DAVID GEARY • CAY HORSTMANN

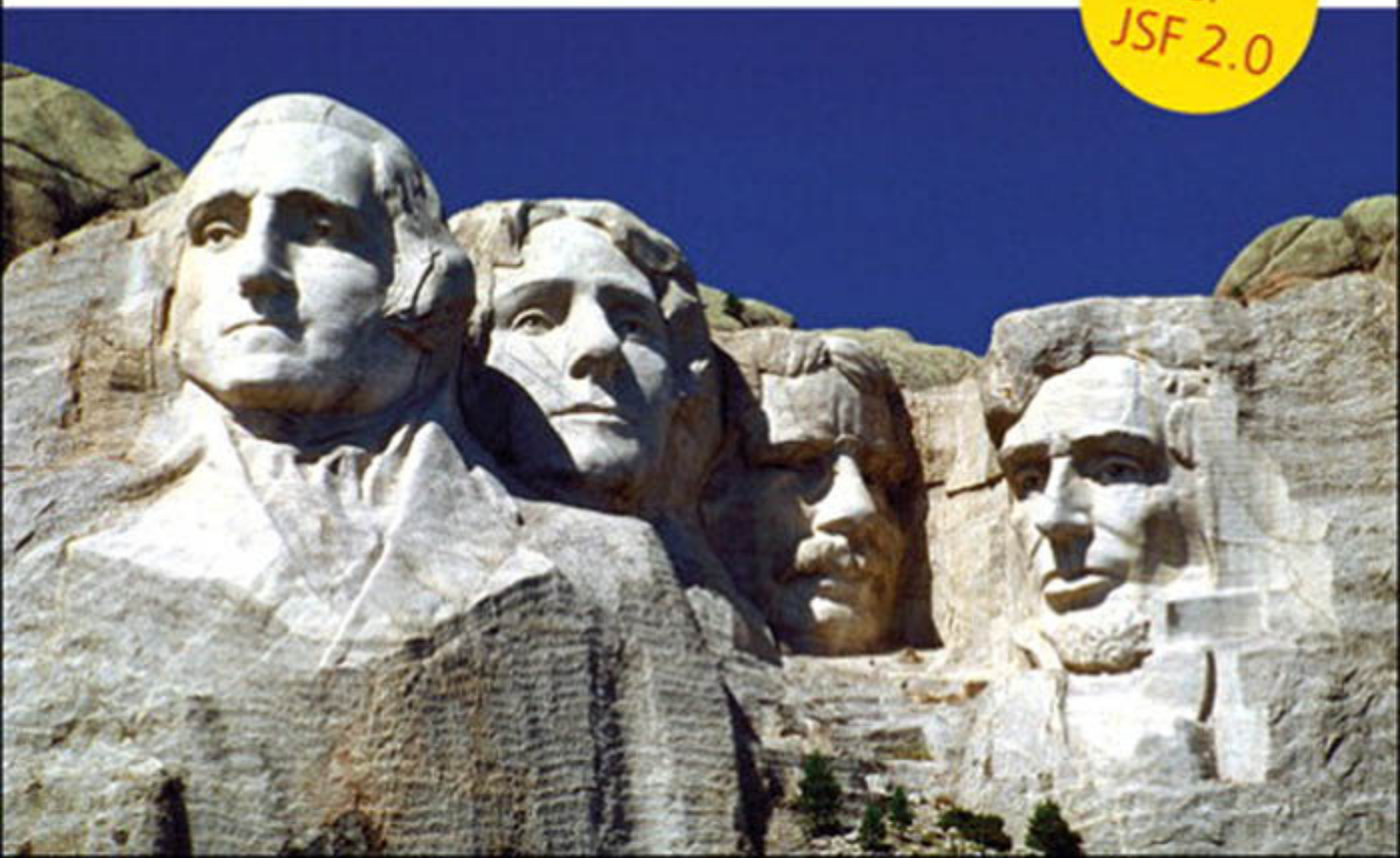
PRENTICE HALL

Core JavaServer™ Faces

THIRD EDITION

Java™

Updated for JSF 2.0



Library of Congress Cataloging-in-Publication Data

Geary, David M.
Core JavaServer faces / David Geary, Cay Horstmann.—3rd ed.
p. cm.
Includes index.
ISBN 978-0-13-701289-3 (pbk. : alk. paper)
1. JavaServer pages. 2. Web site development. 3. Web sites—Design.
I. Horstmann, Cay S., 1959- II. Title.
TK5105.8885.J38G433 2010
006.7'8—dc22

2010011569

ISBN-13: 978-0-13-701289-3
ISBN-10: 0-13-701289-6

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, May 2010

Preface

When we heard about JavaServer Faces (JSF) at the 2002 JavaOne conference, we were very excited. Both of us had extensive experience with client-side Java programming—David in *Graphic Java*™, and Cay in *Core Java*™, both published by Sun Microsystems Press—and we found web programming with servlets and JavaServer Pages (JSP) to be rather unintuitive and tedious. JSF promised to put a friendly face in front of a web application, allowing programmers to think about text fields and menus instead of dealing with page flips and request parameters. Each of us proposed a book project to our publisher, who promptly suggested that we should jointly write the Sun Microsystems Press book on JSF.

In 2004, the JSF Expert Group (of which David is a member) released the JSF 1.0 specification and reference implementation. A bug fix 1.1 release emerged shortly afterward, and an incremental 1.2 release added a number of cleanups and convenience features in 2006.

The original JSF specification was far from ideal. It was excessively general, providing for use cases that turned out to be uninteresting in practice. Not enough attention was given to API design, forcing programmers to write complex and tedious code. Support for GET requests was clumsy. Error handling was plainly unsatisfactory, and developers cursed the "stack trace from hell".

JSF had one saving grace, however. It was highly extensible, and therefore it was very attractive to framework developers. Those framework developers

built cutting edge open-source software that plugged into JSF, such as Facelets, Ajax4jsf, Seam, JSF Templates, Pretty Faces, RichFaces, ICEFaces, and so on.

JSF 2.0, released in 2009, is built on the experience of those open-source frameworks. Nearly all of the original authors of the aforementioned frameworks participated on the JSF 2 Expert Group, so JSF 2.0, unlike JSF 1.0, was forged from the crucible of real-world open-source projects that had time to mature.

JSF 2.0 is *much* simpler to use and better integrated into the Java EE technology stack than JSF 1.0. Almost every inch of JSF 1.0 has been transformed in JSF 2.0 in some way for the better. In addition, the specification now supports new web technologies such as Ajax and REST.

JSF is now the preeminent server-side Java web framework, and it has fulfilled most of its promises. You really can design web user interfaces by putting components on a form and linking them to Java objects, without having to mix code and markup. A strong point of JSF is its extensible component model, and a large number of third-party components have become available. The flexible design of the framework has allowed it to grow well and accommodate new technologies.

Because JSF is a specification and not a product, you are not at the mercy of a single vendor. JSF implementations, components, and tools are available from multiple sources. We are very excited about JSF 2.0, and we hope you will share in this excitement when you learn how this technology makes you a more effective web application developer.

About This Book

This book is suitable for web developers whose main focus is on implementing user interfaces and business logic. This is in stark contrast to the official JSF specification, a dense and pompously worded document whose principal audience is framework implementors, as well as long-suffering book authors. JSF is built on top of servlets, but from the point of view of the JSF developer, this technology merely forms the low-level plumbing. While it can't hurt to be familiar with servlets, JSP, or Struts, we do not assume any such knowledge.

The first half of the book, extending through Chapter 7, focuses on the JSF *tags*. These tags are similar to HTML form tags. They are the basic building blocks for JSF user interfaces. Anyone with basic HTML skills (for web page design) and standard Java programming (for the application logic) can use the JSF tags to build web applications.

The first part of the book covers these topics:

- Setting up your programming environment (Chapter 1)
- Connecting JSF tags to application logic (Chapter 2)
- Navigating between pages (Chapter 3)
- Using the standard JSF tags (Chapter 4)
- Using Facelets tags for templating (Chapter 5) **NEW**
- Data tables (Chapter 6)
- Converting and validating input (Chapter 7)

Starting with Chapter 8, we begin JSF programming in earnest. You will learn how to perform advanced tasks, and how to extend the JSF framework. Here are the main topics of the second part:

- Event handling (Chapter 8)
- Building composite components—reusable components with sophisticated behavior that are composed from simpler components (Chapter 9) **NEW**
- Ajax (Chapter 10) **NEW**
- Implementing custom components (Chapter 11)
- Connecting to databases and other external services (Chapter 12)

We end the book with a chapter that aims to answer common questions of the form "How do I . . . ?" (Chapter 13). We encourage you to have a peek at that chapter as soon as you become comfortable with the basics of JSF. There are helpful notes on debugging and logging, and we also give you implementation details and working code for features that are missing from JSF, such as file uploads, pop-up menus, and a pager component for long tables.

All chapters have been revised extensively in this edition to stress the new and improved features of JSF 2.0. Chapters 5, 9, and 10 are new to this edition.

Required Software

All software that you need for this book is freely available. You can use an application server that supports Java EE 6 (such as GlassFish version 3) or a servlet runner (such as Tomcat 6) together with a JSF implementation. The software runs on Linux, Mac OS X, Solaris, and Windows. Both Eclipse and NetBeans have extensive support for JSF development with GlassFish or Tomcat.

Web Support

The web site for this book is `http://corejsf.com`. It contains:

- The source code for all examples in this book
- Useful reference material that we felt is more effective in browseable form than in print
- A list of known errors in the book and the code
- A form for submitting corrections and suggestions

Standard JSF Tags

Topics in This Chapter

- "An Overview of the JSF Core Tags" on page 102
- "An Overview of the JSF HTML Tags" on page 105
- "Panels" on page 115
- "The Head, Body, and Form Tags" on page 118
- "Text Fields and Text Areas" on page 123
- "Buttons and Links" on page 134
- "Selection Tags" on page 145
- "Messages" on page 171

Chapter 4

Development of compelling JSF applications requires a good grasp of the JSF tag libraries. JSF 1.2 had two tag libraries: core and HTML. As of JSF 2.0, there are six libraries with over 100 tags—see Table 4–1. In this chapter, we cover the core library and most of the HTML library. One HTML library component—the data table—is so complex that it is covered separately in Chapter 6.

Table 4–1 JSF Tag Libraries

Library	Namespace Identifier	Commonly Used Prefix	Number of Tags	See Chapter
Core	http://java.sun.com/jsf/core	f:	27	See Table 4–2
HTML	http://java.sun.com/jsf/html	h:	31	4 and 6
Facelets JSF 2.0	http://java.sun.com/jsf/facelets	ui:	11	5
Composite Components JSF 2.0	http://java.sun.com/jsf/composite	composite:	12	9
JSTL Core JSF 2.0	http://java.sun.com/jsp/jstl/core	c:	7	13
JSTL Functions JSF 2.0	http://java.sun.com/jsp/jstl/functions	fn:	16	2

An Overview of the JSF Core Tags

The core library contains the tags that are independent of HTML rendering. The core tags are listed in Table 4–2.

Table 4–2 JSF Core Tags

Tag	Description	See Chapter
attribute	Sets an attribute (key/value) in its parent component.	4
param	Adds a parameter child component to its parent component.	4
facet	Adds a facet to a component.	4
actionListener	Adds an action listener to a component.	8
setPropertyActionListener JSF 1.2	Adds an action listener that sets a property.	8
valueChangeListener	Adds a value change listener to a component.	8
phaseListener JSF 1.2	Adds a phase listener to the parent view.	8
event JSF 2.0	Adds a component system event listener.	8
converter	Adds an arbitrary converter to a component.	7
convertDateTime	Adds a datetime converter to a component.	7
convertNumber	Adds a number converter to a component.	7
validator	Adds a validator to a component.	7
validateDoubleRange	Validates a double range for a component's value.	7
validateLength	Validates the length of a component's value.	7
validateLongRange	Validates a long range for a component's value.	7
validateRequired JSF 2.0	Checks that a value is present.	7
validateRegex JSF 2.0	Validates a value against a regular expression.	7

Table 4–2 JSF Core Tags (cont.)

Tag	Description	See Chapter
validateBean JSF 2.0	Uses the Bean Validation API (JSR 303) for validation.	7
loadBundle	Loads a resource bundle, stores properties as a Map.	2
selectitems	Specifies items for a select one or select many component.	4
selectitem	Specifies an item for a select one or select many component.	4
verbatim	Turns text containing markup into a component.	4
viewParam JSF 2.0	Defines a "view parameter" that can be initialized with a request parameter.	3
metadata JSF 2.0	Holds view parameters. May hold other metadata in the future.	3
ajax JSF 2.0	Enables Ajax behavior for components.	11
view	Use for specifying the page locale or a phase listener.	2 and 7
subview	Not needed with facelets.	

Most of the core tags represent objects you add to components, such as the following:

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

All of the core tags are discussed at length in different places in this book, as shown in Table 4–1.

Attributes, Parameters, and Facets

The f:attribute, f:param, and f:facet tags are general-purpose tags to add information to a component. Any component can store arbitrary name/value pairs in its *attribute map*. You can set an attribute in a page and later retrieve it programatically. For example, in "Supplying Attributes to Converters" on page 289 of Chapter 7, we set the separator character for credit card digit groups like this:

```
<h:outputText value="#{payment.card}">
   <f:attribute name="separator" value="-" />
</h:outputText>
```

The converter that formats the output retrieves the attribute from the component.

The f:param tag also lets you define a name/value pair, but the value is placed *in a separate child component*, a much bulkier storage mechanism. However, the child components form a list, not a map. You use f:param if you need to supply a number of values with the same name (or no name at all). You saw an example in "Messages with Variable Parts" on page 42 of Chapter 2, where the h:outputFormat component contains a list of f:param children.

NOTE: the h:commandlink component turns its f:param children into HTTP request name/value pairs. The event listener that is activated when the user clicks the link can then retrieve the name/value pairs from the request map. We demonstrate this technique in Chapter 8.

Finally, f:facet adds a named component to a component's *facet map*. A facet is not a child component; each component has *both* a list of child components and a map of named facet components. The facet components are usually rendered in a special place. The root of a Facelets page has two facets named "head" and "body". You will see in "Headers, Footers, and Captions" on page 212 of Chapter 6 how to use facets named "header" and "footer" in data tables.

Table 4–3 shows the attributes for the f:attribute, f:param, and f:facet tags.

Table 4–3 Attributes for f:attribute, f:param, and f:facet

Attribute	Description
name	The attribute, parameter component, or facet name
value	The attribute or parameter component value (does not apply to f:facet)
binding, id	See Table 4–5 on page 107 (f:param only)

NOTE: All tag attributes in this chapter, except for `var` and `id`, accept value or method expressions. The `var` attribute must be a string. The `id` attribute can be a string or an immediate `${...}` expression.

An Overview of the JSF HTML Tags

Table 4–4 lists all HTML tags. We can group these tags in the following categories:

- Inputs (`input...`)
- Outputs (`output...`, `graphicImage`)
- Commands (`commandButton` and `commandLink`)
- GET Requests (`button`, `link`, `outputLink`)
- Selections (`checkbox`, `listbox`, `menu`, `radio`)
- HTML pages (`head`, `body`, `form`, `outputStylesheet`, `outputScript`)
- Layouts (`panelGrid`, `panelGroup`)
- Data table (`dataTable` and `column`); see Chapter 6
- Errors and messages (`message`, `messages`)

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

Table 4–4 JSF HTML Tags

Tag	Description
`head` JSF 2.0	Renders the head of the page
`body` JSF 2.0	Renders the body of the page
`form`	Renders a HTML form
`outputStylesheet` JSF 2.0	Adds a stylesheet to the page
`outputScript` JSF 2.0	Adds a script to the page
`inputText`	Single-line text input control
`inputTextarea`	Multiline text input control
`inputSecret`	Password input control
`inputHidden`	Hidden field

Table 4–4 JSF HTML Tags (cont.)

Tag	Description
`outputLabel`	Label for another component for accessibility
`outputLink`	Link to another web site
`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
`button` JSF 2.0	Button for issuing a GET request
`link` JSF 2.0	Link for issuing a GET request
`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`	Tabular layout
`panelGroup`	Two or more components that are laid out as one
`dataTable`	A feature-rich table control (see Chapter 6)
`column`	Column in a `dataTable` (see Chapter 6)

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.

Common Attributes

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

- Basic
- HTML 4.0
- DHTML events

Next, we look at each type.

Basic Attributes

As you can see from Table 4–5, basic attributes are shared by the majority of JSF HTML tags.

Table 4–5 Basic HTML Tag Attributes[a]

Attribute	Component Types	Description
`id`	A (*31*)	Identifier for a component
`binding`	A (*31*)	Links this component with a backing bean property
`rendered`	A (*31*)	A Boolean; `false` suppresses rendering
`value`	I, O, C (*21*)	A component's value, typically a value expression
`valueChangeListener`	I (*11*)	A method expression to a method that responds to value changes
`converter`	I, O (*15*)	Converter class name
`validator`	I (*11*)	Class name of a validator that is created and attached to a component

Table 4–5 Basic HTML Tag Attributes[a] (cont.)

Attribute	Component Types	Description
required	I (*11*)	A Boolean; if true, requires a value to be entered in the associated field
converterMessage, validatorMessage, requiredMessage JSF 1.2	I (*11*)	A custom message to be displayed when a conversion or validation error occurs, or when required input is missing

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

All components can have id, binding, and rendered attributes, which we discuss in the following sections.

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 7 for more information about validators and converters.

