The JavaServer™ Faces Technology Tutorial

June 15, 2003

Please send feedback to jsfguidefeedback@sun.com

Contents

Preface

THE JavaServer™ Faces Technology Tutorial is a beginner's guide to creating Web applications using JavaServer Faces technology. JavaServer Faces technology is a framework for building Java Web applications with server-side user interface functionality. JavaServer Faces technology simplifies Java Web application development by handling all of the complexities associated with managing a user interface.

This section covers all the things you need to know to make the best use of this tutorial.

Who Should Use This Tutorial

This tutorial is intended for page authors, application developers, and component writers interested in developing and deploying JavaServer applications with server-side UI functionality.

In addition to explaining how to use JavaServer Faces technology to build simple applications, this guide first goes over some of the benefits of using JavaServer Faces technology and how JavaServer Faces applications work. The first chapter, Introduction to JavaServer™ Faces Technology (page 1), will help you understand the general JavaServer Faces concepts and architecture. The second chapter, Using JavaServer Faces Technology (page 31), uses a simple, working application to explain the main features of JavaServer Faces technology. The third chapter, Creating Custom UI Components (page 117), explains how to create custom components using JavaServer Faces technology.

How to Print This Tutorial

To print this tutorial, follow these steps:

- Ensure that Adobe Acrobat Reader is installed on your system.
- Open the PDF version of this book.
- Click the printer icon in Adobe Acrobat Reader.

About the Examples

This release includes five complete, working examples, which are located in the `example` directory of your installation. Table 1–1 lists the examples and where they are located.

This tutorial uses the `cardemo` and `guessNumber` to explain JavaServer Faces technology. It also uses some extra code snippets not contained in `cardemo` or `guessNumber` to explain features not demonstrated by these applications.

Table 1–1 Examples

Example	Location	Function
`cardemo`	`<JWSDP_HOME>/jsf/samples/cardemo`	A car store application
`guessNumber`	`<JWSDP_HOME>/jsf/samples/guessNumber`	Duke asks you to guess a number
`non-jsp`	`<JWSDP_HOME>/jsf/samples/non-jsp`	Demonstrates non-JSP rendering
`components`	`<JWSDP_HOME>/jsf/samples/components`	Showcases tabbed-panes, tree-control, and result-set custom components

Prerequisites for the Examples

In addition to having good knowledge of the Java programming language, the audience of this tutorial should have some knowledge of JavaServer Pages (JSP) technology, including custom tag libraries, and the JavaServer Pages Standard Tag Library (JSTL).

Required Software

This tutorial assumes you are using the Java WSDP as your deployment environment. To build, deploy, and run the examples you need a copy of the Java WSDP and the Java™ 2 Platform, Standard Edition (J2SE™) SDK 1.3.1 or 1.4. You download the Java WSDP from:

```
http://java.sun.com/webservices/downloads/webservicespack.html
```

the J2SE 1.3.1 SDK from

```
http://java.sun.com/j2se/1.3/
```

or the J2SE 1.4 SDK from

```
http://java.sun.com/j2se/1.4/
```

Add the `bin` directories of the Java WSDP and J2SE SDK installations to the front of your `PATH` environment variable so that the Java WSDP startup scripts for Tomcat override other installations.

Set the environment variable JWSDP_HOME to the location of your Java WSDP installation.

Download the JavaServer Faces technology implementation from:

```
http://java.sun.com/j2ee/javaserverfaces/download.html
```

Running the Examples Using the Pre-Installed XML Files

The Java Web Services Developer Pack ("Java WSDP"), v. 1.2 includes an XML file for each example application in the `<JWSDP_HOME>/webapps` directory. This file causes an application to be automatically deployed when you start Tomcat. To run an example that is already deployed:

1. Set the environment variables:
   a. Set `JAVA_HOME` to your J2SE installation directory
   b. Set `JWSDP_HOME` to your Java WSDP 1.2 installation directory
   c. Set `ANT_HOME` to `$JWSDP_HOME/apache-ant` (Solaris) or `%JWSDP_HOME%\apache-ant` (Windows).

   d. Set `JSF_HOME` to `$JWSDP_HOME/jsf` (Solaris) or `%JWSDP_HOME%\jsf` (Windows)
2. On a system running the Solaris or Linux operating system, go to the `<JWSDP_HOME>/bin` directory and execute the `catalina.sh` script to bring up the Java WSDP. On a system running Microsoft Windows, from the Start menu, select Programs, Java(tm) Web Services Developer Pack 1.2, and Start Tomcat.
3. Once the server is up and running, point your browser to `http://localhost:8080`, the default port at which the process is running. The page that is displayed contains links to several sample programs and administration tools.
4. Click on one of the links to run the corresponding example.

Building and Running the Sample Apps Manually

It is also possible to build each of the sample apps manually. Before doing so, you need to set the environment variables, as described in Running the Examples Using the Pre-Installed XML Files (page ix) and edit your `build.properties` file.

