used as labels for the menu items. Sometimes you want to specify different values for request parameter values and item labels, so `f:selectItem` also has an `itemLabel` attribute:

```
<h:selectOneMenu value="#{form.condiments}">
    <f:selectItem itemValue="1" itemLabel="Cheese"/>
    <f:selectItem itemValue="2" itemLabel="Pickle"/>
    <f:selectItem itemValue="3" itemLabel="Mustard"/>
    <f:selectItem itemValue="4" itemLabel="Lettuce"/>
    <f:selectItem itemValue="5" itemLabel="Onions"/>
</h:selectOneRadio>
```

In the preceding code, the item values are strings. "Binding the value Attribute" on page 136 shows you how to use different data types for item values.

In addition to labels and values, you can also supply item descriptions and specify an item's disabled state:

```
    <f:selectItem itemLabel="Cheese" itemValue="#{form.cheeseValue}"
            itemDescription="used to be milk"
               itemDisabled="true"/>
```

Item descriptions are for tools only—they do not affect the generated HTML. The `disabled` attribute, however, is passed to HTML. The `f:selectItem` tag has the following attributes:

Table 4–22 Attributes for `f:selectItem`

Attribute	Description
`binding`	Component binding—see Chapter 2 for more information about component bindings.
`id`	Component ID
`itemDescription`	Description used by tools only
`itemDisabled`	Boolean value that sets the item's disabled property
`itemLabel`	Text shown by the item
`itemValue`	Item's value, which is passed to the server as a request parameter
`value`	Value binding expression that points to a `SelectItem` instance

You can use `f:selectItem`'s `value` attribute to access `SelectItem` instances created in a bean:

```
<f:selectItem value="#{form.cheeseItem}"/>
```

The value binding expression for the `value` attribute points to a method that returns a `javax.faces.model.SelectItem` instance:

```
public SelectItem getCheeseItem()  {
    return new SelectItem("Cheese");
}
```

API `javax.faces.model.SelectItem`

- `SelectItem(Object value)`
  Creates a `SelectItem` with a value. The item label is obtained by applying `toString()` to the value.
- `SelectItem(Object value, String label)`
  Creates a `SelectItem` with a value and a label.
- `SelectItem(Object value, String label, String description)`
  Creates a `SelectItem` with a value, label, and description.
- `SelectItem(Object value, String label, String description, boolean disabled)`
  Creates a `SelectItem` with a value, label, description, and disabled state.

f:selectItems

As we saw in "f:selectItem" on page 130, `f:selectItem` is versatile, but it's tedious for specifying more than a few items. The first code fragment in "f:selectItem" on page 130 can be reduced to the following with `f:selectItems`:

```
<h:selectOneRadio>
    <f:selectItems value="#{form.condiments}"/>
</h:selectOneRadio>
```

The value binding expression `#{form.condiments}` could point to an array of `Selec-tItem` instances:

```
private SelectItem[] condiments = {
    new SelectItem(new Integer(1), "Cheese"),
    new SelectItem(new Integer(2), "Pickle"),
    new SelectItem(new Integer(3), "Mustard"),
    new SelectItem(new Integer(4), "Lettuce"),
    new SelectItem(new Integer(5), "Onions")
};

public SelectItem[] getCondiments() {
    return condiments;
}
```

The `f:selectItems value` attribute must be a value binding expression that points to one of the following:

- A single `SelectItem` instance
- A collection of `SelectItem` instances
- An array of `SelectItem` instances
- A map whose entries represent `SelectItem` labels and values

If you specify a map, the JSF implementation creates a SelectItem instance for every entry in the map. The entry's key is used as the item's label, and the entry's value is used as the item's value. For example, here are condiments specified with a map:

```
private Map condiments = null;

public Map getCondiments() {
    if(condiments == null) {
        condiments = new HashMap();
        condiments.put("Cheese",  new Integer(1)); // key,value
        condiments.put("Pickle",  new Integer(2));
        condiments.put("Mustard", new Integer(3));
        condiments.put("Lettuce", new Integer(4));
        condiments.put("Onions",  new Integer(5));
    }
    return condiments;
}
```

Note that you cannot specify item descriptions or disabled status when you use a map.

NOTE: A single f:selectItems tag is usually better than multiple f:selectItem tags. If the number of items changes, you have to modify only Java code if you use f:selectItems, whereas f:selectItem may require you to modify both Java code and JSF pages.

NOTE: If you use SelectItem, you couple your code to the JSF API. This makes the Map alternative seemingly attractive.

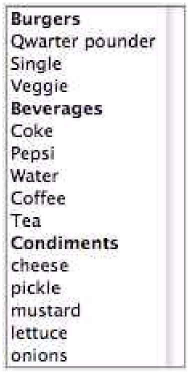
CAUTION: If you use a Map for select items, JSF turns map keys into item labels and map values into item values. When a user selects an item, the JSF implementation returns a value in your map, not a key. That makes it painful to dig out the corresponding key.

CAUTION: If you use a Map for select items, pay attention to the item ordering. If you use a TreeMap, the values (which are the keys of the map) are sorted alphabetically. For example, weekdays would be neatly arranged as Friday Monday Saturday Sunday Thursday Tuesday Wednesday.

NOTE: It's a SCAM: Can't remember what you can specify for the `f:selectItems` `value` attribute? Single select item; Collection of select items; Array of select items; Map.

Item Groups

You can group menu or listbox items together, like this.