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 120 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. There is another 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:inputText binding="#{form.nameField}" .../>
```

The `binding` attribute is specified with a value expression. That expression refers to a read-write bean property, such as this one:

```
private UIComponent nameField = new UIInput();
public UIComponent getNameField() { return nameField; }
public void setNameField(UIComponent newValue) { nameField = newValue; }
```

See "Backing Beans" on page 38 of Chapter 2 for more information about the `binding` attribute. The JSF implementation sets the property to the component, so you can programatically manipulate components.

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:commandButton value="Logout" .../>
```

Most of the time you will use a value expression—for example:

```
<h:inputText 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 7.

Conditional Rendering

You use the rendered attribute to include or exclude a component, depending on a condition. For example, you may want to render a "Logout" button only if the user is currently logged in:

```
<h:commandButton ... rendered="#{user.loggedIn}"/>
```

To conditionally include a group of components, include them in an `h:panelGrid` with a `rendered` attribute. See "Panels" on page 115 for more information.

TIP: Remember, you can use operators in value 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 user selects the `Movies` tab.

NOTE: Sometimes, you will see the JSTL `c:if` construct used for conditional rendering. However, that is less efficient than the `rendered` attribute.

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 4–6.

Table 4–6 HTML 4.0 Pass-Through Attributes[a]

Attribute	Description
`accesskey` (*16*)	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.
`acceptcharset` (*1*)	Comma- or space-separated list of character encodings for a form. The HTML `accept-charset` attribute is specified with the JSF attribute named `acceptcharset`.
`alt` (*5*)	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` (*3*)	Coordinates for an element whose shape is a rectangle, circle, or polygon.
`dir` (*26*)	Direction for text. Valid values are `"ltr"` (left to right) and `"rtl"` (right to left).

Table 4–6 HTML 4.0 Pass-Through Attributes[a] (cont.)

Attribute	Description
`disabled` (*14*)	Disabled state of an input element or button.
`hreflang` (*3*)	Base language of a resource specified with the `href` attribute; `hreflang` may only be used with `href`.
`lang` (*26*)	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` (*3*)	Relationship between the current document and a link specified with the `href` attribute.
`rev` (*3*)	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 is not an HTML pass-through attribute.
`shape` (*3*)	Shape of a region. Valid values: `default`, `rect`, `circle`, `poly` (`default` signifies the entire region).
`size` (*4*)	Size of an input field.
`style` (*26*)	Inline style information.
`styleClass` (*26*)	Style class; rendered as HTML `class` attribute.
`tabindex` (*16*)	Numerical value specifying a tab index.
`target` (*5*)	The name of a frame in which a document is opened.
`title` (*25*)	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 4–6 are defined in the HTML specification, which you can access online at `http://www.w3.org/TR/REC-html40`. That web site is an excellent resource for deep digging into HTML.

Styles

You can use CSS styles, either inline (`style`) or classes (`styleClass`), to influence how components are rendered:

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

CSS style attributes can be value expressions—that gives you programmatic control over styles.

Resources JSF 2.0

You can include a stylesheet in the usual way, with an HTML `link` tag. But that is tedious if your pages are at varying directory nesting levels—you would always need to update the stylesheet directory when you move a page. More importantly, if you assemble pages from different pieces—as described in Chapter 5—you don't even know where your pieces end up.

Since JSF 2.0, there is a better way. You can place stylesheets, JavaScript files, images, and other files into a `resources` directory in the root of your web application. Subdirectories of this directory are called *libraries*. You can create any libraries that you like. In this book, we often use libraries `css`, `images`, and `javascript`.

To include a stylesheet, use the tag:

```
<h:outputStylesheet library="css" name="styles.css"/>
```

The tag adds a link of the form

```
<link href="/context-root/faces/javax.faces.resource/styles.css?ln=css"
   rel="stylesheet" type="text/css"/>
```

to the header of the page.

To include a script resource, use the `outputScript` tag instead:

```
<h:outputScript name="jsf.js" library="javascript" target="head" />
```

If the `target` attribute is `head` or `body`, the script is appended to the `"head"` or `"body"` facet of the root component, which means that it appears at the end of the `head` or `body` in the generated HTML. If there is no `target` element, the script is inserted in the current location.

To include an image from a library, you use the `graphicImage` tag:

```
<h:graphicImage name="logo.png" library="images"/>
```

There is a *versioning* mechanism for resource libraries and individual resources. You can add subdirectories to the library directory and place newer versions of

files into them. The subdirectory names are simply the version numbers. For example, suppose you have the following directories:

```
resources/css/1_0_2
resources/css/1_1
```

Then the latest version (`resources/css/1_1`) will be used. Note that you can add new versions of a library in a running application.

Similarly, you can add new versions of an individual resource, but the naming scheme is a bit odd. You replace the resource with a directory of the same name, then use the version name as the file name. You can add an extension if you like. For example:

```
resources/css/styles.css/1_0_2.css
resources/css/styles.css/1_1.css
```

The version numbers must consist of decimal numbers, separated by underscores. They are compared in the usual way, first comparing the major version numbers and using the minor numbers to break ties.

There is also a mechanism for supplying localized versions of resources. Unfortunately, that mechanism is unintuitive and not very useful. Localized resources have a prefix, such as `resources/de_DE/images`, but the prefix is *not* treated in the same way as a bundle suffix. There is no fallback mechanism. That is, if an image is not found in `resources/de_DE/images`, then `resources/de/images` and `resources/images` are *not* consulted.

Moreover, the locale prefix is *not* simply the current locale. Instead, it is obtained by a curious lookup, which you enable by following these steps:

1. Add the line

   `<message-bundle>`*name of a resource bundle used in your application*`</message-bundle>`

   inside the `application` element of `faces-config.xml`
2. Inside each localized version of that resource bundle, place a name/value pair

   `javax.faces.resource.localePrefix=`*prefix*
3. Place the matching resources into `resources/`*prefix*`/`*library*`/...`

For example, if you use the message bundle `com.corejsf.messages`, and the file `com.corejsf.messages_de` contains the entry

```
javax.faces.resource.localePrefix=german
```

then you place the German resources into `resources/german`. (The prefix need not use the standard language and country codes, and in fact it is a good idea not to use them so that you don't raise false hopes.)

CAUTION: Unfortunately, this localization scheme is unappealing in practice. Once you define a locale prefix, that prefix is used for *all* resources. Suppose you wanted to have different images for the German and English versions of your site. Then you would also have to duplicate *every other* resource. Hopefully, this will be fixed in a future version of JSF.

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 called *dynamic HTML (DHTML) event attributes*. JSF supports DHTML event attributes for nearly all of the JSF HTML tags. Those attributes are listed in Table 4–7.

Table 4–7 DHTML Event Attributes[a]

Attribute	Description
`onblur` (*16*)	Element loses focus
`onchange` (*11*)	Element's value changes
`onclick` (*17*)	Mouse button is clicked over the element
`ondblclick` (*21*)	Mouse button is double-clicked over the element
`onfocus` (*16*)	Element receives focus
`onkeydown` (*21*)	Key is pressed
`onkeypress` (*21*)	Key is pressed and subsequently released
`onkeyup` (*21*)	Key is released
`onload` (*1*)	Page is loaded
`onmousedown` (*21*)	Mouse button is pressed over the element
`onmousemove` (*21*)	Mouse moves over the element
`onmouseout` (*21*)	Mouse leaves the element's area
`onmouseover` (*21*)	Mouse moves onto an element
`onmouseup` (*21*)	Mouse button is released
`onreset` (*1*)	Form is reset

Table 4–7 DHTML Event Attributes[a] (cont.)

Attribute	Description
onselect (*11*)	Text is selected in an input field
onsubmit (*1*)	Form is submitted
onunload (*1*)	Page is unloaded

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

The DHTML event attributes listed in Table 4–7 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 will 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()"...>`

Panels

Up to this point, we have used HTML tables to lay out components. Creating table markup by hand is tedious, so now we'll look at alleviating some of that tedium with `h:panelGrid`, which generates the HTML markup for laying out components in rows and columns.

NOTE: The `h:panelGrid` tag uses HTML tables for layout, which some web designers find objectionable. You can certainly use CSS layout instead of `h:panelGrid`. A future version of `h:panelGrid` may have an option for using CSS layout as well.

You can specify the number of columns with the `columns` attribute, like this:

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

The `columns` attribute is not mandatory—if you do not specify it, the number of columns defaults to 1. The `h:panelGrid` tag 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 will wind up with three rows, each containing three columns. If you specify three columns and 10 components, you will have four rows, and in the last row only the first column will contain the tenth component.

Table 4–8 lists `h:panelGrid` attributes.

Table 4–8 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 rows
`rules`	Specification for lines drawn between cells; valid values: `groups`, `rows`, `columns`, `all`
`summary`	Summary of the table's purpose and structure used for nonvisual feedback, such as speech
`captionClass` JSF 1.2, `captionStyle` JSF 1.2	CSS class or style for the caption; a panel caption is optionally supplied by a facet named `"caption"`
`binding`, `id`, `rendered`, `value`	Basic attributes[a]
`dir`, `lang`, `style`, `styleClass`, `title`, `width`	HTML 4.0[b]

Table 4–8 Attributes for h:panelGrid (cont.)

Attributes	Description
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	DHTML events[c]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 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 with h:panelGroup, which groups two or more components so they are treated as one. For example, you might group an input field and its 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. (We discuss the h:message tag in the section "Messages" on page 171.)

h:panelGroup is a simple tag with only a handful of attributes. Those attributes are listed in Table 4–9.

Table 4–9 Attributes for h:panelGroup

Attributes	Description
layout JSF 1.2	If the value is "block", use an HTML div to lay out the children; otherwise, use a span
binding, id, rendered	Basic attributes[a]
style, styleClass	HTML 4.0[b]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.

The Head, Body, and Form Tags

Table 4–10 shows the attributes of the h:head and h:body tags. All of them are basic or HTML/DHTML attributes.

Table 4–10 Attributes for h:head and h:body

Attributes	Description
id, binding, rendered	Basic attributes[a]
dir, lang h:body only: style, styleClass, target, title	HTML 4.0[b] attributes
h:body only: onclick, ondblclick, onkeydown, onkeypress, onkeyup, onload, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onunload	DHTML events[c]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 for information about DHTML event attributes.

Web applications run on form submissions, and JSF applications are no exception. Table 4–11 lists all h:form attributes.

Table 4–11 Attributes for `h:form`

Attributes	Description
`prependId` JSF 1.2	`true` (default) if the ID of this form is prepended to the IDs of its components; `false` to suppress prepending the form ID (useful if the ID is used in JavaScript code)
`binding`, `id`, `rendered`	Basic attributes[a]
`accept`, `acceptcharset`, `dir`, `enctype`, `lang`, `style`, `styleClass`, `target`, `title`	HTML 4.0[b] attributes
`onclick`, `ondblclick`, `onfocus`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `onreset`, `onsubmit`	DHTML events[c]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 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.

There is no need for an `anchor` attribute since JSF form submissions always post to the current page. (Navigation to a new page happens after the form data have been posted.)

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

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

If you do not specify the `id` attribute explicitly, a value is generated by the JSF implementation, as is the case for all generated HTML elements. You can explicitly specify the `id` attribute for forms so that it can be referenced in stylesheets or scripts.

Form Elements and JavaScript

Java*Server* Faces is all about *server*-side components, but it is 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 do not match, a JavaScript dialog is displayed. If they do match, the form is submitted.

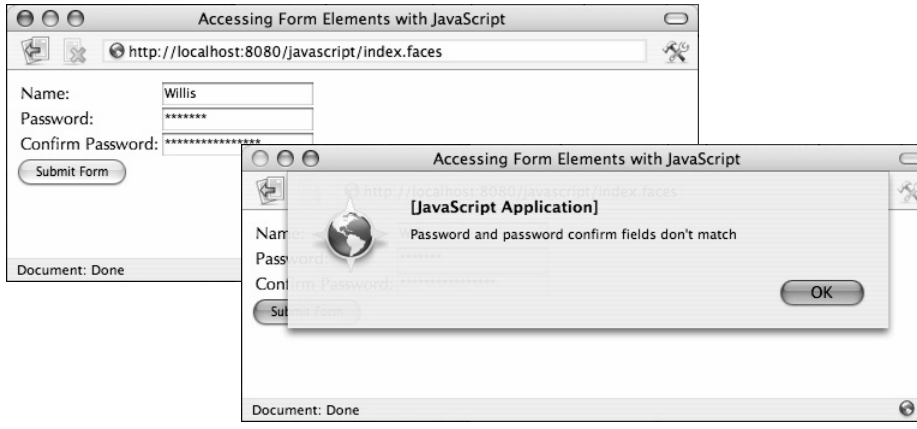


Figure 4–1 Using JavaScript to access form elements

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

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

When the user clicks the button, a JavaScript function `checkPassword` is invoked. Here is the implementation of the function:

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

   if (password == passwordConfirm)
      form.submit();
```

```
    else
       alert("Password and password confirm fields don't match");
}
```

To understand the syntax used to access form elements, look at the HTML produced by the preceding code:

```
<form id="_id0" method="post"
   action="/javascript/faces/index.xhtml"
   enctype="application/x-www-form-urlencoded">
   ...
   <input id="_id0:password"
      type="text" name="registerForm:password"/>
   ...
   <input type="button" name="_id0:_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 do not specify `id` attributes, the JSF implementation creates identifiers for you. In our case, we didn't specify an `id` for the form. Therefore, to access the password field in the preceding example, the script uses the expression:

```
form[form.id + ":password"]
```

NOTE: The ID values generated by the JSF implementation seem to get more complex with every version of JSF. In the past, they were fairly straightforward (such as `_id0`), but more recent versions use IDs such as `j_id2059540600_7ac21823`. For greater clarity, we use the simpler IDs in our examples.

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. The JavaScript code, stylesheets, and resource bundle are listed in Listings 4–2 through 4–4.

Figure 4–2 The JavaScript example directory structure

Listing 4–1 javascript/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.windowTitle}</title>
      <h:outputStylesheet library="css" name="styles.css"/>
      <h:outputScript library="javascript" name="checkPassword.js"/>
   </h:head>
   <h:body>
      <h:form>
         <h:panelGrid columns="2" columnClasses="evenColumns, oddColumns">
            #{msgs.namePrompt}
            <h:inputText/>
            #{msgs.passwordPrompt}
            <h:inputSecret id="password"/>
            #{msgs.confirmPasswordPrompt}
            <h:inputSecret id="passwordConfirm"/>
         </h:panelGrid>
         <h:commandButton type="button" value="Submit Form"
                          onclick="checkPassword(this.form)"/>
      </h:form>
   </h:body>
</html>
```

Listing 4–2 javascript/web/resources/javascript/checkPassword.js

```
1. function checkPassword(form) {
2.    var password = form[form.id + ":password"].value;
3.    var passwordConfirm = form[form.id + ":passwordConfirm"].value;
4.
5.    if (password == passwordConfirm)
6.       form.submit();
7.    else
8.       alert("Password and password confirm fields don't match");
9. }
```

Listing 4–3 javascript/web/resources/css/styles.css

```
1. .evenColumns {
2.    font-style: italic;
3. }
4.
5. .oddColumns {
6.    padding-left: 1em;
7. }
```

Listing 4–4 javascript/src/java/com/corejsf/messages.properties

```
1. windowTitle=Accessing Form Elements with JavaScript
2. namePrompt=Name:
3. passwordPrompt=Password:
4. 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 4–12 lists attributes for all three.

Table 4–12 Attributes for `h:inputText`, `h:inputSecret`, `h:inputTextarea`, and `h:inputHidden`

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 is notified of value changes.
`label` JSF 1.2	A description of the component for use in error messages. Does not apply to `h:inputHidden`.
`binding`, `converter`, `converterMessage` JSF 1.2, `id`, `rendered`, `required`, `requiredMessage` JSF 1.2, `value`, `validator`, `validatorMessage` JSF 1.2	Basic attributes.[a]
`accesskey`, `alt`, `dir`, `disabled`, `lang`, `maxlength`, `readonly`, `size`, `style`, `styleClass`, `tabindex`, `title`	HTML 4.0 pass-through attributes[b]—`alt`, `maxlength`, and `size` do not apply to `h:inputTextarea`. None apply to `h:inputHidden`.
`autocomplete`	If the value is `"off"`, render the nonstandard HTML attribute `autocomplete="off"` (`h:inputText` and `h:inputSecret` only).
`onblur`, `onchange`, `onclick`, `ondblclick`, `onfocus`, `onkeydown`, `onkeypress`, `onkeyup`, `onmousedown`, `onmousemove`, `onmouseout`, `onmouseover`, `onmouseup`, `onselect`	DHTML events. None apply to `h:inputHidden`.[c]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 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 "Immediate Components" on page 320 of Chapter 8 for more information about the `immediate` attribute.

Three attributes in Table 4–12 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 4–13 shows sample uses of the `h:inputText` and `h:inputSecret` tags.

Table 4–13 `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"/>`	12345
`<h:inputText value="1234567890" maxlength="6" size="10"/>`	123456

The first example in Table 4–13 produces the following HTML:

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

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

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

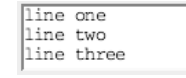
The h:inputSecret examples illustrate the use of the redisplay attribute. If that attribute is true, the text field stores its value between requests and, therefore, the value is redisplayed when the page reloads. If redisplay is false, the value is discarded and is not redisplayed.

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 4–13, 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 4–14 shows examples of the h:inputTextarea tag.

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

Table 4–14 h:inputTextarea Examples

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

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 4–14. If you want to put data on separate lines, you can insert newline characters (\n) to force a line break. For example, the last example in Table 4–14 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;
}
```

Hidden Fields

JSF provides support for hidden fields with h:inputHidden. Hidden fields are often used with JavaScript actions to send data back to the server. The h:inputHidden tag has the same attributes as the other input tags, except that it does not support the standard HTML and DHTML tags.

Using Text Fields and Text Areas

Next, we 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. The values of those components are wired to bean properties, which are accessed in the thankYou.xhtml 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.

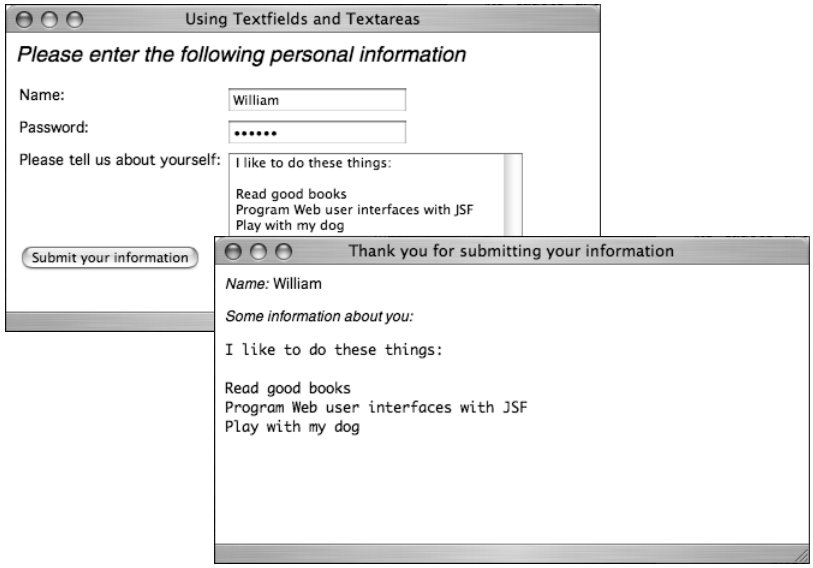


Figure 4–3 Using text fields and text areas

Figure 4–4 shows the directory structure for the application shown in Figure 4–3. Listings 4–5 through 4–8 show the pertinent JSF pages, managed beans, faces configuration file, and resource bundle.

Figure 4–4 Directory structure of the text fields and text areas example

Listing 4–5 personalData/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.indexWindowTitle}</title>
   </h:head>
   <h:body>
      <h:outputText value="#{msgs.indexPageTitle}"
                    style="font-style: italic; font-size: 1.5em"/>
      <h:form>
         <h:panelGrid columns="2">
            #{msgs.namePrompt}
            <h:inputText value="#{user.name}"/>
            #{msgs.passwordPrompt}
            <h:inputSecret value="#{user.password}"/>
            #{msgs.tellUsPrompt}
            <h:inputTextarea value="#{user.aboutYourself}" rows="5" cols="35"/>
         </h:panelGrid>
         <h:commandButton value="#{msgs.submitPrompt}" action="thankYou"/>
      </h:form>
   </h:body>
</html>
```

Listing 4–6 personalData/web/thankYou.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.thankYouWindowTitle}</title>
   </h:head>
   <h:body>
      <h:outputText value="#{msgs.namePrompt}" style="font-style: italic"/>
      #{user.name}
      <br/>
      <h:outputText value="#{msgs.aboutYourselfPrompt}" style="font-style: italic"/>
      <br/>
      <pre>#{user.aboutYourself}</pre>
   </h:body>
</html>
```

Listing 4–7 personalData/src/java/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. import javax.inject.Named;
6.    // or import javax.faces.bean.ManagedBean;
7. import javax.enterprise.context.SessionScoped;
8.    // or import javax.faces.bean.SessionScoped;
9.
10. @Named("user") // or @ManagedBean(name="user")
11. @SessionScoped
12. public class UserBean implements Serializable {
13.    private String name;
14.    private String password;
15.    private String aboutYourself;
16.
17.    public String getName() { return name; }
18.    public void setName(String newValue) { name = newValue; }
19.
20.    public String getPassword() { return password; }
21.    public void setPassword(String newValue) { password = newValue; }
22.
23.    public String getAboutYourself() { return aboutYourself; }
24.    public void setAboutYourself(String newValue) { aboutYourself = newValue; }
25. }
```

Listing 4–8 personalData/src/java/com/corejsf/messages.properties

```
1. indexWindowTitle=Using Textfields and Textareas
2. thankYouWindowTitle=Thank you for submitting your information
3. thankYouPageTitle=Thank you!
4. indexPageTitle=Please enter the following personal information
5. namePrompt=Name:
6. passwordPrompt=Password:
7. tellUsPrompt=Please tell us about yourself:
8. aboutYourselfPrompt=Some information about you:
9. 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.