Here are the JSF tags that define the listbox:

```
<h:selectManyListbox>
    <f:selectItems value="#{form.menuItems}"/>
</h:selectManyListbox>
```

The `menuItems` method returns a `SelectItem` array:

```
public SelectItem[] getMenuItems() { return menuItems; }
```

The `menuItems` array is instantiated like this:

```
private static SelectItem[] menuItems = { burgers,  beverages, condiments };
```

The `burgers`, `beverages`, and `condiments` variables are `SelectItemGroup` instances that are instantiated like this:

```
private SelectItemGroup burgers =
    new  SelectItemGroup("Burgers", // value
                "burgers on the menu", // description
                false, // disabled
                burgerItems); // select items

  private SelectItemGroup beverages =
      new  SelectItemGroup("Beverages", // value
                "beverages on the menu", // description
                false, // disabled
                beverageItems); // select items

    private SelectItemGroup condiments =
         new  SelectItemGroup("Condiments", // value
                "condiments on the menu", // description
                false, // disabled
                condimentItems); // select items
```

Notice we're using SelectItemGroups to populate an array of SelectItems. We can do that because SelectItemGroup extends SelectItem. The groups are created and initialized like this:

```
private SelectItem[] burgerItems = {
    new SelectItem("Qwarter pounder"),
    new SelectItem("Single"),
    new SelectItem("Veggie"),
};
private SelectItem[] beverageItems = {
    new SelectItem("Coke"),
    new SelectItem("Pepsi"),
    new SelectItem("Water"),
    new SelectItem("Coffee"),
    new SelectItem("Tea"),
};
private SelectItem[] condimentItems = {
    new SelectItem("cheese"),
    new SelectItem("pickle"),
    new SelectItem("mustard"),
    new SelectItem("lettuce"),
    new SelectItem("onions"),
};
```

SelectItemGroup instances encode HTML optgroup elements. For example, the preceding example generates the following HTML:

```
<select name="_id0:_id1" multiple size="16" onclick="submit()">
    <optgroup label="Burgers">
        <option value="1"  selected>Qwarter pounder</option>
        <option value="2">Single</option>
        <option value="3">Veggie</option>
    </optgroup>

    <optgroup label="Beverages">
        <option value="4"  selected>Coke</option>
        <option value="5">Pepsi</option>
        <option value="6">Water</option>
        <option value="7">Coffee</option>
        <option value="8">Tea</option>
    </optgroup>

    <optgroup label="Condiments">
        <option value="9">cheese</option>
        <option value="10">pickle</option>
        <option value="11">mustard</option>
        <option value="12">lettuce</option>
        <option value="13">onions</option>
    </optgroup>
</select>
```

NOTE: The HTML 4.01 specification does not allow nested `optgroup` elements, which would be useful for things like cascading menus. The specification does mention that future HTML versions may support that behavior.

`javax.faces.model.SelectItemGroup`

- `SelectItemGroup(String label)`
  Creates a group with a label, but no selection items
- `SelectItemGroup(String label, String description, boolean disabled, SelectItem[] items)`
  Create a group with a label, a description (which is ignored by the JSF 1.0 Reference Implementation), a `boolean` that disables all of the items when `true`, and an array of select items used to populate the group
- `setSelectItems(SelectItem[] items)`
  Sets a group's array of `SelectItems`

Binding the value Attribute

In all likelihood, whether you're using a set of checkboxes, a menu, or a listbox, you'll want to keep track of selected items. For that purpose, you can exploit `selectOne` and `selectMany` tags, which have `value` attributes that represent selected items. For example, you can specify a selected item with the `h:selectOneRadio` `value` attribute, like this:

```
<h:selectOneRadio value="#{form.education}">
   <f:selectItems value="#{form.educationItems}"/>
</h:selectOneRadio>
```

The #{form.education} value reference expression refers to the education property of a bean named form. That property is implemented like this:

```
private Integer education = null;
public Integer getEducation() {
    return education;
}
public void setEducation(Integer newValue) {
    education = newValue;
}
```

Notice that the education property type is Integer. That means the radio buttons must have Integer values. Those radio buttons are specified with f:selectItems, with a value attribute that points to the educationItems property of the form bean:

```
private SelectItem[] educationItems = {
    new SelectItem(new Integer(1), "High School"), // value, label
    new SelectItem(new Integer(2), "Bachelors"),
    new SelectItem(new Integer(3), "Masters"),
    new SelectItem(new Integer(4), "PHD")
};
public SelectItem[] getEducationItems() {
    return educationItems;
}
```

In the preceding example, we arbitrarily chose Integer to represent education level. You can choose any type you like as long as the properties for items and selected item have matching types.

You can keep track of multiple selections with a selectMany tag. Those tags also have a value attribute that lets you specify one or more selected items. That attribute's value must be an array or list of convertible types.