In JSF 2.0, you don't usually need the `h:outputText` tag since you can simply insert value expressions, such as `#{msgs.namePrompt}` into your page. You would use `h:outputText` in the following circumstances:

- To produce styled output
- In a panel grid to make sure that the text is considered one cell of the grid
- To generate HTML markup

The `h:outputText` and `h:outputFormat` tags have one attribute that is unique among all JSF tags: `escape`. By default, the `escape` attribute is `true`, which causes the characters < > & to be converted to `<` `>` and `&` 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.) Set this attribute to `false` if you want to programmatically generate HTML markup.

NOTE: The `value` attribute of `h:outputText` can never contain < characters. The only way to produce HTML markup with `h:outputText` is with a value expression.

NOTE: When you include a value expression such as `#{msgs.namePrompt}` in your page, the resulting value is always escaped. You must use `h:outputText` if you want to generate HTML markup.

Table 4–15 lists all `h:outputText` attributes.

Table 4–15 Attributes for `h:outputText` and `h:outputFormat`

Attributes	Description
`escape`	If set to `true` (default), escapes <, >, and & characters
`binding`, `converter`, `id`, `rendered`, `value`	Basic attributes[a]
`style`, `styleClass`, `title`, `dir` JSF 1.2, `lang` JSF 1.2	HTML 4.0[b]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 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 its output.

The `h:graphicImage` tag generates an HTML `img` element. You can specify the image location with the `url` or `value` attribute, as a context-relative path—meaning relative to the web application's context root. As of JSF 2.0, you can place images into the `resources` directory and specify a library and name:

```
<h:graphicImage library="images" name="de_flag.gif"/>
```

Here, the image is located in `resources/images/de_flag.gif`. Alternatively, you can use this:

```
<h:graphicImage url="/resources/images/de_flag.gif"/>
```

You can also use the `resources` map:

```
<h:graphicImage value="#{resources['images:de_flag.gif']}"/>
```

Table 4–16 shows all the attributes for `h:graphicImage`.

Table 4–16 Attributes for h:graphicImage

Attributes	Description
binding, id, rendered, value	Basic attributes[a]
alt, dir, height, ismap, lang, longdesc, style, styleClass, title, url, usemap, width	HTML 4.0[b]
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	DHTML events[c]
library, name JSF 2.0	The resource library and name for this image

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 for information about DHTML event attributes.

Table 4–17 shows some examples of using h:outputText and h:graphicImage.

Table 4–17 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="#{form.htmlCode}" escape="false"/>` where the getHtmlCode method returns the string `"<input type='text' value='hello'/>"`	hello
`<h:outputText value="#{form.htmlCode}"/>` where the getHtmlCode method returns the string `"<input type='text' value='hello'/>"`	`<input type="text" value="hello">`
`<h:graphicImage value="/tjefferson.jpg"/>`	
`<h:graphicImage library="images" name="tjefferson.jpg" style="border: thin solid black"/>`	

The third and fourth examples in Table 4–17 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 4–17—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 4–17—that output is transformed to harmless text, thereby thwarting a potential attack.

The final two examples in Table 4–17 show you how to use `h:graphicImage`.

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:button`
- `h:link`
- `h:outputLink`

The `h:commandButton` and `h:commandLink` are the primary components for navigating within a JSF application. When a button or link is activated, a POST request sends the form data back to the server.

JSF 2.0 introduced the `h:button` and `h:link` components. These components also render buttons and links, but clicking on them issues a bookmarkable GET request instead. We discussed this mechanism in Chapter 3.

The `h:outputLink` tag generates an HTML `anchor` element that points to a resource such as an image or a web page. Clicking the generated link takes you to the designated resource without further involving the JSF framework. These links are most suitable for navigating to a different web site.

Table 4–18 lists the attributes shared by `h:commandButton`, `h:commandLink`, `h:button`, and `h:link`.

Table 4–18 Attributes for h:commandButton, h:commandLink, h:button, and h:link

Attribute	Description
action (h:commandButton and h:commandLink only)	*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 expression:* The method has this signature: `String methodName()`; the string represents the outcome. *If omitted:* Activating the button or link redisplays the current page.
outcome (h:button and h:link only)	The outcome, used by the navigation handler to determine the target view when the component is rendered.
fragment (h:button and h:link only)	A fragment that is to be appended to the target URL. The # separator is applied automatically and should not be included in the fragment.
actionListener	A method expression that refers to a method with this signature: `void methodName(ActionEvent)`.
image (h:commandButton and h:button only)	The path to an image displayed in a button. If you specify this attribute, the HTML input's type will be image. If the path starts with a /, the application's context root is prepended.
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 8 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 and h:link—The content type of the linked resource; for example, text/html, image/gif, or audio/basic.

Table 4–18 Attributes for h:commandButton, h:commandLink, h:button, and h:link (cont.)

Attribute	Description
value	The label displayed by the button or link. You can specify a string or a value expression.
binding, id, rendered	Basic attributes.[a]
accesskey, charset (h:commandLink and h:link only), coords (h:commandLink and h:link only), dir JSF 1.1, disabled (h:commandButton only in JSF 1.1), hreflang (h:commandLink and h:link only), lang, rel (h:commandLink and h:link only), rev (h:commandLink and h:link only), shape (h:commandLink and h:link only), style, styleClass, tabindex, target (h:commandLink and h:link only), title	HTML 4.0.[b]
onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	DHTML events.[c]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 for information about DHTML event attributes.

Using Buttons

The h:commandButton and h:button tags generate an HTML input element whose type is button, image, submit, or reset, depending on the attributes you specify. Table 4–19 illustrates some uses of these tags.

The third example in Table 4–19 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.

Table 4–19 `h:commandButton` and `h:button` Examples

Example	Result
`<h:commandButton value="submit" type="submit" action="#{form.submitAction}"/>`	submit
`<h:commandButton value="reset" type="reset"/>`	reset
`<h:commandButton value="click this button..." onclick="alert('button clicked')" type="button"/>`	click this button to execute JavaScript
`<h:commandButton value="disabled" disabled="#{not form.buttonEnabled}"/>`	disabled
`<h:button value="#{form.buttonText}" outcome="#{form.pressMeOutcome}"/>`	press me

CAUTION: In JSF 1.1, there was an inconsistency in the handling of image paths between `h:graphicImage` and `h:commandButton`. The context root is automatically added by `h:graphicImage`, but not by `h:commandButton`. For example, for an application named `myApp`, here is how you specified the same image for each tag:

```
<h:commandButton image="/myApp/imageFile.jpg"/> <!-- JSF 1.1 -->
<h:graphicImage value="/imageFile.jpg"/>
```

This was annoying because it required the page to know the context root. The `h:commandButton` behavior changed in JSF 1.2. Now the context root is automatically added if the path starts with a `/`.

To preserve a level of annoyance, this feature interacts poorly with a `resource` map. You cannot use

```
<h:commandButton image="#{resources['images:imageFile.jpg']}"/>
```

because the string returned by the `resource` map starts with */context-root*. The result would be `<input type="image" src="/context-root/context-root/..."/>`

The `h:commandLink` and `h:link` tags generates an HTML `anchor` element that acts like a form submit button. Table 4–20 shows some examples.

Table 4–20 h:commandLink and h:link Examples

Example	Result
`<h:commandLink>register</h:commandLink>`	register

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

`<h:commandLink>`
`  #{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:link value="welcome"`
`  outcome="#{form.welcomeOutcome}">`
`  <f:param name="id" value="#{form.userId}"/>`
`</h:link>`
	welcome

The h:commandLink and h:link tags generate JavaScript to make links act like buttons. For example, here is the HTML generated by the first example in Table 4–20:

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

When the user clicks the link, 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.

You can place as many children 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 4–20, the link's form will be submitted.

The next-to-last example in Table 4–20 attaches an action listener, in addition to an action, to a link. Action listeners are discussed in "Action Events" on page 312 of Chapter 8.

The last example in Table 4–20 embeds an f:param tag in the body of the h:link tag. When you click the link, a request parameter with the name and value specified with the f:param tag is created by the link. In Chapter 2, we discussed how the request parameters can be processed. You can also use request parameters with an h:commandLink or h:commandButton. See "Passing Data from the UI to the Server" on page 324 of Chapter 8 for an example.

Like h:commandLink and h:link, 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 4–21 lists all attributes for h:outputLink., and Table 4–22 shows some h:outputLink examples.

Table 4–21 Attributes for h:outputLink

Attributes	Description
binding, converter, id, lang, rendered, value	Basic attributes[a]
accesskey, charset, coords, dir, disabled JSF 1.2, hreflang, lang, rel, rev, shape, style, styleClass, tabindex, target, title, type	HTML 4.0[b]
onblur, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	DHTML events[c]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 for information about DHTML event attributes.

Table 4–22 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}"> #{form.welcomeLinkText} </h:outputLink>	go to welcome page

Table 4–22 `h:outputLink` **Examples (cont.)**

Example	Result

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

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

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

The first example in Table 4–22 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 4–22 are links to named anchors in the same JSF page. Those anchors look like this:

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

CAUTION: If you use JSF 1.1, you need to use the `f:verbatim` tag when you want to place text inside a tag. For example, the last example in Table 4–22 had to be:

```
<h:outputLink...><f:verbatim>Table of Contents</f:verbatim></h:outputLink>
```

In JSF 1.1, the text would appear outside the link. The remedy is to place the text inside another component, such as `h:outputText` or `f:verbatim`. This problem has been fixed in JSF 1.2.

Using Command Links

Now that we have discussed the details of JSF tags for buttons and links, we take a look at a complete example. Figure 4–5 shows the application discussed in "Using Text Fields and Text Areas" on page 127, 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.

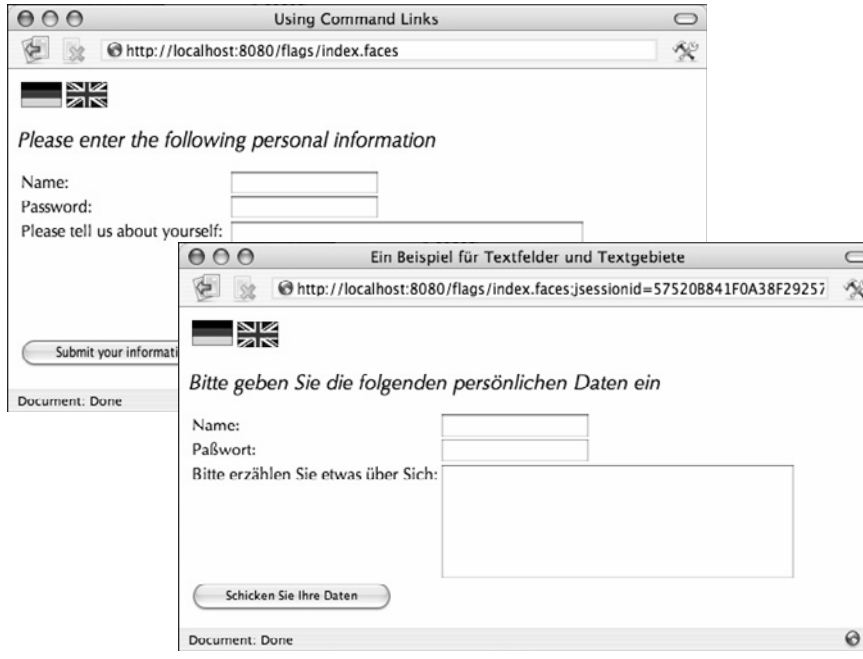


Figure 4–5 Using command links to change locales

The links are implemented like this:

```
<h:commandLink action="#{localeChanger.englishAction}">
   <h:graphicImage library="images" name="en_flag.gif" style="border: 0px" />
</h:commandLink>
```

Both links specify an image and an action method. The method to change to the English locale looks like this:

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

Because we have not specified any navigation for this action, 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 Listings 4–9 through 4–11 show the associated JSF pages and Java classes.

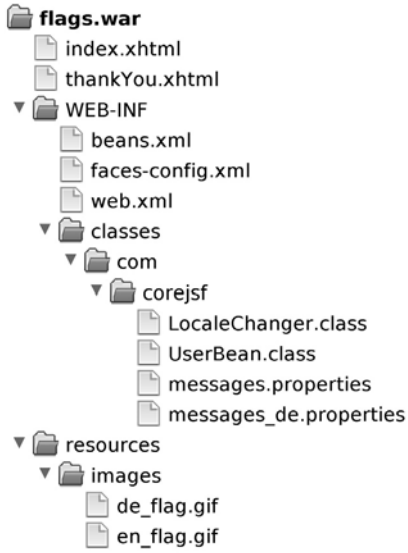


Figure 4–6 Directory structure of the flags example

Listing 4–9 flags/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.indexWindowTitle}</title>
   </h:head>
   <h:body>
      <h:form>
         <h:commandLink action="#{localeChanger.germanAction}">
            <h:graphicImage library="images" name="de_flag.gif"
                            style="border: 0px; margin-right: 1em;"/>
         </h:commandLink>
         <h:commandLink action="#{localeChanger.englishAction}">
            <h:graphicImage library="images"
                            name="en_flag.gif" style="border: 0px"/>
         </h:commandLink>
         <p><h:outputText value="#{msgs.indexPageTitle}"
         style="font-style: italic; font-size: 1.3em"/></p>
         <h:panelGrid columns="2">
            #{msgs.namePrompt}
            <h:inputText value="#{user.name}"/>
            #{msgs.passwordPrompt}
            <h:inputSecret value="#{user.password}"/>
            #{msgs.tellUsPrompt}
            <h:inputTextarea value="#{user.aboutYourself}" rows="5" cols="35"/>
         </h:panelGrid>
         <h:commandButton value="#{msgs.submitPrompt}" action="thankYou"/>
      </h:form>
   </h:body>
</html>
```

Listing 4–10 flags/src/java/com/corejsf/UserBean.java

```
package com.corejsf;

import java.io.Serializable;

import javax.inject.Named;
   // or import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // or import javax.faces.bean.SessionScoped;

```

```
@Named("user") // or @ManagedBean(name="user")
@SessionScoped
public class UserBean implements Serializable {
   private String name;
   private String password;
   private String aboutYourself;

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

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

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

Listing 4–11 flags/src/java/com/corejsf/LocaleChanger.java

```
package com.corejsf;

import java.io.Serializable;
import java.util.Locale;

import javax.inject.Named;
   // or import javax.faces.bean.ManagedBean;
import javax.enterprise.context.SessionScoped;
   // or import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;

@Named // or @ManagedBean
@SessionScoped
public class LocaleChanger implements Serializable {
   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;
   }
}
```

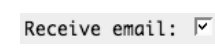
Selection Tags

JSF has seven tags for making selections:

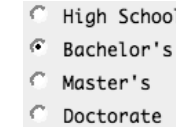
- h:selectBooleanCheckbox
- h:selectManyCheckbox
- h:selectOneRadio
- h:selectOneListbox
- h:selectManyListbox
- h:selectOneMenu
- h:selectManyMenu

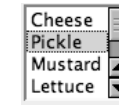
Table 4–23 shows examples of each tag.

Table 4–23 Selection Tag Examples

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

h:selectManyCheckbox	`<table>`
	`...`
	`<label>`
	`<input type="checkbox"/>`
	`</label>`
	`...`
	`</table>`
		

h:selectOneRadio	`<table>`
	`...`
	`<label>`
	`<input type="radio"/>`
	`</label>`
	`...`
	`</table>`
		

h:selectOneListbox	`<select>`
	`<option value="Cheese">`
	`Cheese`
	`</option>`
	`...`
	`</select>`
		

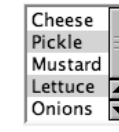
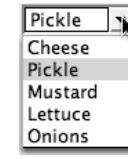
h:selectManyListbox	`<select multiple>`
	`<option value="Cheese">`
	`Cheese`
	`</option>`
	`...`
	`</select>`
		

Table 4–23 Selection Tag Examples (cont.)

Tag	Generated HTML	Examples
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 4–24.

Table 4–24 Attributes for h:selectBooleanCheckbox, h:selectManyCheckbox, h:selectOneRadio, h:selectOneListbox, h:selectManyListbox, h:selectOneMenu, and h:selectManyMenu

Attributes	Description
enabledClass, disabledClass JSF 2.0	CSS class for enabled or disabled elements—for h:selectOneRadio and h:selectManyCheckbox only.
selectedClass, unselectedClass JSF 2.0	CSS class for selected or unselected elements—for h:selectManyCheckbox only.
layout	Specification for how elements are laid out: lineDirection (horizontal) or pageDirection (vertical)—for h:selectOneRadio and h:selectManyCheckbox only.

Table 4–24 Attributes for h:selectBooleanCheckbox, h:selectManyCheckbox, h:selectOneRadio, h:selectOneListbox, h:selectManyListbox, h:selectOneMenu, and h:selectManyMenu (cont.)

Attributes	Description
label JSF 1.2	A description of the component for use in error messages.
collectionType JSF 2.0	(selectMany tags only) A string or a value expression that evaluates to a fully qualified collection class name, such as java.util.TreeSet. See "The value Attribute and Multiple Selections" on page 162.
hideNoSelectionOption JSF 2.0	Hide any item that is marked as the "no selection option". See "The f:selectItem Tag" on page 153.
binding, converter, converterMessage JSF 1.2, requiredMessage JSF 1.2, id, immediate, required, rendered, validator, validatorMessage JSF 1.2, value, valueChangeListener	Basic attributes.[a]
accesskey, border, dir, disabled, lang, readonly, style, styleClass, 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]

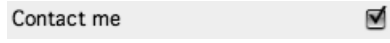
a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.
c. See Table 4–7 on page 114 for information about DHTML event attributes.

Checkboxes and Radio Buttons

Two JSF tags represent checkboxes:

- h:selectBooleanCheckbox
- h:selectManyCheckbox

The h:selectBooleanCheckbox tag 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 153 for more information about those core tags. For example, here is a group of checkboxes for selecting colors:

The h:selectManyCheckbox tag looks like this:

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

The checkboxes are specified with f:selectItem (page 153) or f:selectItems (page 155).

The h:selectManyCheckbox tag generates an HTML table element; here is 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. That label is placed in a td element.

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

The value attribute of the h:selectOneRadio tag 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" itemLabel="High School"/>
   <f:selectItem itemValue="Bachelor's" itemLabel="Bachelor's"/>
   <f:selectItem itemValue="Master's" itemLabel="Master's"/>
   <f:selectItem itemValue="Doctorate" itemLabel=Doctorate"/>
</h:selectOneRadio>
```

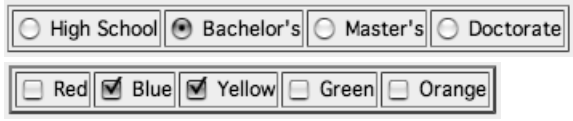
Like h:selectManyCheckbox, h:selectOneRadio generates an HTML table. Here is the table generated by the preceding tag:

```
<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:



The `enabledClass` and `disabledClass` attributes 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`:

NOTE: You might wonder why `layout` attribute values are not `horizontal` and `vertical`, instead of `lineDirection` and `pageDirection`, respectively. Although `lineDirection` and `pageDirection` are indeed horizontal and vertical for Latin-based languages, that is 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

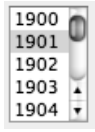
Menus and listboxes are represented by the following tags:

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

The attributes for the preceding tags are listed in Table 4–24 on page 146, 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 is a single-select listbox:



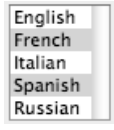
The corresponding listbox tag looks like this:

```
<h:selectOneListbox value="#{form.year}" size="5">
   <f:selectItem itemValue="1900" itemLabel="1900"/>
   <f:selectItem itemValue="1901" itemLabel="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" itemLabel="English"/>
   <f:selectItem itemValue="French" itemLabel="French"/>
   <f:selectItem itemValue="Italian" itemLabel="Italian"/>
   <f:selectItem itemValue="Spanish" itemLabel="Spanish"/>
   <f:selectItem itemValue="Russian" itemLabel="Russian"/>
</h:selectManyListbox>
```

This time we do not 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="1" itemLabel="Sunday"/>
   <f:selectItem itemValue="2" itemLabel="Monday"/>
   <f:selectItem itemValue="3" itemLabel="Tuesday"/>
   <f:selectItem itemValue="4" itemLabel="Wednesday"/>
   <f:selectItem itemValue="5" itemLabel="Thursday"/>
   <f:selectItem itemValue="6" itemLabel="Friday"/>
   <f:selectItem itemValue="7" itemLabel="Saturday"/>
</h:selectOneMenu>
```

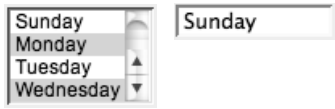
Here is the generated HTML:

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

The `h:selectManyMenu` tag is used for multiselect menus. That tag generates HTML, which looks like this:

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

That HTML does not yield consistent results among browsers. For example, here is `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, some browsers render absurdities such as tiny scrollbars that are nearly impossible to manipulate (Internet Explorer) or no scrollbar at all, leaving you to navigate with arrow keys (Firefox).

Items

Starting with "Checkboxes and Radio Buttons" on page 148, we have used multiple `f:selectItem` tags to populate `select` components. Now that we are familiar with the visual appearance of selection tags, we take a closer look at `f:selectItem` and the related `f:selectItems` tags.

The `f:selectItem` Tag

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

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

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. The `itemLabel` values are used as labels for the menu items. Sometimes you want to specify different values for request parameter values and item labels:

```
<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:selectOneMenu>
```

In the preceding code, the item values are strings. "Binding the value Attribute" on page 161 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 itemDisabled attribute, however, is passed to HTML. The f:selectItem tag has the attributes shown in Table 4–25.

As of JSF 2.0, there is a noSelectionOption attribute for marking an item that is included for navigational purposes, such as "Select a condiment". This attribute is used in conjunction with validation. If an entry is required and the user selects the "no selection option", a validation error occurs.

Table 4–25 Attributes for f:selectItem

Attribute	Description
binding, id	Basic attributes[a]
itemDescription	Description used by tools only
itemDisabled	Boolean value that sets the item's disabled HTML attribute
itemLabel	Text shown by the item
itemValue	Item's value, passed to the server as a request parameter
value	Value expression that points to a SelectItem instance
escape JSF 1.2	true if special characters in the value should be converted to character entities (default), false if the value should be emitted without change
noSelectionOption JSF 2.0	true if this item is the "no selection" option that, when selected, indicates that the user intends to made no selection

a. See Table 4–5 on page 107 for information about basic attributes.

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

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

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

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

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.
- SelectItem(Object value, String label, String description, boolean disabled, boolean noSelectionOption) JSF 2.0
  Creates a SelectItem with a value, label, description, disabled state, and "no selection option" flag.

The f:selectItems Tag

As we saw in "The f:selectItem Tag" on page 153, f:selectItem is versatile, but it is tedious for specifying more than a few items. The first code fragment shown in that section can be reduced to the following with f:selectItems:

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

The value expression #{form.condimentItems} could point to an array of SelectItem instances:

```
private static SelectItem[] condimentItems = {
   new SelectItem(1, "Cheese"),
   new SelectItem(2, "Pickle"),
   new SelectItem(3, "Mustard"),
   new SelectItem(4, "Lettuce"),
```

```
        new SelectItem(5, "Onions")
    };

    public SelectItem[] getCondimentItems() {
        return condimentItems;
    }
```

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

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

The first option is not very useful. We discuss the other options in the following sections.

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

Table 4–26 summarizes the attributes of the `f:selectItems` tag.

Table 4–26 Attributes for `f:selectItems`

Attribute	Description
`binding`, `id`	Basic attributes[a]
`value`	Value expression that points to a `SelectItem` instance, an array or collection, or a map
`var` JSF 2.0	The name of a variable, used in the value expressions below when traversing an array or collection of objects other than `SelectItem`
`itemLabel` JSF 2.0	Value expression yielding the text shown by the item referenced by the `var` variable

Table 4–26 Attributes for `f:selectItems` (cont.)

Attribute	Description
`itemValue` JSF 2.0	Value expression yielding the value of the item referenced by the `var` variable
`itemDescription` JSF 2.0	Value expression yielding the description of the item referenced by the `var` variable; the description is intended for use by tools
`itemDisabled` JSF 2.0	Value expression yielding the `disabled` HTML attribute of the item referenced by the `var` variable
`itemLabelEscaped` JSF 2.0	Value expression yielding `true` if special characters in the item's value should be converted to character entities (default), `false` if the value should be emitted without change
`noSelectionOption` JSF 2.0	Value expression that yields the "no selection option" item or string that equals the value of the "no selection option" item

a. See Table 4–5 on page 107 for information about basic attributes.

Using Collections and Arrays with `f:selectItems`

Before JSF 2.0, collections and arrays had to contain `SelectItem` instances. That was unfortunate because it coupled your business logic to the JSF API. As of JSF 2.0, the `value` of `f:selectItems` can be a collection or array containing objects of *any* type.

If they are instances of `SelectItem`, no further processing is done. Otherwise, the labels are obtained by calling `toString` on each object.

Alternatively, you cau use the `var` attribute to define a variable that iterates over the array or collection. Then you supply value expressions for the label and value in the attributes `itemLabel` and `itemValue`.

For example, suppose you want users to select objects of the following class:

```
public class Weekday {
   public String getDayName() { ... } // name in current locale, such as "Monday"
   public int getDayNumber() { ... } // number such as Calendar.MONDAY (2)
   ...
}
```

Use the following tag:

```
<f:selectItems value="#{form.daysOfTheWeek}"
   var="w"
```

```
    itemLabel="#{w.dayName}"
    itemValue="#{w.dayNumber}" />
```

Here, `#{form.daysOfTheWeek}` yields an array or collection of `Weekday` objects. The variable `w` is set to each of the elements. Then a `SelectItem` object is constructed with the results of the `itemLabel` and `itemValue` expressions.

NOTE: The `var` attribute in the `f:selectItems` tag is conceptually similar to the use of the `var` attribute in the `h:dataTable` which we will discuss in Chapter 6.

Using Maps with `f:selectItems`

If the `value` attribute of the `f:selectItems` tag yields 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 static Map<String, Object> condimentItems;
static {
   condimentItems = new LinkedHashMap<String, Object>();
   condimentItems.put("Cheese",  1);  // label, value
   condimentItems.put("Pickle",  2);
   condimentItems.put("Mustard", 3);
   condimentItems.put("Lettuce", 4);
   condimentItems.put("Onions",  5);
}

public Map<String, Object> getCondimentItems() {
   return condimentItems;
}
```

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

Pay attention to these two issues when using a map:

1. You will generally want to use a `LinkedHashMap`, not a `TreeMap` or `HashMap`. In a `LinkedHashMap`, you can control the order of the items because items are visited in the order in which they were inserted. If you use a `TreeMap`, the labels that are presented to the user (which are the keys of the map) are sorted alphabetically. That may or may not be what you want. For example, days of the week would be neatly arranged as Friday Monday Saturday Sunday Thursday Tuesday Wednesday. If you use a `HashMap`, the items are ordered randomly.

2. Map keys are turned into item labels and map values into item values. When a user selects an item, your backing bean receives a value in your map, not a key. For example, in the example above, if the backing bean receives a value of 5, you would need to iterate through the entries if you wanted to find the matching "Onions". Since the value is probably more meaningful to your application than the label, this is usually not a problem, just something to be aware of.

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` property is 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 that we are 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 code generates the following HTML:

```
<select name="_id0:_id1" multiple size="16">
   <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)`
  Creates a group with a label, a description (which is ignored by the JSF Reference Implementation), a `boolean` that disables all 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

Whether you are using a set of checkboxes, a menu, or a listbox, you will want to keep track of the item or items selected by the user. For that purpose, you use the `value` attribute of the `selectOne` and `selectMany` tags. Consider this example:

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

The value attribute of h:selectOneMenu refers to the value that the user selects. The value attribute of f:selectItems specifies all possible values.

Suppose the radio buttons were specified with an array of SelectItem objects, containing the following:

```
new SelectItem(1, "Sunday"),   // value, label
new SelectItem(2, "Monday"),
...
```

The user sees the labels (Sunday, Monday, ...), but the application uses the values (1, 2, ...).

There is an important but subtle issue about the Java type of the values. In the web page, *the values are always strings*:

```
<option value="1">Sunday</option>
<option value="2">Monday</option>
```

When the page is submitted, the server receives the selected string and must convert it to an appropriate type. The JSF implementation knows how to convert to numbers and enumerated types, but for other types you need to define a converter. (We discuss converters in Chapter 7.)

In our example, the #{form.bestDay} value expression should refer to a property of type int or Integer. Listing 4–13 has an example where the value is an enumerated type.

CAUTION: Because the value of a SelectItem is an Object, it can be tempting to set it to the value that you actually need in your application. However, keep in mind that the value is turned into a string when it is sent to the client. For example, consider a SelectItem(Color.RED, "Red"). The client receives the client is the string "java.awt.Color[r=255,g=0,b=0]". That string is returned when the user selects the option with label "Red". You would have to parse it to turn it back into a color. It is easier to send the RGB value of the color instead.

The value Attribute and Multiple Selections

You can keep track of multiple selections with a selectMany tag. These tags have a value attribute that specifies zero or more selected items, using an array or collection.

Consider an h:selectManyListbox that lets 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(1, "Cheese"),
   new SelectItem(2, "Pickle"),
   new SelectItem(3, "Mustard"),
   new SelectItem(4, "Lettuce"),
   new SelectItem(5, "Onions"),
};
public SelectItem[] getCondimentItems() {
   return condimentItems;
}

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

Instead of an int[] array for the condiments property, you could have used an Integer[] array.

The value of a selectMany tag can be a collection instead of an array, but there are two technical issues that you need to keep in mind. Most importantly, the elements cannot be converted because the collection's element type is not known at runtime. (This is an unfortunate aspect of Java generics. At runtime, an ArrayList<Integer> or ArrayList<String> is only a raw ArrayList, and there is no way of determining the element type. In contrast, Integer[] and String[] are distinct types at runtime.) That means, you should use collections only for strings.

The other complexity is more subtle. When the JSF application receives the user choices, it must construct a new instance of the collection, populate it, and pass the collection to the property setter. But suppose the property type is Set<String>. What kind of Set should be constructed?

Before JSF 2.0, this was not clearly specified. JSF 2.0 lays down the following rules:

1. If the tag has a collectionType attribute, its value must be a string or a value expression that evaluates to a fully qualified classname, such as java.util.TreeSet. Instantiate that class.
2. Otherwise, get the existing value and try cloning and clearing it.

3. If that fails (perhaps because the existing value was `null` or not cloneable), look at the type of the `value` expression. If that type is `SortedSet`, `Set`, or `Queue`, construct a `TreeSet`, `HashSet`, or `LinkedList`.
4. Otherwise, construct an `ArrayList`.

For example, suppose you define a `languages` property:

```
private Set<String> languages; // initialized with null
public Set<String> getLanguages() {
   return languages;
}
public void setLanguages(Set<String> newValue) {
   languages = newValue;
}
```

When the form is submitted for the first time, the property setter is called with a `HashSet` that contains the user choices (step 3). In subsequent invocations, that set is cloned (step 2). However, suppose you initialize the set:

```
private Set<String> languages = new TreeSet();
```

Then a clone of that `TreeSet` is always returned.

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

We close out our section on selection tags with an example that exercises nearly all 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 be contacted, and `h:selectOneMenu` lets the user select the best day of the week for us to do so.

The year listbox is implemented with `h:selectOneMenu`, and it demonstrates the use of a "no selection" item. The language checkboxes are implemented with `h:selectManyCheckbox`; the education level is implemented with `h:selectOneRadio`.

Note that the languages are collected in a `Set<String>`. Also note the styles in the color selector. The disabled Orange option is colored gray, and the selected colors are marked in bold. We use the attribute `onchange="submit()"` in order to update the styles immediately upon selection.

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

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 shown in Listings 4–12 through 4–16.

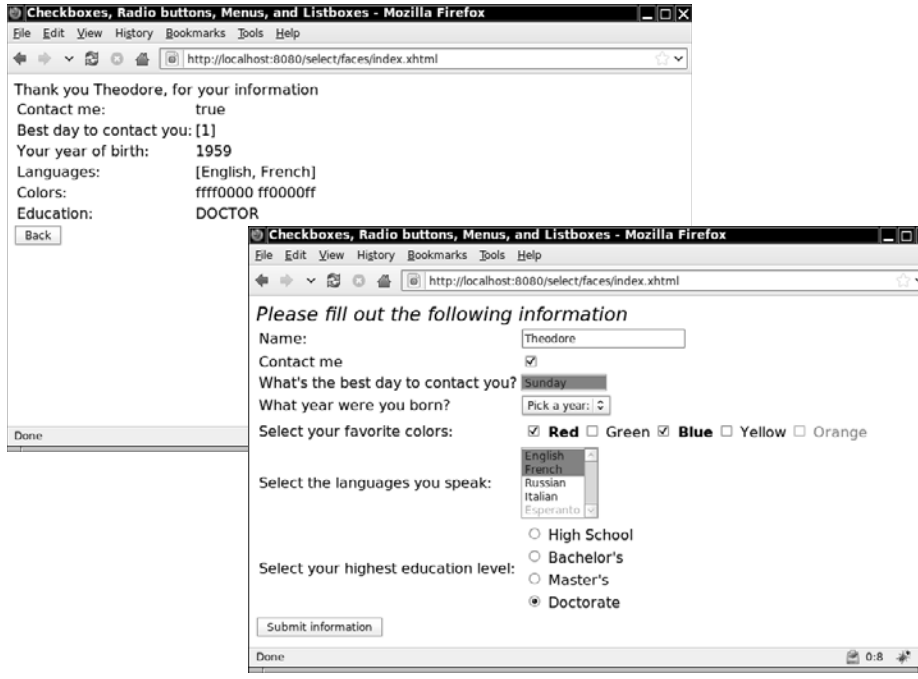


Figure 4–7 Using checkboxes, radio buttons, menus, and listboxes

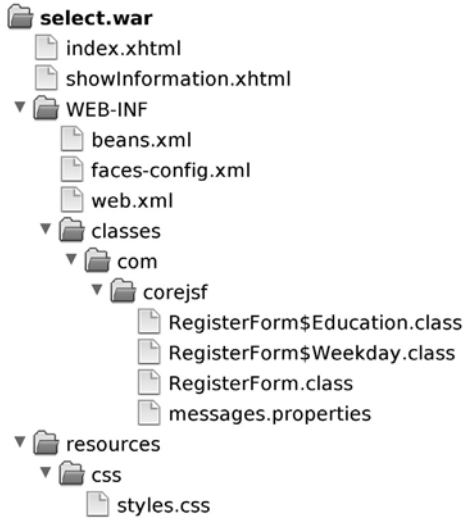


Figure 4–8 The directory structure of the selection example

Listing 4–12 select/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <h:outputStylesheet library="css" name="styles.css"/>
      <title>#{msgs.indexWindowTitle}</title>
   </h:head>

   <h:body>
      <h:outputText value="#{msgs.indexPageTitle}" styleClass="emphasis"/>
      <h:form>
         <h:panelGrid columns="2">
            #{msgs.namePrompt}
            <h:inputText value="#{form.name}"/>
            #{msgs.contactMePrompt}
            <h:selectBooleanCheckbox value="#{form.contactMe}"/>
            #{msgs.bestDayPrompt}
            <h:selectManyMenu value="#{form.bestDaysToContact}">
               <f:selectItems value="#{form.daysOfTheWeek}" var="w"
                              itemLabel="#{w.dayName}" itemValue="#{w.dayNumber}"/>
            </h:selectManyMenu>
            #{msgs.yearOfBirthPrompt}
            <h:selectOneMenu value="#{form.yearOfBirth}" required="true">
               <f:selectItems value="#{form.yearItems}"/>
            </h:selectOneMenu>
            #{msgs.colorPrompt}
            <h:selectManyCheckbox value="#{form.colors}"
                                  selectedClass="selected" disabledClass="disabled"
                                  onchange="submit()">
               <f:selectItems value="#{form.colorItems}"/>
            </h:selectManyCheckbox>
            #{msgs.languagePrompt}
            <h:selectManyListbox size="5" value="#{form.languages}">
               <f:selectItems value="#{form.languageItems}"/>
            </h:selectManyListbox>
            #{msgs.educationPrompt}
            <h:selectOneRadio value="#{form.education}"
                              selectedClass="selected" layout="pageDirection">
               <f:selectItems value="#{form.educationItems}"/>
            </h:selectOneRadio>
         </h:panelGrid>
         <h:commandButton value="#{msgs.buttonPrompt}" action="showInformation"/>
```

```
45.       </h:form>
46.       <h:messages/>
47.    </h:body>
48. </html>
```

Listing 4–13 select/web/showInformation.xhtml

```
 1. <?xml version="1.0" encoding="UTF-8"?>
 2. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 3. "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
 4. <html xmlns="http://www.w3.org/1999/xhtml"
 5.       xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
 6.    <h:head>
 7.       <title>#{msgs.indexWindowTitle}</title>
 8.    </h:head>
 9.    <h:body>
10.       <h:form>
11.          <h:outputStylesheet library="css" name="styles.css" target="head"/>
12.          <h:outputFormat value="#{msgs.thankYouLabel}">
13.             <f:param value="#{form.name}"/>
14.          </h:outputFormat>
15.          <h:panelGrid columns="2">
16.             #{msgs.contactMeLabel}
17.             <h:outputText value="#{form.contactMe}"/>
18.             #{msgs.bestDayLabel}
19.             <h:outputText value="#{form.bestDaysConcatenated}"/>
20.             #{msgs.yearOfBirthLabel}
21.             <h:outputText value="#{form.yearOfBirth}"/>
22.             #{msgs.languageLabel}
23.             <h:outputText value="#{form.languages}"/>
24.             #{msgs.colorLabel}
25.             <h:outputText value="#{form.colorsConcatenated}"/>
26.             #{msgs.educationLabel}
27.             <h:outputText value="#{form.education}"/>
28.          </h:panelGrid>
29.          <h:commandButton value="#{msgs.backPrompt}" action="index"/>
30.       </h:form>
31.    </h:body>
32. </html>
```