Let's take a look at some different data types. We'll use h:selectManyListbox to let a user choose multiple condiments:

```
<h:selectManyListbox value="#{form.condiments}">
   <f:selectItems value="#{form.condimentItems}"/>
</h:selectManyListbox>
```

Here are the condimentItems and condiments properties:

```
private static SelectItem[] condimentItems = {
      new SelectItem(new Integer(1), "Cheese"),
      new SelectItem(new Integer(2), "Pickle"),
      new SelectItem(new Integer(3), "Mustard"),
      new SelectItem(new Integer(4), "Lettuce"),
      new SelectItem(new Integer(5), "Onions"),
   };
   public SelectItem[] getCondimentItems() {
      return condimentItems;
   }

   private Integer[] condiments;
   public void setCondiments(Integer[] newValue) {
      condiments = newValue;
   }
   public Integer[] getCondiments() {
      return condiments;
   }
```

Instead of an Integer array for the condiments property, we could have used the corresponding primitive type, int:

```
private int[] condiments;
public void setCondiments(int[] newValue) {
      condiments = newValue;
}
public int[] getCondiments() {
    return condiments;
}
```

If you use strings for item values, you can use a string array or list of strings for your selected items property:

```
private static SelectItem[] condimentItems = {
      new SelectItem("cheese", "Cheese"),
      new SelectItem("pickle", "Pickle"),
      new SelectItem("mustard", "Mustard"),
      new SelectItem("lettuce", "Lettuce"),
      new SelectItem("onions", "Onions"),
   };
   public SelectItem[] getCondimentItems() {
      return condimentItems;
   }

   private String[] condiments;
   public void setCondiments(String[] newValue) {
      condiments = newValue;
   }
   public String[] getCondiments() {
      return condiments;
   }
```

The preceding condiments property is an array of strings. You could use a list instead:

```
private List condiments = null;
public List getCondiments() {
    if(condiments == null) {
        condiments = new LinkedList();
        condiments.add(new SelectItem("cheese", "Cheese"));
        condiments.add(new SelectItem("pickle", "Pickle"));
        condiments.add(new SelectItem("mustard", "Mustard"));
        condiments.add(new SelectItem("lettuce", "Lettuce"));
        condiments.add(new SelectItem("onions", "Onions"));
    }
    return condiments;
}
public void setCondiments(List newValue) {
    selectedCondiments = newValue;
}
```

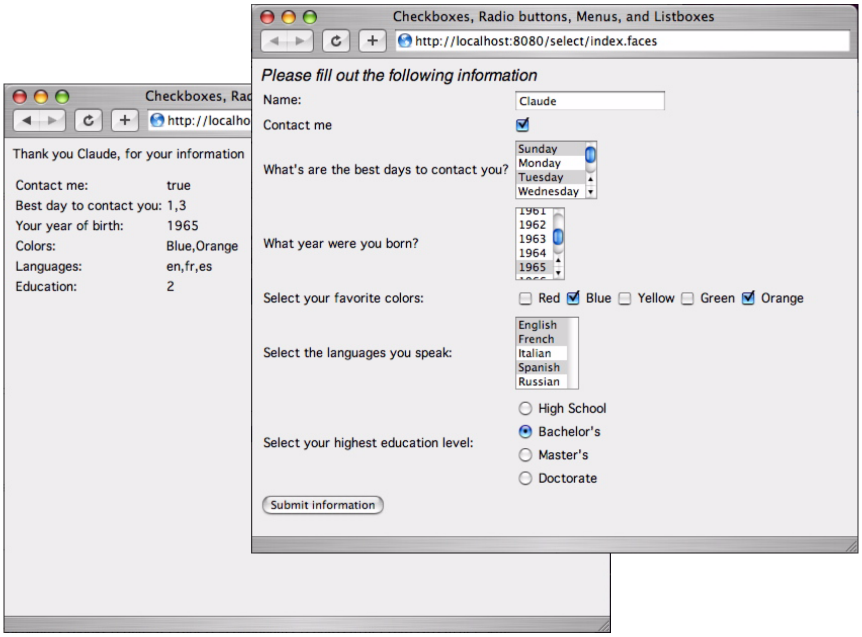
Finally, you can specify user-defined types for selected items. If you do, you must provide a converter so that the JSF implementation can convert the strings on the client to objects on the server.

NOTE: If you use application-specific objects in selection lists, you must implement a converter and either register it for your application-specific class, or specify the converter ID with the tag's `converter` attribute. If you do not provide a converter, the JSF implementation will create conversion errors when it tries to convert strings to instances of your class, and your model values will not be updated. See Chapter 6 for more information on converters.

All Together: Checkboxes, Radio Buttons, Menus, and Listboxes

We close out our section on selection tags with an example that exercises nearly all of those tags. That example, shown in Figure 4–7, implements a form requesting personal information. We use an `h:selectBooleanCheckbox` to determine whether the user wants to receive email, and `h:selectOneMenu` lets the user select the best day of the week for us to call. The year listbox is implemented with `h:selectOneListbox`; the race checkboxes are implemented with `h:selectManyCheckbox`; the education level is implemented with `h:selectOneRadio`.

When the user submits the form, JSF navigation takes us to a JSF page that shows the data the user entered.


Checkboxes, Rad
http://localho
Checkboxes, Radio buttons, Menus, and Listboxes
http://localhost:8080/select/index.faces
Submit information

Thank you Claude, for your information

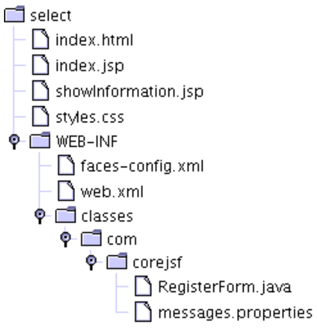
Contact me:	true
Best day to contact you:	1,3
Your year of birth:	1965
Colors:	Blue,Orange
Languages:	en,fr,es
Education:	2

Please fill out the following information

Name:	Claude
Contact me	✓
What's are the best days to contact you?	Sunday, Monday, Tuesday, Wednesday
What year were you born?	1961, 1962, 1963, 1964, 1965
Select your favorite colors:	Red [✓ Blue] Yellow Green [✓ Orange]
Select the languages you speak:	English, French, Italian, Spanish, Russian
Select your highest education level:	High School [✓ Bachelor's] Master's Doctorate

Figure 4–7 Using Checkboxes, Radio Buttons, Menus, and Listboxes

The directory structure for the application shown in Figure 4–7 is shown in Figure 4–8. The JSF pages, `RegisterForm` bean, faces configuration file and resource bundle are listed in Listing 4–11 through Listing 4–15.



```
select
  index.html
  index.jsp
  showInformation.jsp
  styles.css
  WEB-INF
    faces-config.xml
    web.xml
    classes
      com
        corejsf
          RegisterForm.java
          messages.properties
```

Figure 4–8 The Directory Structure

Listing 4–11 select/index.jsp

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

      <body>
         <h:outputText value="#{msgs.indexPageTitle}" styleClass="emphasis"/>
         <h:form>
            <table>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.namePrompt}"/>
                  </td>
                  <td>
                     <h:inputText value="#{form.name}"/>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.contactMePrompt}"/>
                  </td>
                  <td>
                     <h:selectBooleanCheckbox value="#{form.contactMe}"/>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.bestDayPrompt}"/>
                  </td>
                  <td>
                     <h:selectManyMenu value="#{form.bestDaysToContact}">
                        <f:selectItems value="#{form.daysOfTheWeekItems}"/>
                     </h:selectManyMenu>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.yearOfBirthPrompt}"/>
```

Listing 4–11 select/index.jsp (cont.)

```
                  </td>
                  <td>
                     <h:selectOneListbox size="5" value="#{form.yearOfBirth}">
                        <f:selectItems value="#{form.yearItems}"/>
                     </h:selectOneListbox>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.colorPrompt}"/>
                  </td>
                  <td>
                     <h:selectManyCheckbox value="#{form.colors}">
                        <f:selectItems value="#{form.colorItems}"/>
                     </h:selectManyCheckbox>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.languagePrompt}"/>
                  </td>
                  <td>
                     <h:selectManyListbox value="#{form.languages}">
                        <f:selectItems value="#{form.languageItems}"/>
                     </h:selectManyListbox>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.educationPrompt}"/>
                  </td>
                  <td>
                     <h:selectOneRadio value="#{form.education}"
                        layout="pageDirection">
                        <f:selectItems value="#{form.educationItems}"/>
                     </h:selectOneRadio>
                  </td>
               </tr>
            </table>
            <h:commandButton value="#{msgs.buttonPrompt}"
               action="showInformation"/>
         </h:form>
         <h:messages/>
      </body>
   </f:view>
</html>
```

Listing 4–12 select/showInformation.jsp

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <head>
         <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
         <link href="styles.css" rel="stylesheet" type="text/css"/>

         <title>
            <h:outputText value="#{msgs.indexWindowTitle}"/>
         </title>
      </head>

      <body>
         <h:outputFormat value="#{msgs.thankYouLabel}">
            <f:param value="#{form.name}"/>
         </h:outputFormat>
         <p>
         <table>
            <tr>
               <td><h:outputText value="#{msgs.contactMeLabel}"/></td>
               <td><h:outputText value="#{form.contactMe}"/></td>
            </tr>

            <tr>
               <td><h:outputText value="#{msgs.bestDayLabel}"/></td>
               <td><h:outputText value="#{form.bestDaysConcatenated}"/></td>
            </tr>

            <tr>
               <td><h:outputText value="#{msgs.yearOfBirthLabel}"/></td>
               <td><h:outputText value="#{form.yearOfBirth}"/></td>
            </tr>

            <tr>
               <td><h:outputText value="#{msgs.languageLabel}"/></td>
               <td><h:outputText value="#{form.languagesConcatenated}"/></td>
            </tr>

            <tr>
               <td><h:outputText value="#{msgs.colorLabel}"/></td>
               <td><h:outputText value="#{form.colorsConcatenated}"/></td>
            </tr>

```

Listing 4–12 select/showInformation.jsp (cont.)

```
            <tr>
               <td><h:outputText value="#{msgs.educationLabel}"/></td>
               <td><h:outputText value="#{form.education}"/></td>
            </tr>
         </table>
      </body>
   </f:view>
</html>
```

Listing 4–13 select/WEB-INF/com/classes/corejsf/RegisterForm.java

```
package com.corejsf;

import java.text.DateFormatSymbols;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.List;

import javax.faces.model.SelectItem;

public class RegisterForm {
   private String name;
   private boolean contactMe;
   private Integer[] bestDaysToContact;
   private Integer yearOfBirth;
   private String[] colors = null;
   private String[] languages = null;
   private int education;

   // PROPERTY: name
   public String getName() {
      return name;
   }
   public void setName(String newValue) {
      name = newValue;
   }

   // PROPERTY: contactMe
   public boolean getContactMe() {
      return contactMe;
   }
   public void setContactMe(boolean newValue) {
      contactMe = newValue;
   }