Listing 4–14 select/src/java/com/corejsf/RegisterForm.java

```
1. package com.corejsf;
2.
3. import java.awt.Color;
4. import java.io.Serializable;
5. import java.text.DateFormatSymbols;
6. import java.util.ArrayList;
7. import java.util.Arrays;
8. import java.util.Calendar;
9. import java.util.Collection;
10. import java.util.LinkedHashMap;
11. import java.util.Map;
12. import java.util.Set;
13. import java.util.TreeSet;
14.
15. import javax.inject.Named;
16.    // or import javax.faces.bean.ManagedBean;
17. import javax.enterprise.context.SessionScoped;
18.    // or import javax.faces.bean.SessionScoped;
19. import javax.faces.model.SelectItem;
20.
21. @Named("form") // or @ManagedBean(name="form")
22. @SessionScoped
23. public class RegisterForm implements Serializable {
24.    public enum Education { HIGH_SCHOOL, BACHELOR, MASTER, DOCTOR };
25.
26.    public static class Weekday {
27.       private int dayOfWeek;
28.       public Weekday(int dayOfWeek) {
29.          this.dayOfWeek = dayOfWeek;
30.       }
31.
32.       public String getDayName() {
33.          DateFormatSymbols symbols = new DateFormatSymbols();
34.          String[] weekdays = symbols.getWeekdays();
35.          return weekdays[dayOfWeek];
36.       }
37.
38.       public int getDayNumber() {
39.          return dayOfWeek;
40.       }
41.    }
42.
43.    private String name;
44.    private boolean contactMe;
```

```
45.    private int[] bestDaysToContact;
46.    private Integer yearOfBirth;
47.    private int[] colors;
48.    private Set<String> languages = new TreeSet<String>();
49.    private Education education = Education.BACHELOR;
50.
51.    public String getName() { return name; }
52.    public void setName(String newValue) { name = newValue; }
53.
54.    public boolean getContactMe() { return contactMe; }
55.    public void setContactMe(boolean newValue) { contactMe = newValue; }
56.
57.    public int[] getBestDaysToContact() { return bestDaysToContact; }
58.    public void setBestDaysToContact(int[] newValue) { bestDaysToContact = newValue; }
59.
60.    public Integer getYearOfBirth() { return yearOfBirth; }
61.    public void setYearOfBirth(Integer newValue) { yearOfBirth = newValue; }
62.
63.    public int[] getColors() { return colors; }
64.    public void setColors(int[] newValue) { colors = newValue; }
65.
66.    public Set<String> getLanguages() { return languages; }
67.    public void setLanguages(Set<String> newValue) { languages = newValue; }
68.
69.    public Education getEducation() { return education; }
70.    public void setEducation(Education newValue) { education = newValue; }
71.
72.    public Collection<SelectItem> getYearItems() { return birthYears; }
73.
74.    public Weekday[] getDaysOfTheWeek() { return daysOfTheWeek; }
75.
76.    public SelectItem[] getLanguageItems() { return languageItems; }
77.
78.    public SelectItem[] getColorItems() { return colorItems; }
79.
80.    public Map<String, Education> getEducationItems() { return educationItems; }
81.
82.    public String getBestDaysConcatenated() {
83.       return Arrays.toString(bestDaysToContact);
84.    }
85.
86.    public String getColorsConcatenated() {
87.       StringBuilder result = new StringBuilder();
88.       for (int color : colors) result.append(String.format("%06x ", color));
89.       return result.toString();
90.    }
91.
```

```
92.    private SelectItem[] colorItems = {
93.       new SelectItem(Color.RED.getRGB(), "Red"), // value, label
94.       new SelectItem(Color.GREEN.getRGB(), "Green"),
95.       new SelectItem(Color.BLUE.getRGB(), "Blue"),
96.       new SelectItem(Color.YELLOW.getRGB(), "Yellow"),
97.       new SelectItem(Color.ORANGE.getRGB(), "Orange", "", true) // disabled
98.    };
99.
100.   private static Map<String, Education> educationItems;
101.   static {
102.      educationItems = new LinkedHashMap<String, Education>();
103.      educationItems.put("High School", Education.HIGH_SCHOOL); // label, value
104.      educationItems.put("Bachelor's", Education.BACHELOR);
105.      educationItems.put("Master's", Education.MASTER);
106.      educationItems.put("Doctorate", Education.DOCTOR);
107.   };
108.
109.   private static SelectItem[] languageItems = {
110.      new SelectItem("English"),
111.      new SelectItem("French"),
112.      new SelectItem("Russian"),
113.      new SelectItem("Italian"),
114.      new SelectItem("Esperanto", "Esperanto", "", true) // disabled
115.   };
116.
117.   private static Collection<SelectItem> birthYears;
118.   static {
119.      birthYears = new ArrayList<SelectItem>();
120.      // The first item is a "no selection" item
121.      birthYears.add(new SelectItem(null, "Pick a year:", "", false, false, true));
122.      for (int i = 1900; i < 2020; ++i) birthYears.add(new SelectItem(i));
123.   }
124.
125.   private static Weekday[] daysOfTheWeek;
126.   static {
127.      daysOfTheWeek = new Weekday[7];
128.      for (int i = Calendar.SUNDAY; i <= Calendar.SATURDAY; i++) {
129.         daysOfTheWeek[i - Calendar.SUNDAY] = new Weekday(i);
130.      }
131.   }
132.}
```

Listing 4–15 select/src/java/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
backPrompt=Back
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 4–16 select/web/resources/css/styles.css

```
.emphasis {
   font-style: italic;
   font-size: 1.3em;
}
.disabled {
   color: gray;
}
.selected {
   font-weight: bold;
}
```

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 generated for a component, `h:message` shows only the last one.

NOTE: When you use JSF 2.0 and your project stage is set to `Development`, then an `h:messages` child is automatically added to your page (provided you didn't add one yourself).

The `h:message` and `h:messages` tags share many attributes. Table 4–27 lists all attributes for both tags.

Table 4–27 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 4–27 Attributes for h:message and h:messages (cont.)

Attributes	Description
for	The id of the component for which to display the message (h:message only).
globalOnly	Instruction to display only global messages—applicable only to h:messages. Default is 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	Basic attributes.[a]
style, styleClass, title, dir JSF 1.2, lang JSF 1.2	HTML 4.0.[b]

a. See Table 4–5 on page 107 for information about basic attributes.
b. See Table 4–6 on page 110 for information about HTML 4.0 attributes.

The majority of the attributes in Table 4–27 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 have

also set `showDetail` and `showSummary` to `true`, the message's detail will be wrapped in a tooltip that is shown when the mouse hovers over the error message.

Now that we have a grasp of message fundamentals, we 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 its 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.

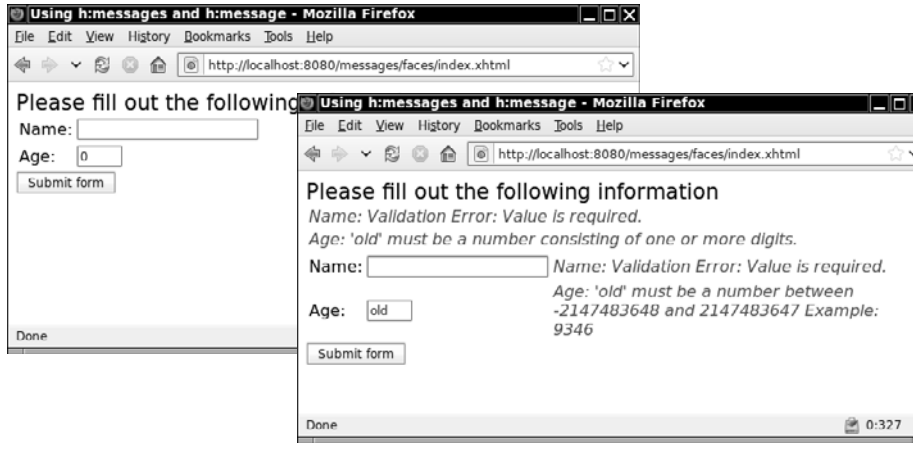


Figure 4–9 Displaying messages

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"
      value="#{user.name}" required="true" label="#{msgs.namePrompt}"/>
   <h:message for="name" errorClass="errors"/>
      ...
   <h:inputText id="age"
      value="#{form.age}" required="true" label="#{msgs.agePrompt}"/>
   <h:message for="age" errorClass="errors"/>
   ...
</h:form>
```

Note that the input fields have `label` attributes that describe the fields. These labels are used in the error messages—for example, the `Age:` label (generated by `#{msgs.agePrompt}`) in this message:

```
Age: 'old' must be a number between -2147483648 and 2147483647 Example: 9346
```

Both message 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;
   color: red;
}
```

We have 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 that shown in Figure 4–10.

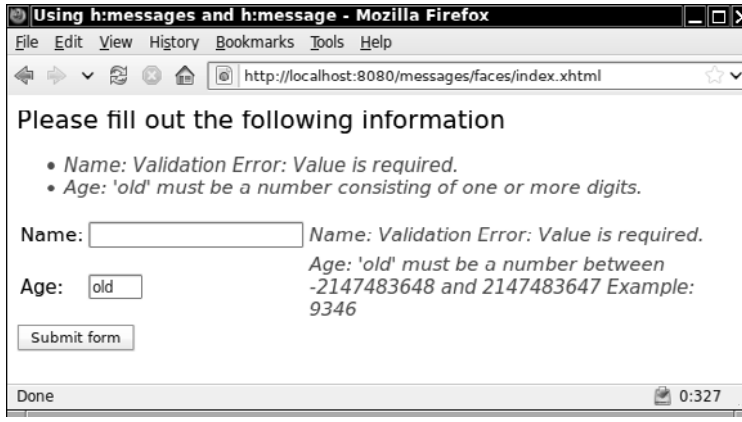


Figure 4–10 Messages displayed as a list

The `list` layout encodes the error messages in an unnumbered list (whose appearance you can control through styles).

CAUTION: In JSF 1.1, the "`list`" style placed the messages one after the other, without any separators, which was not very useful.

Figure 4–11 shows the directory structure for the application shown in Figure 4–9. Listings 4–17 through 4–19 list the JSF page, resource bundle, and stylesheet for the application. For this example, we added `getAge` and `setAge` methods to the `UserBean` class.

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.

Figure 4–11 Directory structure for the messages example

Listing 4–17 messages/web/index.xhtml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core" xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>#{msgs.windowTitle}</title>
      <h:outputStylesheet library="css" name="styles.css"/>
   </h:head>
   <h:body>
      <h:form>
         <h:outputText value="#{msgs.greeting}" styleClass="emphasis"/>
         <br/>
         <h:messages errorClass="errors" layout="table"/>
         <h:panelGrid columns="3">
            #{msgs.namePrompt}:
            <h:inputText id="name" value="#{user.name}" required="true"
                         label="#{msgs.namePrompt}"/>
```