```

Listing 4–13 select/WEB-INF/com/classes/corejsf/RegisterForm.java (cont.)

```
   // PROPERTY: bestDaysToContact
   public Integer[] getBestDaysToContact() {
      return bestDaysToContact;
   }
   public void setBestDaysToContact(Integer[] newValue) {
      bestDaysToContact = newValue;
   }

   // PROPERTY: yearOfBirth
   public Integer getYearOfBirth() {
      return yearOfBirth;
   }
   public void setYearOfBirth(Integer newValue) {
      yearOfBirth = newValue;
   }

   // PROPERTY: colors
   public String[] getColors() {
      return colors;
   }
   public void setColors(String[] newValue) {
      colors = newValue;
   }

   // PROPERTY: languages
   public String[] getLanguages() {
      return languages;
   }
   public void setLanguages(String[] newValue) {
      languages = newValue;
   }

   // PROPERTY: education
   public int getEducation() {
      return education;
   }
   public void setEducation(int newValue) {
      education = newValue;
   }

   // PROPERTY: yearItems
   public List getYearItems() {
      return birthYears;
   }

```

Listing 4–13 select/WEB-INF/com/classes/corejsf/RegisterForm.java (cont.)

```
   // PROPERTY: daysOfTheWeekItems
   public SelectItem[] getDaysOfTheWeekItems() {
      return daysOfTheWeek;
   }

   // PROPERTY: languageItems
   public SelectItem[] getLanguageItems() {
      return languageItems;
   }

   // PROPERTY: colorItems
   public SelectItem[] getColorItems() {
      return colorItems;
   }

   // PROPERTY: educationItems
   public SelectItem[] getEducationItems() {
      return educationItems;
   }

   // PROPERTY: bestDaysConcatenated
   public String getBestDaysConcatenated() {
      return concatenate(bestDaysToContact);
   }

   // PROPERTY: languagesConcatenated
   public String getLanguagesConcatenated() {
      return concatenate(languages);
   }

   // PROPERTY: colorsConcatenated
   public String getColorsConcatenated() {
      return concatenate(colors);
   }

   private static String concatenate(Object[] values) {
      if (values == null)
         return "";
      StringBuffer r = new StringBuffer();
      for (int i = 0; i < values.length; ++i) {
         if (i > 0)
            r.append(',');
         r.append(values[i].toString());
      }
```

Listing 4–13 select/WEB-INF/com/classes/corejsf/RegisterForm.java (cont.)

```
      return r.toString();
   }

   private static final int HIGH_SCHOOL = 1;
   private static final int BACHELOR = 2;
   private static final int MASTER = 3;
   private static final int DOCTOR = 4;

   private static SelectItem[] colorItems = new SelectItem[]{
      new SelectItem("Red"),
      new SelectItem("Blue"),
      new SelectItem("Yellow"),
      new SelectItem("Green"),
      new SelectItem("Orange") };

   private static SelectItem[] languageItems = new SelectItem[]{
      new SelectItem("en", "English"),
      new SelectItem("fr", "French"),
      new SelectItem("it", "Italian"),
      new SelectItem("es", "Spanish"),
      new SelectItem("ru", "Russian") };

   private static SelectItem[] educationItems = new SelectItem[]{
      new SelectItem(new Integer(HIGH_SCHOOL), "High School"),
      new SelectItem(new Integer(BACHELOR), "Bachelor's"),
      new SelectItem(new Integer(MASTER), "Master's"),
      new SelectItem(new Integer(DOCTOR), "Doctorate") };

   private static List birthYears;
   private static SelectItem[] daysOfTheWeek;
   static {
      birthYears = new ArrayList();
      for (int i = 1900; i < 2000; ++i) {
         birthYears.add(new SelectItem(new Integer(i)));
      }

      DateFormatSymbols symbols = new DateFormatSymbols();
      String[] weekdays = symbols.getWeekdays();
      daysOfTheWeek = new SelectItem[7];
      for (int i = Calendar.SUNDAY; i <= Calendar.SATURDAY; i++) {
         daysOfTheWeek[i - 1] = new SelectItem(new Integer(i), weekdays[i]);
      }
   }
}
```

Listing 4–14 select/WEB-INF/classes/com/corejsf/messages.properties

```
indexWindowTitle=Checkboxes, Radio buttons, Menus, and Listboxes
indexPageTitle=Please fill out the following information

namePrompt=Name:
contactMePrompt=Contact me
bestDayPrompt=What's the best day to contact you?
yearOfBirthPrompt=What year were you born?
buttonPrompt=Submit information
languagePrompt=Select the languages you speak:
educationPrompt=Select your highest education level:
emailAppPrompt=Select your email application:
colorPrompt=Select your favorite colors:

thankYouLabel=Thank you {0}, for your information
contactMeLabel=Contact me:
bestDayLabel=Best day to contact you:
yearOfBirthLabel=Your year of birth:
colorLabel=Colors:
languageLabel=Languages:
educationLabel=Education:
```

Listing 4–15 select/WEB-INF/faces-config.xml

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

<!DOCTYPE faces-config PUBLIC
   "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
   "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
   <navigation-rule>
      <navigation-case>
         <from-outcome>showInformation</from-outcome>
         <to-view-id>/showInformation.jsp</to-view-id>
      </navigation-case>
   </navigation-rule>

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

Messages

During the JSF life cycle, any object can create a message and add it to a queue of messages maintained by the faces context. At the end of the life cycle—in the render response phase—you can display those messages in a view. Typically, messages are associated with a particular component and indicate either conversion or validation errors.

Although error messages are usually the most prevalent message type in a JSF application, messages come in four varieties:

- Information
- Warning
- Error
- Fatal

All messages can contain a summary and a detail. For example, a summary might be `Invalid Entry` and a detail might be `The number entered was greater than the maximum`.

JSF applications use two tags to display messages in JSF pages: `h:messages` and `h:message`.

The `h:messages` tag displays all messages that were stored in the faces context during the course of the JSF life cycle. You can restrict those messages to global messages—meaning messages not associated with a component—by setting `h:message`'s `globalOnly` attribute to `true`. By default, that attribute is `false`.

The `h:message` tag displays a single message for a particular component. That component is designated with `h:message`'s mandatory `for` attribute. If more than one message has been stored in the Faces context for a component, `h:message` shows only the last message that was created.

`h:message` and `h:messages` share many attributes. Table 4–23 lists all attributes for both tags.

Table 4–23 Attributes for `h:message` and `h:messages`

Attributes	Description
`errorClass`	CSS class applied to error messages
`errorStyle`	CSS style applied to error messages
`fatalClass`	CSS class applied to fatal messages
`fatalStyle`	CSS style applied to fatal messages

Table 4–23 Attributes for `h:message` and `h:messages` (cont.)

Attributes	Description
`globalOnly`	Instruction to display only global messages—applicable only to `h:messages`. Default: `false`
`infoClass`	CSS class applied to information messages
`infoStyle`	CSS style applied to information messages
`layout`	Specification for message layout: `table` or `list`—applicable only to `h:messages`
`showDetail`	A boolean that determines whether message details are shown. Defaults are `false` for `h:messages`, `true` for `h:message`.
`showSummary`	A boolean that determines whether message summaries are shown. Defaults are `true` for `h:messages`, `false` for `h:message`.
`tooltip`	A boolean that determines whether message details are rendered in a tooltip; the tooltip is only rendered if `showDetail` and `showSummary` are `true`
`warnClass`	CSS class for warning messages
`warnStyle`	CSS style for warning messages
`binding, id, rendered, styleClass`	Basic attributes[a]
`style, title`	HTML 4.0[b]

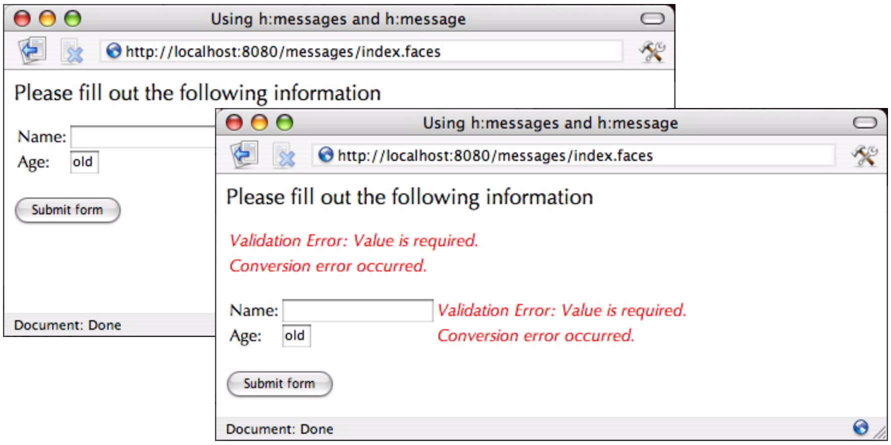
a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.

The majority of the attributes in Table 4–23 represent CSS classes or styles that `h:message` and `h:messages` apply to particular types of messages.

You can also specify whether you want to display a message's summary or detail, or both, with the `showSummary` and `showDetail` attributes, respectively.

The `h:messages layout` attribute can be used to specify how messages are laid out, either as a list or a table. If you specify `true` for the `tooltip` attribute and you've also set `showDetail` and `showSummary` to `true`, the message's detail will be wrapped in a tooltip that's shown when the mouse hovers over the error message.

Now that we have a grasp of message fundamentals, let's take a look at an application that uses the `h:message` and `h:messages` tags. The application shown in Figure 4–9 contains a simple form with two text fields. Both text fields have required attributes. Moreover, the Age text field is wired to an integer property, so it's value is converted automatically by the JSF framework. Figure 4–9 shows the error messages generated by the JSF framework when we neglect to specify a value for the Name field and provide the wrong type of value for the Age field.


Using h:messages and h:message
http://localhost:8080/messages/index.faces
Please fill out the following information
Name:
Age:
old
Submit form
Document: Done

Validation Error: Value is required.
Conversion error occurred.

Name: Validation Error: Value is required.
Age: old Conversion error occurred.

Figure 4–9 Displaying Messages

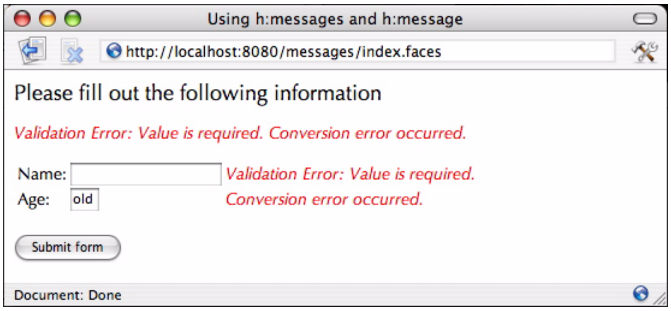
At the top of the JSF page we use h:messages to display all messages. We use h:message to display messages for each input field.

```
<h:form>
    <h:messages layout="table" errorClass="errors"/>
    ...
    <h:inputText id="name" required="true"/>
    <h:message for="name" errorClass="errors"/>
    ...
    <h:inputText id="date" value="#{form.date}" required="true"/>
    <h:message for="date" errorClass="errors"/>
    ...
</form>
```

Both tags in our example specify a CSS class named errors, which is defined in styles.css. That class definition looks like this:

```
.errors {
   font-style: italic;
}
```

We've also specified layout="table" for the h:messages tag. If we had omitted that attribute (or alternatively specified layout="list"), the output would look like this:


Using h:messages and h:message
http://localhost:8080/messages/index.faces
Submit form
Document: Done

Please fill out the following information

Validation Error: Value is required. Conversion error occurred.