```
            <h:message for="name" errorClass="errors"/>
            #{msgs.agePrompt}:
            <h:inputText id="age" value="#{user.age}" required="true"
                         size="3" label="#{msgs.agePrompt}"/>
            <h:message for="age" errorClass="errors"/>
          </h:panelGrid>
          <h:commandButton value="#{msgs.submitPrompt}"/>
        </h:form>
    </h:body>
</html>
```

Listing 4–18 messages/src/java/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 4–19 messages/web/resources/css/styles.css

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

Conclusion

You have now seen all HTML tags in the standard library with the exception of the tags used for tables, which are covered in Chapter 6. In the next chapter, you will learn how to use the Facelets tags.

Index

#{...} delimiter, 38
${...} delimiter, 38
[ ] notation, 64
! (or not) operator, 69

A

accept attribute, 110
acceptcharset attribute, 110
Accept-Language value, 43
Access control application, 525–531
Access control information, 520
Action(s), compared to action listeners, 312
action attribute
  ActionSource2, 441
  of a button, 6, 73
  h:commandButton, 135
  h:commandLink, 135
  as method expression, 75
  requiring a method expression, 70
Action events, 29, 306, 312–320
Action listener classes, 319
Action listeners
  adding, 312, 459
  attaching
    to buttons, 80
    to links, 138
    to login component's submit button, 364
  compared to actions, 312
  invoking, 313–314
  passing event objects, 327
Action methods
  example of, 75
  executing, 31
  passing desired locale, 325
  passing row items to, 225–226
  returning null, 75
  roles of, 80
  setting locales, 324
Action sources, 306
actionListener attribute
  of ActionSource, 441
  h:button, 135
  h:commandButton, 135
  h:commandLink, 135
  h:link, 135
  requiring a method expression, 70
actionMethod attribute, 354
actionSource composite tag, 349, 364
ActionSource interface, 421
ActionSource2 interface, 421

addOnError function, 403
addOnEvent function, 403
Address field, changing, 86–87
ADF Faces component set, 548
ADF Faces components, in JDeveloper, 15
Ajax, 21–24, 388–389
  adding to custom components, 473–474
  attaching to an input, 389
  calls, associating with events, 388
  in composite components, 409–416
  echoing output, 390
  embedding into custom components, 473
  events, 393
  field validation, 394–396
  functions in JavaScript Library, 403–404
  groups of components, 392–394
  handling errors, 400
  implementing in custom components, 475–484
  passing request parameters, 405–407
  request monitoring, 396–398
  requests
    associating Java functions with, 409
    associating with a group of Ajax components, 392–393
    queueing, 407
  requests compared to HTTP requests, 386
  responses, 400–403
  supported by JSF, 26
  validation monitoring application, 396–398
  welcome message without a page flip, 22
  XML elements for responses, 402
Ajax JavaScript API, in XHTML pages, 404
@all, in execute attribute of f:ajax tag, 391
alt attribute, 110
Alternative renderers, 26
anchor attribute, 119
anchor element, generating, 134, 137–138
Annotations
  defining bean scopes, 51
  life cycle, 58
  O/R mapper translating, 507
  referencing the validator class, 272
  for resource injection, 495–496
  validation, 270
Apache Derby database, 493–495
Apache MyFaces components library, 581–582
Apache Tomahawk, dataScroller component, 569
Apache Tomahawk library, 548
Apache Trinidad library, 548
Application(s)
  accessing external services, 487–544
  analysis of a sample, 15–21
  behind the scenes of, 26–31
  building, 9–11
  configuring, 595–596
  cookies sent by, 52
  deploying, 8, 11–12
  ingredients of, 7
  locale for, 43–44
  registering symbolic ID with, 279
  sample illustrating essential features of beans, 45–51
  testing tools for developing, 604–605
Application configuration file, 80, 85–86
Application scope, 51, 54
Application servers
  authenticating users on, 519
  compatible with Java EE 6, 10
  realms supported, 522
  starting, 11
Application source, directory structure of, 9
applicationScope predefined object, 68
@ApplicationScoped annotation, 51
Application-scoped bean, marking as eager, 59
Apply Request Values phase, 30, 247, 307, 321
Arithmetic operators, 69
Array of names, 207, 208
ArrayDataModel instance, 236
ArrayList, displaying, 47
Arrays, with f:selectItems, 157–158
Asynchronous JavaScript with XMLHttpRequest. *See* Ajax

atMax attribute, 443–444
atMin attribute, 443–444
Atomic methods, 188
Attacks
  cross-site scripting, 131, 134
  SQL injection, 492
attribute composite tag, 348
Attribute map, 104, 422, 442
Attribute value, converting to an integer, 429
Attributes
  attaching JavaScript to components, 308
  basic for HTML tags, 107–110
  common for HTML tabs, 107–115
  components and, 426
  in the core library, 104
  requiring method expressions, 70
  specifying name and scope of a managed bean, 17
  supplying to converters, 289–290
  types of, 354–355
attributes element, 402
Authentication
  basic, 521–522
  error screen, 500
  test application, 525–531
Autocommit mode, 493
autocomplete attribute, 124
Autocomplete composite component, 409–416

B

Backing beans
  existing tags with, 338–341
  for web forms, 38–39
Backing components, 373–378
Basic authentication, 521–522
Bean(s). *See also* CDI (Contexts and Dependency Injection) beans; Java bean; Managed beans
  annotations for naming, 7
  application code contained in, 16
  configuring, 56–63
  creating and manipulating without programming, 33
  defined, 16
  defining, 46, 59
  definition of, 33–36
  life cycle annotations, 58
  managing, 17
  managing user data, 7
  properties of, 36–37
  separating presentation and business logic, 33
Bean classes, 36
Bean methods, 294–295
Bean name, deriving, 35
Bean scopes, 16, 51–56
Bean Validation Framework
  annotations in, 270
  compared to page-level validation, 271
  directory structure of example, 273
  extensibility of, 272
  JSF 2.0 integrating with, 270
  validator from, 266
Bean validators, validation groups for, 264
@BeanProperty annotation, 606
beans.xml file, 7
begin attribute, 398
begin request status, 397
Behaviors, attaching to components, 385, 389
bgcolor attribute
  h:dataTable, 210
  h:panelGrid, 116
Binary data
  generating directly from JSF, 566
  producing, 559–568
Binding, value attribute, 161–162
binding attribute, 39
  as a basic attribute, 107
  f:param, 104
  in JSF 1.2, 253, 265
  specifying with a value expression, 109
  ui:component, 198
Black-box test automation frameworks, 604
Bookmarkable GET request, 134
Bookmarkable links, 88, 92–96
boolean properties, prefixes, 37
border attribute, 150
  h:dataTable, 210
  h:panelGrid, 116
  HTML pass-through, 110

Brackets, instead of dot notation, 64
Branching behavior, 75
British flag, link for, 324
Browser(s)
choosing the locale, 43
monitoring traffic with server, 601–602
pointing to default URLs, 12
rendering multiple select menus, 153
Browser language, switching to German, 47
Browser screen, JSF page for each, 17
Bundle files, supplying localized, 41
Bundle name, retrieving, 282
Business logic
separating from presentation, 33
separating from user interface logic, 314
of a Web application, 15
Buttons
attaching action listeners to, 80
firing action events, 312
tags supporting, 134–136
using, 136–141
Bypassing, validation, 266–267

C

Cacheable pages, 89
Cagatay Civici's client-side validation package, 595
Canada provinces, pop-up window listing, 574
Cancel button, demonstrating validation bypass, 267
`caption` facet, 213
`captionClass` attribute
`h:dataTable`, 210
`h:panelGrid`, 116
`captionStyle` attribute
`h:dataTable`, 210
`h:panelGrid`, 116
`card` property, of the `PaymentBean`, 279
`cc` predefined object, 68
CDI (Contexts and Dependency Injection), 517
CDI (Contexts and Dependency Injection) beans, 39–40
injecting, 56
`cellpadding` attribute
`h:dataTable`, 210
`h:panelGrid`, 116, 117
`cellspacing` attribute
`h:dataTable`, 210
`h:panelGrid`, 116, 117
`charset` attribute, 110
Chart image, tag creating, 559–561
`check` method, of the `user` bean, 25
Checkboxes
rendering wired to a `boolean` bean property, 146
selection example using, 164–171
tags for, 148–150
`checkForm` function, 369
`checkPassword` function, 120–121
Child components
managed by `UIComponent`, 421
placing name/value pairs in, 104
rendering supplied facets as, 366
setting day, month, and year values of, 376
Child elements, inside a JSF component, 198
Children
component processing, 460
inserting in composite components, 366–367
`choice` format, 43
`c:if` or `c:choose` construct, 581
Class files, directory holding, 10–11
Classes
implementing fake backend activities, 604
receiving system events, 330–331
requirements to be entities, 507
Click behavior, attached to spinner, 481
Client, saving state on, 468
Client behaviors, decoding, 481
Client devices, JSF framework interacting with, 24
`ClientBehaviorHolder` interface, 479, 480
Client-side credit card validation, 589
Client-side scripts, 115
Client-side validation, 368, 588
Client-side validation tag, 588–595
`close` method, in a `finally` block, 490
Coalescing, events, 408

Code, for this book, 10
Collections, with f:selectItems, 157–158
collectionType attribute, 147, 163
Colors, selecting, 148
cols attribute, 124, 126
Column headers, 212
columnClasses attribute
  h:dataTable, 210
  h:panelGrid, 116
Columns, CSS classes applied to, 117
columns attribute, h:panelGrid, 115–116
com.corejsf.util package, 283
Command links, 141–144. *See also* Links
commit method, 493
Committed transaction, 493
Commons file upload library, 548
Commons Validator, 595
complete attribute, of a data object, 398
complete request status, 397
Component(s), 3
  attaching Ajax behavior to, 389
  building
    Ajax, 473–474
    custom, 419
  developing reusable, 457–468
  executing, 388, 389
  exposing, 361
  finding more, 547–548
  paired with renderers, 107
  referencing from other tags, 108
  relationships between multiple, 295–297
  specifying multiple listeners for, 320
  in tables, 218–221
  tags exposing composite, 364
  validating a group of, 331–332
Component author, 348
Component classes
  implementing, 420–423
  responsibilities of, 420
  standard, 421
Component developers, 25–26
Component hierarchy, 422
Component ID, 439
Component identifiers, aliasing, 362–363
Component libraries, 548
Component objects, request values in, 247–248
component predefined object, 68
Component tag, generating JavaScript code, 588
Component tree
  displaying in Debug Output, 199, 200
  retrieving or constructing, 29
  of the sample application, 26–27
Component type, mapping, 434
Component/renderer pair, 205
Composed Method pattern, 188
Composite components
  adding Java code to, 373
  Ajax in, 409–416
  configuring, 353–354
  facets of, 365–366
  implementing, 352–353
  interfaces and implementations, 349
  in JARs, 382
  JavaScript with, 368–369
  in JSF 2.0, 198, 347–348
  localizing, 359–360
  passing managed beans to, 358
  tags exposing, 364
  using, 350–352
Composite Components library, 101
Composite date application, 379–382
Composite date component, 373–378
Composite expressions, 69–70
Composite login sample application, 369–373
Composite tab library, 348–350
composite:actionSource tag, 363
composite:attribute tag, 355
composite:editableValueHolder tags, 360–363
composite:facet tag, 366
composite:insertChildren tag, 366, 367
composite:insertFacet tag, 366
composite:renderFacet tag, 366
composite:valueHolder tag, 363
Concurrent access, transactions for, 493
Conditional navigation, 99
CONFIG_FILES initialization parameter, 59
Configuration file, 282, 291
Configuration parameters, for applications, 595–596
Connection, closing, 488
Connection leaks, plugging, 490–491

Connection object, 487, 488
Connection pool, 489, 496
Connections
  closing properly, 490–491
  management of, 489
ConstraintValidator interface, 272
Constructors, 237
Container-managed authentication, 519
Container-managed resource, accessing, 495–496
Container-managed security application, 525–531
Content, defining pieces of, 188
Context
  beans bound to, 39
  external compared to real, 432–433
Context parameter, reading, 595
Contexts and Dependency Injection beans. *See* CDI (Contexts and Dependency Injection) beans
Controller, JSF implementation as, 25
Controller servlet, processing requests, 306
Conversation scope, 51, 54–55
@ConversationScoped annotation, 55
Conversion error messages
  displaying, 254–255
  standard, 257–258
"Conversion error occurred"generic message, 258
Conversion errors
  actions taken, 253–259
  reporting, 280–281
Conversion process, strings into types, 248
convertClientID method, 440
Converter(s), 275
  associated with a component, 452
  creating, 453
  method for setting, 433
  programming with custom, 275–297
  registering as default, 279
  in a resusable library, 282
  specifying, 279–285
  standard, 249–262
  supplying attributes to, 289–290
converter attribute
  adding to the component tag, 252
  attaching a converter to a component, 109
  as a basic attribute, 107, 108
  compared to validator, 295
  specifying converter ID, 279
Converter interface, 275
Converter object, 253
Converter sample, 259–262
ConverterException, throwing, 275, 280, 281
converterMessage attribute
  as a basic attribute, 108
  of the component, 259
cookie predefined object, 68
Cookies, tracking sessions, 52
coords attribute, HTML pass-through, 110
Core library, 101, 102–105
Core tags
  in the core library, 102–105
  defined by JSF, 17
  representing objects added to components, 103
corejsf-examples directory, 10
corejsf:planet tag, 195–198
corejsf:spinner, 423
createMetaRuleset method, 444
Credit card numbers
  custom converter for, 276
  verifying and generating, 271
Credit card validation, 588, 589
Cross-site scripting attacks, 131, 134
CSS classes
  applying to messages, 172–173
  specifying, 215–218
    by column, 215–216
    for different table parts, 117
    for images, 355–356
    by rows, 216–217
CSS layout, 115
CSS styles
  applying to messages, 172–173
  attributes, 112
  rendering components, 112
  specifying for column headers and footers, 213
  for a tabbed pane, 459
cumulativeOffset function, 406
currencyCode attribute, 251

currencySymbol attribute, 251
Custom components, 25
  adding Ajax functionality, 473–474
  building, 419
  duplicating code for converters, 440
  self-contained Ajax in, 475–484
Custom converter
  classes, 275–278
  code for, 276, 277–278
  defining, 286, 288
  error message, 259
  programming, 254
  sample application, 286–289
  tags, 297–303
Custom font spinner, 473–474
Custom scopes, 51, 56
Custom tags, 195–198
Custom validator(s)
  classes, 290
  example, 291–294
  registering, 290–291
  tags, 297–303
  writing, 272

D

Data access objects, 514
Data comparator, 237
Data conversion, rules for, 25
Data model listener, 242
data object, 398
Data set, showing, 568–573
Data source
  configuring, 495–506
  using, 487
Data table, pager as a companion to, 568
Database, accessing with JPA, 509–513
Database application, complete example, 499–506
Database connection(s)
  management of, 489
  prepared statement tied to, 492
  specifying in GlassFish, 497
Database connection pool, 489
Database example, directory structure of, 230
Database integrity, transactions for, 493
Database resource
  configuring in GlassFish, 496–498
  configuring in Tomcat, 498–499
Databases query, results of, 228–232
DataModel API, 236–237
DataModel classes
  constructors in, 237
  getRowData method, 234
  getRowIndex method, 233–234
Date picker, 473
Date value, 377, 378
Dates, conversion of, 249–253
dateStyle attribute, 252
Date-valued property, 378
Debug component
  adding to the component tree, 198
  directory structure of, 352
  implementing, 352–353
  using, 350
Debug Output window, 198–200
Debugging
  h:message tag useful for, 256
  parameter adding support for, 21
  a stuck page, 602–603
  tools, 329
decode method
  calling, 432
  calling setSubmittedValue method, 429
  of the file upload component, 550
  of PageRenderer, 568
  of the spinner, 427–428
  trapping invalid inputs, 430
Decoding process, 28
Decorators, 193
Default Ajax events, 393
Default tag handler, 441–442
delete element, for Ajax responses, 402
Derby. *See* Apache Derby database
Design pattern, advocated by Smalltalk, 188
Detached entity object, 514
Detail, in a message, 172
Detail error message, 255
Development environments, 13–15, 39
Development stage, error message in, 582
DHTML events, 114–115
dir attribute
  h:dataTable, 210
  HTML pass-through, 110
Directory structure, 8–9

disabled attribute
f:ajax tag, 391
HTML pass-through, 111
disabledClass attribute, 150
DiskFileUpload object, 551
Document Object Model (DOM) elements, 386
doPopup function, 573, 575
Dot notation, 64
double value, 263
Drop-down lists, single-select menus as, 153
DRY (Don't Repeat Yourself) principle, 181
Dynamic HTML (DHTML), 114–115
Dynamic navigation, 74–75
Dynamic target view IDs, 99

E

eager application-scoped bean, 59
eager attribute, 54
echo component, on the client, 390
echo output text, 390
Eclipse
Groovy Eclipse Plugin, 607–608
importing a project into, 13
monitoring traffic between browser and server, 601–602
editable property, 223
Editable value holders, 306
editableValueHolder composite tag, 349, 364
EditableValueHolder interface, 421, 433
@EJB annotation, 514
EJB container, 517
Element object, 397
Email message, application sending, 535
Embedded mode, application servers in, 604
empty operator, 69
emptyResponse value, 400
enabledClass attribute, 150
Encapsulation, 181
encode method, 569
encodeBegin method
described, 426
for encoding markup, 424
setting day, month, and year, 376
of UISpinner, 425
encodeChildren method
calling, 439
for encoding markup, 424
getRendersChildren and, 461
invoking, 426
overriding, 460
encodeEnd method, 424, 461, 462
encodeTab method, 462
Encoding process, 27
endElement method, 425, 427
English locale, 142
Entities, in JPA, 507
Entity manager
with entity objects, 508
injecting in session beans, 606
obtaining, 509
Entity manager factory, 509
Error(s)
handling in Ajax, 400
handling of image paths, 25
Error display, 585
error element, 402
error facet, 367
Error facets, adding to login component, 365–366
Error functions, 400
Error messages
changing text of standard, 256–258
displaying, 254–256
getting from resource bundles, 281–284
placing data in request scope, 53
showing in a different color, 255
versions of, 255
Error pages, customizing, 582–587
Error sample application, 585–587
Error screens, for a database application, 499, 500
errorClass attribute, 172
error-page mechanism, servlet exception attributes, 584
error-page tag, 583–584
errors CSS class, 175
errorStyle attribute, 172
escape attribute
f:selectItem, 154
h:outputFormat, 132

h:outputText, 132
as unique, 131
use of, 134
event attribute, f:ajax tag, 391
Event handlers, registering with components, 305
Event handling
example demonstrating, 338–345
in the JSF life cycle, 306–307
Event listeners
affecting the JSF life cycle, 307
attaching, 331
collection managed by UIComponent, 422
Events
coalescing, 408
executed on the server, 306
JSF life cycle and, 306–307
kinds of, 305
naming convention for, 392
queueing, 407–408
specifying in Ajax, 393
ExceptionQueuedEvent, 330
execute attribute, of f:ajax tag, 391
execute components, in Ajax, 23
execute key, in Ajax, 405
execute list, in Ajax, 31
Execute portion, of the JSF life cycle, 387
executeQuery method, 230, 488
executeUpdate method, 488
Expression language (EL)
adding a function to, 599–600
compared to value expressions, 38
context object, 452
expressions in, 19
extending, 596–599
not part of JSF, 71
in the outcome attribute, 91
predefined objects in, 68
syntax, 63–71
syntax for method expression types, 71
Expressions, evaluation of, 38
extension composite tag, 349
Extension mapping, 20
External context, 432–433
External context object, 595
External renderer, 438–441
External services, accessing, 487–544
ExternalContext class, 595

F

Facelets, 17, 179–180
Facelets error display, 585
Facelets library, 101
Facelets page, 17
Facelets tag library file, 599, 600
Facelets tags
categories of, 179
limitations of, 198
summary of, 180
templating with, 181–195
FACELETS_SKIP_COMMENTS context parameter, 201
.faces extension, 20
/faces prefix, 20
Faces servlet, 20
@FacesComponent annotation, 374
faces-config.xml file
associating converter ID with, 279
configuration information in, 7, 58
navigation elements in, 96
system event listener in, 331
in WEB-INF directory, 41
facesContext predefined object, 68
@FacesConverter annotation, 279
@FacesRenderer annotation, 438
FacesTrace, 603
Facet(s)
of a composite component, 365–366
in the core library, 104
rendering, 461
specifying column headers and footers, 212–213
specifying tabbed pane content, 458
Facet components, 421
facet composite tag, 349
Facet map, 104
f:actionListener tag
adding action, 318–320
in the core library, 102
f:ajax tag, 389–392
adding a listener, 409
attaching behaviors to components, 385, 416
attributes, 391–392
attributes as strings, 404
in the core library, 103
for custom components, 479

f:ajax tag *(cont.)*
limited functionality of, 403
nesting, 393–394
onevent attribute, 396
supporting, 473
fatalClass attribute, 172
fatalStyle attribute, 172
f:attribute tag
attaching a converter, 289
in the core library, 102, 104
setting a component's attribute with, 326–327
f:convertDateTime converter, 250
f:convertDateTime tag, 102
f:converter tag, 102, 279
f:convertNumber converter, 250
f:convertNumber tag, 102, 251
f:event tag, 102
f:facet tag, 102, 104, 340
Field validation, in Ajax, 394–396
Field value, from browser to model bean, 248
File upload application, 551–557
File upload object, 551
File uploads, supporting, 548–557