Name: Validation Error: Value is required.
Age: old Conversion error occurred.

The list layout encodes the error messages, one after the other, as text, whereas the table layout puts the messages in an HTML table element, one row per message. Most of the time you will probably prefer the table layout (even though it's not the default) because it's easier to read.

Figure 4–10 shows the directory structure for the application shown in Figure 4–9. Listing 4–16 through Listing 4–18 list the JSP page, resource bundle, and stylesheet for the application.

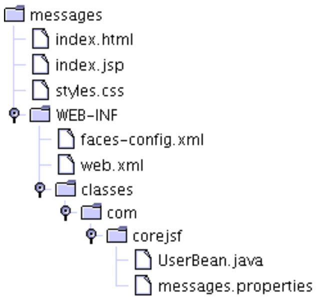


Figure 4–10 Directory Structure for the Messages Example

Listing 4–16 `messages/index.jsp`

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <f:loadBundle basename="com.corejsf.messages" var="msgs"/>
      <head>
         <link href="styles.css" rel="stylesheet" type="text/css"/>
```

Listing 4–16 messages/index.jsp (cont.)

```
         <title>
            <h:outputText value="#{msgs.windowTitle}"/>
         </title>
      </head>
      <body>
         <h:form>
            <h:outputText value="#{msgs.greeting}" styleClass="emphasis"/>
            <br/>
            <h:messages errorClass="errors"/>
            <br/>
            <table>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.namePrompt}"/>
                  </td>
                  <td>
                     <h:inputText id="name"
                        value="#{user.name}" required="true"/>
                  </td>
                  <td>
                     <h:message for="name" errorClass="errors"/>
                  </td>
               </tr>
               <tr>
                  <td>
                     <h:outputText value="#{msgs.agePrompt}"/>
                  </td>
                  <td>
                     <h:inputText id="age"
                        value="#{user.age}" required="true" size="3"/>
                  </td>
                  <td>
                     <h:message for="age" errorClass="errors"/>
                  </td>
               </tr>
            </table>
            <br/>
            <h:commandButton value="#{msgs.submitPrompt}"/>
         </h:form>
      </body>
   </f:view>
</html>
```

Listing 4–17 messages/WEB-INF/classes/com/corejsf/messages.properties

```
windowTitle=Using h:messages and h:message
greeting=Please fill out the following information
namePrompt=Name:
agePrompt=Age:
submitPrompt=Submit form
```

Listing 4–18 messages/styles.css

```
.errors {
   font-style: italic;
}
.emphasis {
   font-size: 1.3em;
}
```

NOTE: By default, h:messages shows message summaries but not details. h:message, on the other hand, shows details but not summaries. If you use h:messages and h:message together, as we did in the preceding example, summaries will appear at the top of the page, with details next to the appropriate input field.

Panels

Throughout this chapter we've used HTML tables to align form prompts and input fields. Creating table markup by hand is tedious and error prone, so let's see how we can alleviate some of that tediousness with `h:panelGrid`, which generates an HTML `table` element. You can specify the number of columns in the table with the `columns` attribute, like this:

```
<h:panelGrid columns="3">
    ...
</h:panelGrid>
```

The `columns` attribute is not mandatory—if you don't specify it, the number of columns defaults to 1. `h:panelGrid` places components in columns from left to right and top to bottom. For example, if you have a panel grid with three columns and nine components, you'll wind up with three rows, each containing three columns. If you specify three columns and ten components, you'll have four rows and in the last row only the first column will contain a component—the tenth component.

Table 4–24 lists `h:panelGrid` attributes.

Table 4–24 Attributes for `h:panelGrid`

Attributes	Description
`bgcolor`	Background color for the table
`border`	Width of the table's border
`cellpadding`	Padding around table cells
`cellspacing`	Spacing between table cells
`columnClasses`	Comma-separated list of CSS classes for columns
`columns`	Number of columns in the table
`footerClass`	CSS class for the table footer
`frame`	Specification for sides of the frame surrounding the table that are to be drawn; valid values: `none`, `above`, `below`, `hsides`, `vsides`, `lhs`, `rhs`, `box`, `border`
`headerClass`	CSS class for the table header
`rowClasses`	Comma-separated list of CSS classes for columns
`rules`	Specification for lines drawn between cells; valid values: `groups`, `rows`, `columns`, `all`
`summary`	Summary of the table's purpose and structure used for non-visual feedback such as speech
`binding, id, rendered, style-Class, value`	Basic attributes[a]
`dir, lang, style, title, width`	HTML 4.0[b]
`onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmouse-down, onmousemove, onmouseout, onmouseover, onmouseup`	DHTML events[c]

a. See Table 4–3 on page 90 for information about basic attributes.
b. See Table 4–4 on page 93 for information about HTML 4.0 attributes.
c. See Table 4–5 on page 95 for information about DHTML event attributes.

You can specify CSS classes for different parts of the table: header, footer, rows, and columns. The `columnClasses` and `rowClasses` specify lists of CSS classes that are applied to columns and rows, respectively. If those lists contain fewer class names than rows or columns, the CSS classes are reused. That makes it possible to specify classes like this: `rowClasses="evenRows, oddRows"` and `columnClasses="evenColumns, oddColumns"`.

The `cellpadding`, `cellspacing`, `frame`, `rules`, and `summary` attributes are HTML pass-through attributes that apply only to tables. See the HTML 4.0 specification for more information.

`h:panelGrid` is often used in conjunction with `h:panelGroup`, which groups two or more components so they are treated as one. For example, you might group an input field and it's error message, like this:

```
<h:panelGrid columns="2">
    ...
    <h:panelGroup>
        <h:inputText id="name" value="#{user.name}">
        <h:message for="name"/>
    </h:panelGroup>
    ...
</h:panelGrid>
```

Grouping the text field and error message puts them in the same table cell. Without that grouping, the error message component would occupy its own cell. In the absence of an error message, the error message component produces no output, but still takes up a cell, so you wind up with an empty cell.

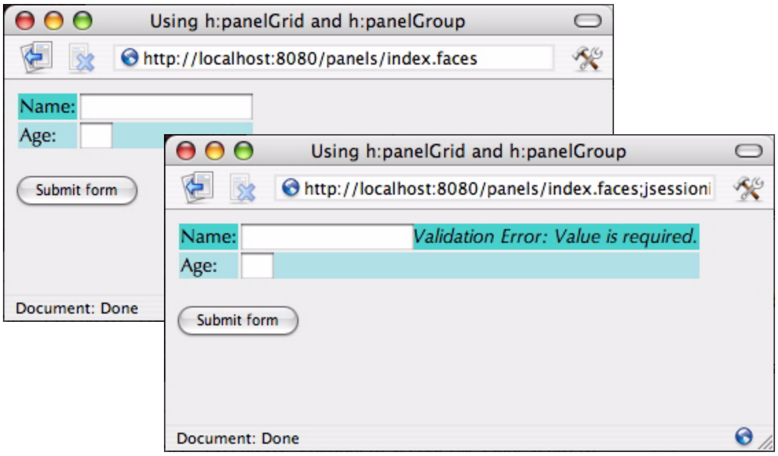
`h:panelGroup` is a simple tag with only a handful of attributes. Those attributes are listed in Table 4–25.

Table 4–25 Attributes for `h:panelGroup`

Attributes	Description
`binding, id, rendered, styleClass`	`binding, id, rendered, styleClass, value`
`style`	HTML 4.0[a]

a. See Table 4–4 on page 93 for information about HTML 4.0 attributes.

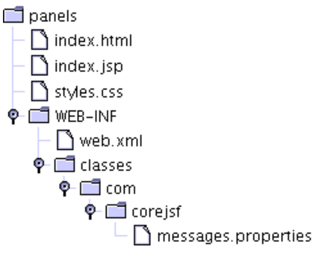
Figure 4–11 shows a simple example that uses `h:panelGrid` and `h:panelGroup`. The application contains a form that asks for the user's name and age. We've added a required validator—with `h:inputText`'s `required` attribute—to the name field and used an `h:message` tag to display the corresponding error when that constraint is violated. We've placed no restrictions on the `Age` text field. Notice that those constraints require three columns in the first row—one each for the name prompt, text field, and error message—but only two in the second: the age prompt and text field. Since `h:panelGrid` allows only one value for its `columns` attribute, we can resolve this column quandary by placing the name text field and error message in a panel group, and because those two components will be treated as one, we actually have only two columns in each row.


Using h:panelGrid and h:panelGroup
http://localhost:8080/panels/index.faces
Name:
Age:
Submit form
Document: Done
Using h:panelGrid and h:panelGroup
http://localhost:8080/panels/index.faces;jsessioni
Name:
Age:
Submit form
Document: Done

Validation Error: Value is required.

Figure 4–11 Using `h:panelGrid` and `h:panelGroup`

The directory structure for the application shown in Figure 4–11 is shown in Figure 4–12. Listings 4–19 through 4–21 contain the code of the JSF page and related files.



```
panels
  index.html
  index.jsp
  styles.css
  WEB-INF
    web.xml
    classes
      com
        corejsf
          messages.properties
```

Figure 4–12 Directory Structure for the Panels Example

Listing 4–19 panels/index.jsp

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

      <body>
         <h:form>
            <h:panelGrid columns="2" rowClasses="oddRows,evenRows">
               <h:outputText value="#{msgs.namePrompt}"/>
               <h:panelGroup>
                  <h:inputText id="name" required="true"/>
                  <h:message for="name" errorClass="errors"/>
               </h:panelGroup>
               <h:outputText value="#{msgs.agePrompt}"/>
               <h:inputText size="3"/>
            </h:panelGrid>
            <br/>
            <h:commandButton value="#{msgs.submitPrompt}"/>
         </h:form>
      </body>
   </f:view>
</html>
```

Listing 4–20 panels/WEB-INF/classes/com/corejsf/messages.properties

```
windowTitle=Using h:panelGrid and h:panelGroup
namePrompt=Name:
agePrompt=Age:
submitPrompt=Submit form
```

Listing 4–21 panels/styles.css

```css
body {
   background: #eee;
}
.errors {
   font-style: italic;
}
.evenRows {
   background: PowderBlue;
}
.oddRows {
   background: MediumTurquoise;
}
```