FileUploadRenderer class, 554–556
finally block, 490
findComponent method, 108–109, 296
findCreditCardValidators method, 588–589
Firebug, viewing Ajax responses, 400–401
first attribute, h:dataTable, 210, 211
Flags sample application, 142–144
Flash object, 88
flash predefined object, 68
f:loadBundle action, 41
f:loadBundle tag, 53, 103
f:metadata tag, 103
Font spinner, 473, 476
Font spinner renderer, 475
footerClass attribute
h:column, 211
h:dataTable, 210
h:panelGrid, 116
Footers, 212, 214
for attribute
f:actionListener, 364
h:message tags, 173
Form, containing a hidden field and invisible link, 575
@form, in execute attribute, 391
Form controls, naming, 121
Form data, 28
Form elements, accessing, 120
Form ID, obtaining, 454
Form login configuration, 520
Form-based authentication, 520
f:param tag
attaching parameter to a component, 325–326
in the core library, 102, 104
embedding in an h:link tag, 139
overriding view parameters, 91
placeholders as child elements, 42
f:phaseListener tag, 102
fragment attribute, 135
frame attribute
h:dataTable, 210
h:panelGrid, 116, 117
Frameworks, 3. *See also* Bean Validation Framework; JSF framework; Seam framework; Selenium test automation framework; Sitemesh framework
from-action element, 98
from-view-id element, 76, 97–98
f:selectItem tag, 153–155
in the core library, 103
in h:selectManyCheckbox, 148
populating radio buttons, 149
f:selectItems tag, 155–157
attributes, 156–157
collections and arrays, 157–158
in the core library, 103
in h:selectManyCheckbox, 148
maps with, 158–159
using a single, 459
values for, 460
var attribute in, 158
f:setPropertyActionListener tag, 102, 327–328
f:subview tag, 103
Function(s)
adding to JSF expression language, 599–600
calling, 66–67

Functionality, defining common, 187
f:validateBean tag, 103, 264
f:validateDoubleRange tag, 102, 263
f:validateLength tag, 102, 263
f:validateLongRange tag, 102, 263
f:validateRegex tag, 102, 263
f:validateRequired tag, 102, 263, 264
f:validator tag
  attaching a validator to a component, 264
  in the core library, 102
  specifying validator ID in, 291
f:valueChangeListener tag
  adding change listeners, 318–320
  adding one or more listeners with, 442–443
  in the core library, 102
f:verbatim tag, 103, 141
f:view element, 44
f:view tag
  in the core library, 103
  enclosing a JSF page in, 329
  enclosing a page in, 333
  setting the locale, 44
f:viewParam tag, 103

G

German flag, link for, 324
German locale, 48
get method, 36
GET method, posting forms with, 119
GET requests
  as idempotent, 89
  links, 90–91
  support for, 89–92
getAjaxScript method, 475–476
getAnswerComponent method, 39
getAsObject method, 276
getAsString method, 276
getAttributes method, 425, 442
getClientID method, 425, 426
getConnection method, 489
getConvertedValue method, 377, 440
getConverter method, 452
getELContext method, 452
getExternalContext method, 432
getFacet method, 461
getFlash method, 88
getGreeting method, 23
getIncrementedValue method, 428–429
getInitParameter method, 595
getMessage method, 283
getName method, 19
getNames method, 233, 236
getRendersChildren method, 424, 439, 460, 461
getRequestParameterMap method, 433
getResourcesAsStream method, 596
getRowData method, 234, 236
getRowIndex method, 233–234
get/set methods, 16
getSkipOutcome method, 90
getStackTrace method, 584
getString method, 488
getType method, 453
getValue method, 425
getValueExpression method, 452
GlassFish
  Apache Derby database, 493–495
  configuring
    connection pool, 496
    database resource, 496–498
    realm, 522–524
  cookies representing current user, 525
  deployment directory, 12
  log file, 12, 501
  specifying mail session parameters, 532
  starting, 11
  welcome page, 12
Global phase listeners, 328–329
globalOnly attribute, 173
GMail account, 533
graphicImage tag, 112
greeting property, 22–23
Groovy Eclipse Plugin, 607–608
Groovy programming language, 607–608
groupingUsed attribute, 251
GUI builder, 34

H

HashMap, 158
h:body tag, 105, 118
h:button tag, 106, 134
  attributes, 135–136
  examples of, 137

h:button tag *(cont.)*
issuing GET requests, 90
outcome attribute, 90
h:column tags, 106
attributes, 211
in the body of h:dataTable tags, 206
h:commandButton tag, 106, 134
action attribute, 19, 90
attributes, 135–136
converting to HTML, 27
examples of, 137
handling of image paths, 137
h:commandlink component, 104
h:commandLink tag, 106, 134
attributes, 135–136
examples of, 137–138
placing children, 138
h:dataTable tag, 106
attributes for, 210–211
attributes specifying styles, 215–218
creating an HTML table, 205–206
data as row oriented, 209
JSF page using, 229
reusing column classes, 216
sorting or filtering tables with, 234
wrapping in an HTML div, 242
wrapping objects in a model, 232
Header facets, 365–366
header predefined object, 68
headerClass attribute
h:column, 211
h:dataTable, 210
h:panelGrid, 116
headerValues predefined object, 68
Helper servlet, 559
h:form tag, 27, 105, 118–119
h:graphicImage tag, 106
attributes, 132–133
examples of, 133–134
handling of image paths, 137
h:head tag, 105, 118
Hibernate Validator JAR files, 271
Hidden fields
encoding, 462
placed after all tabs, 462
support for, 127
Hidden input field, 296
hideNoSelectionOption attribute, 147
h:inputHidden tag, 105, 124–125, 127
h:inputSecret tag, 105
attributes for, 124–125
converting to HTML, 27
uses of, 125–126
h:inputText tag, 105
attributes for, 124–125
converting to HTML, 27
uses of, 125–126
h:inputTextarea tag, 105
attributes for, 124–125
examples of, 126
specifying one long string for, 127
h:link tag, 106, 134
attributes, 135–136
embedding an f:param tag in, 139
examples of, 137–138
issuing GET requests, 90
h:message tags, 106, 172
attributes, 172–173
displaying validation errors, 265
showing details but not summaries, 176
h:messages tags, 106, 172
adding, 254
attributes, 172–173, 255
displaying all messages, 174–175
displaying validation errors, 265
layout attribute, 175
listing of all messages, 256
showing summaries but not details, 176
Hot fixes, 14
hotkey attribute, 199
h:ouputFormat tag, 106
h:ouputLabel tag, 106
h:ouputLink tag, 106
h:ouputText tag, 106, 131
h:outputFormat tag, 42
attaching an f:param tag to, 326
attributes, 132
escape attribute, 131
formatting compound messages, 132
h:outputImage element, 557
h:outputLink tag
attributes, 139
examples of, 139–140
generating an HTML anchor element, 134, 139

h:outputScript tag, 105
  accessing JavaScript, 397
  loading JavaScript with, 368
h:outputStylesheet tag, 105
h:outputText tag
  attributes, 131–132
  escape attribute, 131
  examples of, 133–134
  generating HTML input element, 134
h:panelGrid tag, 106
  attributes, 116
  implementing a tabbed pane, 340
  using HTML tables for layout, 115
  using with h:panelGroup, 117
h:panelGroup tag, 106, 214
  attributes for, 117–118
  with h:panelGrid, 117
hreflang attribute, 111
h:selectBooleanCheckbox tag, 106, 145, 146, 148
h:selectMany tags
  value attribute, 161–162
  value of a, 163
h:selectManyCheckbox tag, 106, 146
  attributes, 150
  creating a group of checkboxes, 148
  example, 145
  generating an HTML table element, 149
  in selection example, 164
h:selectManyListbox tag, 106, 145, 146, 151–152, 159, 162–163
h:selectManyMenu tag, 106, 146, 152–153
h:selectOne tags, 161–162
h:selectOneListbox tag, 106, 145, 146, 151
h:selectOneMenu components, 373–378
h:selectOneMenu tag, 106, 146, 152
  onchange attribute, 308
  in selection example, 164
h:selectOneRadio tag, 106, 146
  attributes, 150
  example, 145
  generating an HTML table, 149
  implementing radio buttons, 149
  in selection example, 164
  value attribute of, 149
HTML
  attributes, supporting scripting, 114
  distinction between menus and listboxes, 153
  elements
    accessing client identifiers of, 368
    unintentional generation of, 134
  encoding in UISpinner, 424–425
  form, compared to JSF page, 18
  form tag, 119
  generated by the spinner, 425
  library, 101
  markup, generating, 131
  output, of the rendering process, 27–28
  pages, rendering, 27
  specification, accessing, 111
  table, 205–206
  tags
    advanced aspects of, 338
    categories of, 105
    in JSF, 105–107
    with a jsfc attribute, 18
    in the login page file, 5
    overview of, 105–115
HTML 4.0, pass-through attributes, 110–111
HTMLUnit test automation framework, 604
HTTP 302 (Moved temporarily) status, 88
HTTP error, displaying, 584
HTTP monitoring, in Eclipse, 601
HTTP protocol, as stateless, 52
HTTP redirect, 86
HTTP requests, 386, 407
httpError value, 400
Hyperlinks, tabs encoded as, 462

I

ICEfaces open source library, 548
Icons, attributes of, 353
icon.xhtml file, 353–354
ID(s)
  associating with a converter, 279
  containing row numbers, 226
  of converters, 252
  placing restrictions on, 440
id attribute, 22
  assigning names to HTML elements, 120

id attribute *(cont.)*
as a basic attribute, 107
described, 108–109
expressions accepted, 105
f:param, 104
specifying for forms, 119
ID strings, assigning, 27
ID values, generated by JSF implementation, 121
Identifiers, adding to URLs, 52–53
Identities, switching, 525
IDEs, 13, 14
if element, 99
ij interactive scripting tool, in Derby, 494
Image(s)
displaying, 131–134
including from a library, 112
image attribute
h:button, 135
h:commandButton, 135
of an icon, 354
requiring, 355
Image map, showing, 557–559
img element, 132
immediate attribute, 322
adding to h:commandLink tag, 324
for a command, 267
of a component, 321
f:ajax tag, 391
h:button, 135
h:commandButton, 135
h:commandLink, 135
h:inputHidden, 124–125
h:inputSecret, 124
h:inputText, 124
h:inputTextarea, 124
h:link, 135
of an icon, 356
Immediate command components, 323–324
Immediate components, 320–321, 322
Immediate input components, 321–323
implementation composite tag, 348
Implicit objects, 67, 68
include directive, 340
Indexed properties, 37
index.xhtml page
reading, 26
with view parameter and h:link tag, 93–94
infoClass attribute, 173
infoStyle attribute, 173
Initial term, resolving, 67–69
initParam predefined object, 68
Injection process, 56. *See also* Resource injection
Input, associating an Ajax call with, 388
Input components
for cells being edited, 223
exposing, 361
firing value change events, 307
input element, 149
Input field(s)
Ajax request for validating, 386
conversion options for, 253–254
displaying, 47
linking to object properties, 5
Input field values, bound with name user, 19
insert element, 402
insertChildren composite tag, 349
insertFacet composite tag, 349
Integer converter, 377
Integer index, 65
Integer wrapper type, 61
integerOnly attribute, 251
Interface, of a composite component, 353
interface composite tag, 348
Interface types, 439
Internal error screen, for a database application, 500
Internationalization, 25, 47, 48
INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL context parameter, 264
invalidate method, 53
Invoke Application phase, 31, 312
isAutoExec function, 404
ISO country codes, for locales, 243–244
ISO-639 language codes, 42
isRendered method, 461
Item(s), 153–161
Item descriptions, 154
Item groups, 159–161
Item labels, 153–154, 459
itemDescription attribute, 157
itemDisabled attribute, 154, 157

itemEscaped attribute, 157
itemLabel attribute, 156
itemValue attribute, 157
Iteration status, 218

J
JAR files
  of the JSF implementation, 8
  packaging composite components in, 382
  packaging Facelets tags as, 197
  providing, 434–435
Java, serialization, 469
Java bean, 33
Java Blueprints conventions, 9
Java BluePrints project, 548
Java code, component references in, 108–109
Java Database Connectivity (JDBC)
  API, 487
  database access with, 487–495
  supporting pooling, 489
Java EE 5 tutorial, bookstore6 web application, 557–559
Java EE 6 specification, 10
Java EE application server, 4, 39
Java Persistence Architecture (JPA)
  crash course in, 507–508
  demo application, 509–513
  in a web application, 508–513
Java Quiz application
  directory structure of, 80, 81
  navigation in, 77–86
  transition diagram of, 82
JavaBeans specification, 33
JavaDB. *See* Apache Derby database
javaee subdirectory, 10
java.lang.Object attribute values of type, 354
JavaMail API, 532–537
JavaScript
  attaching to components with attributes, 308
  code accessing fields in a form, 454
  with composite components, 368–369
  encoding to update components, 453–456
  form elements and, 120–123
  libraries required by font spinner, 475
  making an Ajax call to the server, 475
  namespaces, 398–399
JavaScript API, using directly, 416
JavaScript Library
  accessing in XHTML files, 403
  Ajax functions, 403–404
  in JSF 2.0, 403–405
JavaScript sample application, 121–123
JavaScript timer, coalescing Ajax calls, 408
JavaServer Faces. *See* JSF (JavaServer Faces)
java.util.Date, 374
javax.faces.model package
  context parameter setting, 201
  model classes in, 232–233
JAX-WS technology, 538–539
JDBC. *See* Java Database Connectivity (JDBC)
JDeveloper, visual builder in, 14, 15
JDK (Java SE Development Kit) 5.0 or higher, 10
JFreeChart library, 561
JNDI name, for a mail resource, 533
JPA. *See* Java Persistence Architecture (JPA)
JPQL, query in, 508
JSF (JavaServer Faces)
  application example, 4–9
  code for event handling, 4
  component hierarchy, 422
  component-based, 3
  development environment, 13–15
  parts of, 3
  registering standard validators, 291
  with scripting languages, 120–123
JSF 1.0, components in, 347
JSF 1.2, tag libraries, 101
JSF 2.0, 4
  Ajax with, 388–389
  in Facelets, 17
  integrating with Bean Validation Framework, 270
  JavaScript library, 403–405
  life cycles of, 388
  tag libraries, 101
JSF applications. *See* Application(s)

JSF data table component, showing database data, 228–232
JSF expression language. *See* Expression language (EL)
`.jsf` extension, 20
JSF framework
  error messages generated by, 174
  overview of, 24
  services, 24–26
JSF implementation
  controlling beans, 35
  defining converters with predefined IDs, 252–253
  handling events on the server, 308
  initializing JSF code, 26
  locating a bean class, 35
  rendering or decoding table data, 235
JSF library, converters and validators in, 249–269
JSF life cycle, 29–31
  executing, 306
  parts of, 387
JSF pages, 17–19
  accessing column data, 230
  commenting out parts of, 200–201
  compared to an HTML form, 18
  encoding and decoding, 27
  passing to Faces servlet, 20
  for tabbed pane application, 470, 472
  viewing without `/faces` prefix, 20
JSF tags
  adding to XHTML pages, 17
  in the login page file, 5
JSF URLs, format of, 20
`jsfc` attribute, 18
JSFlive cycle, firing phase events, 306
JSFUnit framework, 604
JSFUnit test, 605
JSP
  replaced by Facelets, 179
  syntax, 18
JSR 245 (JavaServer Pages), expression language defined in, 71
JSR 303 implementation, 271
JSTL (JavaServer Pages Standard Tag Library) constructs, 38
JSTL Core library, 101
JSTL Functions library, 66–67, 101

K

`key` elements, for `map-entry` elements, 61

L

`label` attribute, 124, 147, 254
Labels, for menu items, 153–154
`lang` attribute, 111
Language, selecting preferred, 44
Language code, passing from UI to server, 325
`languageCode` attribute, 327
`languageCode` parameter, accessing, 326
`languageCode` property, 328
`layout` attribute
  `h:messages`, 173, 256
  as horizontal or vertical, 150
  `h:panelGroup`, 118
Lazy collection, 514
`left` expression, evaluating in lvalue mode, 64
`LengthValidator` standard message, 265
Libraries, of the `resources` directory, 112
Life cycle. *See also* JSF life cycle
  annotations of beans, 58
  of JSF, 29–31
  skipping the rest of, 323
Lifetimes, of the standard JSF scopes, 56
`LinkedHashMap`, 158
Links
  acting like buttons, 138
  bookmarkable, 88
  changing locales with, 323
  firing action events, 312
  registration, 367
  RESTful, 92, 93
  tags supporting, 134–136
`List` interface, 65
`list` layout, for messages, 175
`List` type, 60–62
Listbox items, grouping, 159–161
Listboxes
  containing completion items, 405–407
  multiselect, 151–152
  selection example using, 164–171
  single-select, 151
  tags for, 151–153
`listener` attribute
  `f:ajax` tag, 392, 409

requiring a method expression, 70
Listener attributes, compared to tags, 318–320
Listeners
attaching, 319, 333
invoking, 329
specifying multiple, 320
Lists, initializing, 60–62
@Local annotation, 606
Local values
converted values stored as, 248
validating, 30
Locale(s)
changing, 308, 323
getting current, 281
ISO country codes for, 243–244
setting for applications, 43–44
setting programmatically, 44
using command links to change, 141
locale attribute
of f:convertDateTime, 252
of f:convertNumber, 251
Locale prefix, obtaining, 113
Locale suffix, adding to a bundle file, 42
localeChanger bean, languageCode property, 328
localeChanger method, languageCode parameter, 325
Localization, 281
Localized resources, 113
Log files
consulting, 12
for database configurations, 501
Logical operators, 69
Logical outcomes, 75
Login button, 22, 23–24
Login component
adding facets to, 365–366
attaching validators, 360–365
client-side validation for, 368
implementing a simple, 357–359
interacting with a managed bean, 356–357
making more accessible, 358–359
properties file, 359–360
with a registration link, 367
sample application, 369–373
Login form, 28
Login information, securely transmitting, 521
login method, 501
Login page
content area of, 190–191
for planets application, 181–182
viewing source of, 28
Login screen
for a database application, 499
file describing, 4–5
page defining, 7
Login view
header for, 188–189
sidebar for, 189–190
login.war file, 12
login.xhtml page, 190–191
long value, validating, 263
Luhn check, 298
Luhn formula, 271
@LuhnCheck annotation, 271, 272
lvalue mode, 63–64
Lynx browser, 525, 526

M

Mail
as error-prone, 534
sending, 532–537
Mail sample program, 534–537
malformedXML value, 400
Managed beans, 16, 39
communicating with stateless session beans, 513
configuring with XML, 58–63
implementing in Scala, 606
injecting, 57
injecting resources into, 495–496
setting as property of another, 57
testing, 604
using, 40
@ManagedBean annotation, 35
managed-bean elements
defining a managed bean, 59
syntax diagram for, 61–62
@ManagedProperty annotation, 57
Map(s)
f:selectItems, 158–159
implementing as a namespace, 399
initializing, 60–62

Map(s) *(cont.)*
issues when using, 158–159
Map keys, turning into item labels, 159
`Map` type, initializing values of, 60–62
Map values, turning into item values, 159
`map-entry` elements, 61
Markup, generating, 424–427
`maxFractionDigits` attribute, 251
`maxIntegerDigits` attribute, 251
`maxlength` attribute, 111, 126
Menu items, grouping, 159–161
`menuItems` property, 159
Menus
multiselect, 152–153
selection example using, 164–171
single-select, 152
tags for, 151–153
Message(s)
created by objects, 171–177
displayed as a list, 175
displaying, 174
posted by invalid component, 253
types of, 172
with variable parts, 42–43
`message` attribute, 271
Message bundle files, 42
Message bundles, 40–42, 257
Message strings, 40
`message-bundle` tag, 257
`MessageFormat` class, 42–43, 283
Messages sample application, 176–177
`messages.properties` file, 286
`META-INF` directory
in the JAR, 382
of a JAR file, 58
placing tag descriptor file into, 435
placing tag library file into, 197
`META-INF/context.xml` file, 498
Method expressions, 70–71
in an `action` attribute, 74
delaying execution of, 445
example of, 354
parameter values in, 71
for the submit button, 74
supporting, 443
taking parameters, 325
value of, 295
`MethodExpressionActionListener`, 445
Methods, calling, 66–67
`method-signature` attribute, 354, 355
`minFractionDigits` attribute, 251
`minIntegerDigits` attribute, 251
Model, 25
Model classes, 232–233
Model integrity, preserving, 248
Model-view-controller architecture, 25–26
Multi-component validation, 331–332
Multipart/form-data, 549, 554
Multiple select menus, 153
Multiselect listboxes, 151–152
Multiselect menus, 152–153

N

`name` attribute, 104
Name clashes, avoiding, 435
`Name` class, 208–209
`name` component, on the server, 390
`name` input, 394
`name` property
of a user object, 6
of `UserBean`, 16
Named anchors, links to, 140
`@Named` annotation, 39
`nameError` component, 395
Namespace declaration
for Facelets tags, 179
for sample application, 17
Namespaces
for composite components, 351
JavaScript, 398–399
Naming, a managed bean, 16
Naming convention
for composite components, 350–352
for multiple spinners in a page, 428
Naming pattern, for property getters and setters, 36
Navigation
conditional, 99
dynamic, 74–75
static, 73–74
Navigation handler, 73, 98
Navigation handling, actions participating in, 312
Navigation logic, 79
Navigation methods, 79

Navigation rules
advanced, 96–97
grouping multiple, 76
in JSF, 6
navigation-case element, 98
navigation-rule entries, 76
Netbeans
JSF support, 13
monitoring traffic between browser and server, 601, 602
supporting Groovy, 607
Newline characters (\n), 127
next method, 488
@none, 391
none scope, 59
noSelectionOption attribute
f:selectItem, 154
f:selectItems, 157
"not in range" messages, 265
null, initializing a property with, 60
null-value element, 60
Number quiz application
code for, 47, 48–51
directory structure of, 48
illustrating essential features of beans, 45–51
NumberFormatException catch clause, 430
Numbers
conversion of, 249–253
formatting as currency amounts, 43
Numeric ranges, validating, 262–264

O

Object(s)
compared to beans, 33
saving state of, 472
Object literal, in JavaScript, 399
object/relational (O/R) mapper. *See* O/R mapper
offset attribute, 217, 218
onchange attribute, 308
onerror attribute, 391, 392, 400
onerror key, 405
onevent attribute
f:ajax tag, 391, 396, 398
as JavaScript function, 392
onevent key, in Ajax request function, 405
OpenFaces open source library, 548
Operator precedence, 69
Operators, in value expressions, 110
optgroup elements, nested, 161
O/R mapper
in Java Persistence Architecture (JPA), 507
translating annotations, 507
translating between database tables and Java objects, 507
Original model, 236
Outcome, as value of the action attribute, 74
outcome attribute
h:button, 135
h:link, 135
target view ID specified by, 90
Outcome string, parameters in, 91
Outcomes, mapping to view IDs, 75–77
outputScript tag, 112
Overloaded methods, not supported, 66

P

Page authors, 348
Page-level validation, 271
Pager, navigating a large table, 568
Pager widgets, scrolling with, 243–244
Pages. *See also* JSF pages; Web pages
building from common templates, 183–186
cacheable, 89
redisplaying current, 253
showing or hiding parts of, 581–582
stuck, 602–603
Panel stack, 581–582
Panels, 115–118
param predefined object, 68
Parameters
adding support for debugging, 21
in the core library, 104
key/value pairs in Ajax options as, 406
method expressions taking, 325
providing to actions of buttons and links, 71
paramValues predefined object, 68
password property, 16
Passwords, checking, 361–362
@Past validation annotation, of birthday property, 378

pattern attribute
f:convertDateTime, 252
f:convertNumber, 251
persistence.xml file, 509
Phase events, 306, 328–329
Phase listeners
activating after Restore View phase, 566, 567–568
attaching to view root, 328–329
implementing, 329
Phase tracker, installing, 603
PhaseListener interface, 329
Phases
of JSF, 29
specifying all, 329
Placeholders
numbering, 42
substituting values for, 283
Planet application, files comprising template and views, 187
Planetarium, logging into, 181
Planetarium class, 196–197
Planets application, 181–193, 195–198
Pooling, database connections, 489
Pop-up data, sending to original page, 575
Pop-up window
creating as a blank window, 575
generating, 573–581
Pop-up window example application, 576–581
POST method, 119
POST requests, 28
browser sending back, 427
in a RESTful web application, 89
to the server, 88
PostAddToViewEvent, 330
@PostConstruct annotation, 58
PostConstructApplicationEvent, 330
PostConstructCustomScopeEvent, 330
PostConstructViewMapEvent, 330
PostRestoreStateEvent, 330
PostValidateEvent, 330, 331
listener to, 296
validating a group of components, 332
@PreDestroy annotation, 58
PreDestroyApplicationEvent, 330
PreDestroyCustomScopeEvent, 330
PreDestroyViewMapEvent, 330
Preemptive navigation, for target view IDs, 90
Preferred language, 44
Prepared statements, 491–492
prepareStatement method, 491–492
prependID attribute
for h:form tag, 119
setting, 22
PreRemoveFromViewEvent, 330
PreRenderComponentEvent, 330
PreRenderViewEvent, 330
Presentation layer
benefitting from operators, 69
separating from business logic, 16, 33
of a Web application, 15
pretty URLs, 89
PreValidateEvent, 330
PrimeFaces open source library, 548
ProblemBean class, 46
Process Validations phase, 30, 332
processAction method, 319, 364–365
Processing sequence. *See* Life cycle
Production stage, 583
Progress bar, on the client, 396
Project stage, setting to production, 583
PROJECT_STAGE parameter, 21
Properties
of beans, 36–37
defining, 16, 36, 46
naming, 37
specifying for tags, 441
.properties extension, 40
Properties file, for the login component, 359–360
Property setter
putting code into, 47
for tag attributes, 442
Property sheet dialog, property values from, 34
Property values
converting to strings, 62
setting, 59–60
PropertyResolver class, 598
Protected resource, requesting, 521
Prototype JavaScript library, 397
Push button, generating, 136, 137

Q

Queueing, events, 407–408
Quiz applications, 45–51, 92–96
`QuizBean` application, 45–51

R

Radio buttons, 148–150, 164–171
Range validator, 263
`readonly` attribute, 111
Read-only properties, 16
Read-write property, 148
Realms
  configuring in GlassFish, 522–524
  configuring in Tomcat, 524–525
  for web applications, 522
`redirect` element, 402
Redirection, to a new view, 86–87
`redisplay` attribute, 124, 125
Reference implementation, supporting hot deployment, 607
Registration link, adding to login component, 367
`rel` attribute, 111
Relational operators, 69
`render` attribute, 392
`render` components, 23
`render` key, 405
`render` list, 31
Render portion, of the JSF life cycle, 387
Render Response phase
  displaying messages in a view, 171
  of JSF life cycle, 29, 30, 31
`rendered` attribute
  as a basic attribute, 107
  including or excluding a component, 109–110
  rendering content associated with current tab, 340–341
  `ui:component`, 198
Renderer(s)
  by component family and renderer type, 438
  of font spinner, 475
  implementing separate, 421
  power of pluggable, 454
  producing HTML output, 27
`Renderer` class, 438–439
Renderer type, 438, 439
`Renderers` convenience class, 445
`renderFacet` composite tag, 349
Rendering
  delegating to a separate class, 438
  row numbers, 233–234
`renderResponse` method, 307, 322–323, 332
`request` function, 404–405, 409
Request parameter values, 153–154
Request parameters
  passing in Ajax, 405–407
  returning a map of, 433
  specifying, 91–92
Request scope, 51, 53–54, 226
Request scoped beans
  single-threaded, 54
  using with a redirect, 87–88
Request values, 247
  processing, 427–433
  as strings, 248
Requests
  decoding, 28–29
  monitoring, 396–398
`requestScope` predefined object, 68
`@RequestScoped` annotation, 51
`required` attribute(s)
  adding to the `Address` field, 321–322
  as a basic attribute, 108
  `h:inputHidden`, 124–125
  `h:inputSecret`, 124
  `h:inputText`, 124
  `h:inputTextarea`, 124
  on image's `composite:attribute` tag, 356
  in the input component, 264
  of text fields, 174
Required values, 264–265
`requiredMessage` attribute, 108, 265
Resolvers, 67, 596–599
Resource(s)
  displaying missing, 283
  localized versions of, 113
Resource bundles, 257
  associating with composite components, 359
  error messages from, 281–284
  getting, 282
  specifying, 459
Resource ID, of a detail message, 282
Resource injection, 495

resource predefined object, 68
resourceBundle attribute, 459
resource-bundle tag, 257
@ResourceDependency annotation, 454, 475
resources directory, 112, 132
response function, 400, 403, 404
Response writer, 426
responseCode attribute, 398
responseComplete method, 307, 566
Responses, Ajax, 400–403
responseText attribute, 398
ResponseWriter class, 425
responseXML attribute, 398
REST (Representational State Transfer) architectural style, 89
RESTful approach, to web services, 537–538
RESTful links, 92, 93
Restore View phase, 29
restoreState method, 468–469, 472
Result sets, 230
ResultSet class, 488
Reusability, of components, 353–354
rev attribute, 111
Revisited spinner example, 446
RichFaces open source component library, 548
right expression, evaluating in rvalue mode, 64
Roles, assigned during authentication, 520
rollback method, 493
Rolled back transaction, 493
Roundtrips, to the server, 453
Row indexes, sorting, 236
Row numbers, rendering, 233–234
rowClasses attribute
  h:dataTable, 210, 216
  h:panelGrid, 116
Row-oriented data, 209
Rows
  CSS classes applied to, 117
  deleting from tables, 225–228
  finding selected, 234
rows attribute
  h:dataTable, 210
  h:inputTextarea, 124, 126
  HTML pass-through, 111
rules attribute
  h:dataTable, 210
  h:panelGrid, 116, 117
Rushmore application, 313–318
rvalue mode, 63–64

S

Sample application, behind scenes of, 26–31
saveState method, 468–469, 472
Scala programming language, 605–607
Scalar object, 206
Schema declaration, version of, 41
Scope, of a managed bean, 16
Scoped variables, displaying, 199, 200
Scopes
  custom, 56
  defining for beans, 51
  of properties, 57
script tag, 201
Scripting languages, JS Faces with, 120–123
Scripts
  generating, 475–476
  simulating browser sessions, 604
Scrollable div, 242, 243
Scrollbar, in tables, 242–243
Scrolling techniques, 242–244
Seam framework, 517
Search service, invoking, 539
sections directory, 188
select elements, generating HTML, 151
Selection example, using checkboxes, radio buttons, menus, and listboxes, 164–171
Selection items, specifying, 153–155
Selection tags
  examples of, 145–146
  in JSF, 145–153
SelectItem instance, 158
SelectItem objects, 162, 460
SelectItemGroup instances, 159–161
selectMany tags, 146, 162
selectOne tags, 146
Selenium test automation framework, 604
Serializable interface, 35, 298
Serialization algorithm, 469

Server
  monitoring traffic with browser, 601–602
  passing data from the UI to, 324–328
  saving state on, 468
serverError value, 400
Server-side components, 120–123
Server-side data, 356–359
Server-side validation, 588
Service object, obtaining, 540
Servlet containers, 52
Servlet filter, installing, 549, 550, 552–554
Servlet runner, 10
Session(s)
  duration of, 35
  tracking, 52
Session bean methods, 513
Session beans, not single-threaded, 54
Session scope, 51, 52–53
sessionScope map, 69
sessionScope predefined object, 68
@SessionScoped annotation, 40, 51
set method, 36
setConverter method, 429, 433
setCurrentUser method, 57
setLocale method, 44
setProperty action, 63
setRowIndex method, 236
setSubmitedValue method, 433
Setter method, 47
setUser method, 60
Severity, setting, 281
shape attribute, 111
showDetail attribute, 173, 174
showProgress function, 397, 398
showSummary attribute, 173, 174
Simple table, 207–209
Single-select listbox, 151
Single-select menu, 152
Sitemesh framework, 193
size attribute, 110
  of h:selectOneListbox, 151
  HTML pass-through, 110, 111
  specifying visible characters in a text field, 126
  of ui:repeat, 217
SMTP connections, 534
smtpsend test program, 534
Sorting, table columns, 235
Sorting model, 236
source attribute, of a data object, 398
span element, generating, 131
spin function, 453
Spinner application, 435–437
  revisited, 445–452
Spinner component, 419–420
  descriptor file for, 434
  looking at closely, 422–423
Spinner renderer, versions of, 480, 481–484
SpinnerHandler class, 444–445
SQL injection attacks, 492
SQL statements, issuing, 487–488
SQLException, try/finally construct and, 490–491
src/java directory, 9
SSL (Secure Sockets Layer), 521
Stack trace, assembling a complete, 584
Standard conversion error messages, 257–258
Standard converters
  attributes, 251–252
  JSF implementation selecting, 251
  using, 249–262
Standard messages, replacing, 257
Standard validation error messages, 266
Standard validators
  provided with JSF, 263
  using, 262–269
startElement method, 425, 427
State
  algorithm for saving, 469
  saving and restoring, 468–473
  saving for converters or validators, 298
State helper, 470
State helper map, 473
Stateful session beans, 517
StateHelper class, 469
StateHolder interface, 468–469
@Stateless annotation, 514
Stateless session beans, 513, 517, 606
Statement object, 488
Static method, 599–600
Static navigation, 73–74
status attribute, 398

Status messages, 53
status property, 400
step attribute, 217
String conversion rules, 63
String length validator, 263
String lengths, validating, 262–264
Strings
  concatenating, 69–70
  converting, 62–63, 429
  request values as, 248
  validating, 263
Structured Query Language (SQL) commands, 487
Stuck page, debugging, 602–603
style attribute, 111, 255
styleClass attribute
  of h:messages, 255
  HTML pass-through, 111
  specifying a CSS style, 356
Styles
  specifying by column, 215–216
  specifying by row, 216–217
  specifying CSS classes, 215–218
Stylesheet
  including with an HTML link tag, 112
  inserting into head of a page, 184
Subexpressions, resolving, 597
Submit button, 363
submit function, 308
Submitted value, 248, 429, 433
subscribeToEvent method, 331
success attribute, 398
success request status, 397
Summary, in a message, 172
summary attribute
  h:dataTable, 210
  h:panelGrid, 116, 117
Summary error message, 255
@SuppressWarnings annotation, 508
System event demo application, 333–338
System events, 306, 329–331
  using, 331–333
System properties, 597

T

Tabbed pane
  application, 341–345
  children of, 458
  component, 420
  implementing, 338–341, 457–468
  plain-looking, 458
  poor man's implementation of, 338–345
  renderer, 460
  tabs encoded as hyperlinks, 462
  using facet names, 461
TabbedPaneRenderer class, 462–468
tabindex attribute, 111, 340
Tables
  caption, supplying, 213
  cells, editing, 222–225
  columns, sorting, 235
  editing, 222–228
  header, placing multiple components in, 214
  JSF components in, 218–221
  markup, rendering for ui:repeat, 217
  models, 232–242
  models, decorating, 234, 235–236
  rows, deleting, 225–228
  scrolling through, 242–244
  simple, 207–209
  sorting or filtering, 234–242
Tabs, specifying, 458
Tabular data, displaying, 205
Tag attributes, 107–115, 441–445
Tag descriptor files, 297–298
Tag handler class, 26
Tag handlers, 26, 441
Tag libraries, 17–18, 101
Tag library descriptor file, 195–196, 197, 433–435
Tag prefixes, in this book, 18
target attribute, 112
  HTML pass-through, 111
  of upload component, 554
Target view IDs, 90, 99
targets attribute, 361
TCP/IP sniffer, 602
Template
  building pages from, 183–186
  decorating contents, 194
  in Facelets, 181
  for planets application, 183–184
  putting ui:debug tag in, 199
  shared by planet application pages, 182

supplying arguments, 195
used by planets application, 185–186
Ternary ?: selection operator, 69
Testing tools, developing applications, 604–605
Text
displaying, 131–134
localizing in composite components, 359–360
placing inside a tag, 141
Text areas, 127–130
Text fields
adding a validator to, 262
attaching a converter to, 250
using, 127–130
Text fields and text areas example application, 128–130
Text inputs
validating, 389
for web applications, 123
`@this` keyword, 390, 391
Thread safety, 54
Tiles approach, 193
Time zones, predefined, 568, 569
`timeStyle` attribute, 252
`timeZone` attribute, 252
`title` attribute
associating a tooltip with each tab, 340
HTML pass-through, 111
`toInteger` method, 429
Tomahawk component set, 549
Tomcat
adding CDI reference implementation to, 39
configuring a database resource, 498–499
configuring a realm in, 524–525
cookies representing current user, 525
deployment directory, 12
log file, 12, 501
starting, 11
`tomcat` subdirectory, 10
`tooltip` attribute, 173–174
`toString` method, 47
`to-view-id` element, 99
Traffic, monitoring, 601–602
Transaction manager, obtaining, 509
Transactions, forming, 493
Transient property, 472
Transition diagram, 82
`TreeMap`, alphabetical order in, 158
`try/finally` block, 488
`type` attribute, 355
of a `data` object, 398
`f:convertDateTime`, 252
`f:convertNumber`, 251
`h:commandButton`, 135
`h:commandLink`, 135
`h:link`, 135
HTML pass-through, 111

U

UI. *See* User interfaces
`UI` prefix, 421
`UICommand` class, 421
`UICommand` component, 29
`UICommand` object, 29
`UICommand` superclass, 469
`UIComponent` class
backing component as a subclass of, 373
categories of data, 421–422
subclass of, 421
`ui:component` tag, 180, 198
`UIComponentBase` class, 480
`ui:composition` tag, 180
changing to `ui:component`, 198
compared to `ui:decorator`, 195
discarding surrounding XHTML tags, 188–189
with a `template` attribute, 184
`UIData` component, 206
`ui:debug` tag, 180, 198–200
`ui:decorate` tag, 180
generating a component, 198
inserting children of, 194
with a `template` attribute, 193
`ui:define` tags, 180, 195
inside the `ui:composition` tag, 185
overriding defaults, 194
`ui:fragment` tag, 180, 198
`ui:include` tag, 180, 184, 187
`UIInput` class, 39, 421
conversion code in, 440
extending, 374
`UIInput` components, 29

UIInput objects, 29
ui:insert tag, 180, 183, 184
UIOutput class, 39, 421
ui:param tag, 180, 195
ui:remove tag, 180, 201
ui:repeat tag, 180, 217–218
UISpinner class, 430–432, 480
UISpinner component, 423–433
UITabbedPane class
  content instance variable, 461
  property of, 469–470
  saving and restoring state, 470–472
Unicode characters, encoding, 42
update element, 402
Update Model Values phase, 30, 31, 248
Upload component, of the file upload application, 554–557
URLs
  browsers pointing to default, 12
  format of JSF, 20
  for POST requests, 28
  pretty, 89
  redirection updating in browsers, 87
  rewriting, 52–53
USA states, pop-up window listing, 574
User(s)
  gathering input from, 254
  navigating with GET requests, 90
user attribute, login component declaring, 358
user bean
  check methods of, 25
  located on the server, 23
User events, responding to, 305
User information, programmatic access to, 525
User input, 247, 248
User interface builder, 34
User interfaces
  generating markup for, 424
  implementing flexible, 179
  logic, 314
  passing data from to the server, 324–328
  passing information to the server, 325
UserBean class, 16, 35
UserBean objects, 35
UserBean.java file, 7–8
Username/password combination, 499–506
Utility tags, in Facelets, 202

V

validate method, 294
validateCreditCard function, 589–590
validateName method, 394, 395
Validating application, 299
Validation
  built-in, 263
  bypassing, 266–267
  client-side, 588
  of fields in Ajax, 394–396
  JavaScript function for, 588
  in JSF, 25
  of local values, 248
  server-side, 588
  skipping when a value change event fires, 323
  taking place on the server, 396
  triggering, 355, 356
Validation annotations, 270
Validation error messages, 266
Validation errors, displaying, 30, 265–266
Validation method, 294
Validation sample, 267–269
ValidationMessages.properties file, 271, 272
Validator(s)
  attaching to the login component, 360–365
  for local values, 30
  nesting inside input component tag, 264
  programming with custom, 275–297
  provided with JSF, 263
  registering custom, 290–291
  using standard, 262–269
validator attribute
  as a basic attribute, 107, 108
  of the component tag, 265
  of EditableValueHolder, 441
  input tags having, 109
  requiring a method expression, 70
Validator classes
  implementing custom, 290
  referencing annotation type, 272
Validator interface, 290

Validator messages, 265
Validator tags
  adding, 262
  attached to each component, 588
ValidatorException, 290
validatorMessage attribute, 108, 265
validatorScript component, 588–589, 592–594
validatorScript tag, 589
value attribute
  as a basic attribute, 107, 108
  binding, 161–162
  described, 109
  f:attribute, 104
  f:param, 104
  h:dataTable, 206, 211, 235
  h:outputText, 131
  tracking multiple selections, 162–164
  of upload component, 554
  of ValueHolder, 441
Value change events, 306, 307–312, 322
Value change listeners, 308, 442
Value change sample application, 308–312
Value changes, counting, 443
value element, containing a string, 62
Value expressions
  accessing bean properties, 37–38
  accessing message strings, 41
  with CDI beans, 40
  concatenating, 69–70
  evaluating a.b, 65
  example of, 354
  indexed properties and, 65
  invoking methods in, 66
  map managed by UIComponent, 422
  operators in, 69, 110
  in the outcome string, 91
  returning, 452, 453
  supplying, 90
  in user interface components, 64
  using, 109
valueChangeListener attribute, 443
  as a basic attribute, 107, 108
  of EditableValueHolder, 441
  h:inputHidden, 124–125
  h:inputSecret, 124
  h:inputText, 124
  h:inputTextarea, 124
  requiring a method expression, 70
ValueChangeListener interface, 319
valueHolder composite tag, 349, 364
ValueHolder interface, 421, 433, 441
Values
  checking for required, 264–265
  specifying with a string, 109
var attribute
  f:selectItems, 156, 157
  h:dataTable, 206, 211
  string expression accepted, 105
Variable parts, of messages, 42–43
Variable resolver, in JSF 1.1, 599
VariableResolver class, in JSF 1.1, 598
varStatus attribute, 218
Versioning mechanism, for resource libraries, 112–113
View component, wiring to a bean property, 25
View handler, replacing, 17
View IDs, mapping outcomes to, 74, 75–77
View parameters
  including in the query string, 91
  passing values, 89–90
  in a redirect link, 92
view predefined object, 68
View root, 328–329
View scope, 51, 55, 75
View state. *See* State
Views
  of the data model, 25
  implementing, 188
  organizing, 187–193
viewScope predefined object, 68
Visual builder tool, in an IDE, 14

W

.war extension, 8
WAR files
  copying to a deployment directory, 12
  deploying JSF applications, 8
  directory structure of, 9
warnClass attribute, 173
warnStyle attribute, 173
Weather information, searching for, 541
Weather service, accessing, 538

Web applications
displaying views of a model, 25
executing SQL statements from, 488
JPA in, 508–513
parts of, 15
placing most beans into session scope, 53
sending mail in, 532–537
web directory, 9
Web forms
backing beans for, 38–39
user filling in fields of, 247
Web pages
beans storing the state of, 35
design contained in, 16
making changes to, 21
values as always strings, 162
Web project, importing from existing sources, 13
Web service sample application, 541–544
Web services, 537–544
Web Services Definition Language (WSDL), 538
Web user interface, 249
@WebServiceRef annotation, 540
web.xml file, 7, 19
configuration parameters in, 595
specifying a welcome page, 20
supplying, 21
Weekday objects, 158
Welcome message, using Ajax, 22
Welcome page
of authentication test application, 526
specifying, 20
Welcome screen
for a database application, 499, 500
page defining, 7
Whitespace, in Facelets pages, 202
width attribute, 111
Wildcards, in a navigation rule, 97–98
writeAttribute method, 425, 427
Write-only properties, 16
writeValidationFunctions method, 589, 590
WS-* approach, to web services, 537
WS-* web service
calling, 540
JSF built-in support for, 538
WSDL (Web Services Definition Language), 538
wsimport tool, 540

X

.xhtml extension, 17
XHTML file
implementing custom tag in, 195–196
namespace declaration to, 348
XHTML markup, 191–193
XHTML page, 17
XML
configuring managed beans with, 58–63
configuring property values with, 59–60
XML comments, value expressions in, 201
XML configuration, specifying navigation rules in, 92
XML elements, for Ajax responses, 402