To edit the build.properties file:

1. Go to the `<JWSDP_HOME>/jsf/samples` directory.
2. Copy `build.properties.sample` to `build.properties`. This file provides build properties for all of the samples.
3. In build.properties, set `tomcat.home` to `JWSDP_HOME`.
4. Set the username and password to the username and password you configured for the user who has the manager role in the Java WSDP.

To build a sample:

1. Shutdown Tomcat if it's running by executing either `catalina.sh stop` if you are running the UNIX operating system or `catalina stop`, if you are running Windows.
2. Move the pre-installed XML files out of the `<JWSDP_HOME>/webapps` directory.
3. Go to the directory of the example you want to build.

4. At the command line, run Ant with no target:
   `ant`
5. This will cause the sample to be built, and the WAR file for the sample to be put into the `<JWSDP_HOME>/jsf/samples` directory. The existing pre-installed XML files will cause tomcat to find your newly compiled sample.

Basic Requirements of a JavaServer Faces Application

JavaServer Faces applications are Java server applications and must be compliant with the Java Servlet specification, version 2.3 (or later) and the JavaServer Pages specification, version 1.2 (or later). All Java server applications are packaged in a WAR file. The WAR file must conform to specific requirements in order to execute across different JavaServer Faces implementations. At a minimum, a WAR file for a JavaServer Faces application must contain:

- A Web application deployment descriptor, called `web.xml`, to configure resources required by a Web application.
- A specific set of JAR files containing essential classes.
- A set of application classes, JavaServer Faces pages, and other required resources, such as image files.
- An application configuration file, which defines application resources

The `web.xml`, the set of JAR files, and the set of application files must be contained in the `WEB-INF` directory of the WAR file. Usually, you will want to use the `Ant` build tool to compile the classes, build the necessary files into the WAR, and deploy the WAR file. The `Ant` tool is included in the Java WSDP. You configure how the `Ant` build tool builds your WAR file with a `build.xml` file. Each example in the download has its own build file. Look at one of those build files for an example of writing a build file.

Another requirement is that all requests to a JavaServer Faces application that reference previously saved JavaServer Faces components must go through the `FacesServlet`. The `FacesServlet` manages the request processing lifecycle for Web applications and initializes the resources required by the JavaServer Faces implementation. To make sure your JavaServer Faces application complies with this requirement, see the section, Invoking the FacesServlet (page xv).

Writing the web.xml File

The `web.xml` file is located at the top level of the `WEB-INF` directory. See *Configuring Web Applications* in *The Java Web Services Tutorial* to see what a standard `web.xml` file should contain.

The `web.xml` file for a JavaServer Faces application must specify certain configurations, which include:

- The servlet used to process JavaServer Faces requests
- The servlet mapping for the processing servlet

The following XML markup defines the required configurations specific to JavaServer Faces technology for the `cardemo` application:

```
<web-app>
...
  <!-- Faces Servlet -->
  <servlet>
     <servlet-name>Faces Servlet</servlet-name>
     <servlet-class>
        javax.faces.webapp.FacesServlet
     </servlet-class>
     <load-on-startup> 1 </load-on-startup>
  </servlet>

  <!-- Faces Servlet Mapping -->
  <servlet-mapping>
     <servlet-name>Faces Servlet</servlet-name>
     <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Identifying the Servlet for Lifecycle Processing

The `servlet` element identifies the `FacesServlet`, which processes the lifecycle of the application. The `load-on-startup` element has a value of `true`, which indicates that the `FacesServlet` should be loaded when the application starts up.

Provide the Path to the Servlets

The `servlet-mapping` element lists each servlet name defined in the servlet element and gives the URL path to the servlet. Tomcat will map the path to the servlet when a request for the servlet is received.

JSP pages do not need an alias path defined for them because Web containers automatically map an alias path that ends in `*.jsp`.

Including the Required JAR Files

JavaServer Faces applications require several JAR files to run properly. If you are not running the application on the Java WSDP, which already has these JAR files, the WAR file for your JavaServer Faces application must include the following set of JAR files in the `WEB-INF/lib` directory:

- `jsf-api.jar` (contains the `javax.faces.*` API classes)
- `jsf-ri.jar` (contains the implementation classes of the JavaServer Faces RI)
- `jstl.jar` (required to use JSTL tags and referenced by JavaServer Faces reference implementation classes)
- `jstl_el.jar` (required for handling JSTL expression language syntax)
- `standard.jar` (required to use JSTL tags and referenced by JavaServer Faces reference implementation classes)
- `commons-beanutils.jar` (utilities for defining and accessing JavaBeans component properties)
- `commons-digester.jar` (for processing XML documents)
- `commons-collections.jar` (extensions of the Java 2 SDK Collections Framework)
- `commons-logging.jar` (a general purpose, flexible logging facility to allow developers to instrument their code with logging statements)

To run your application standalone, you need to:

Comment out the build.wspack property and uncomment the build.standalone property in your build.properties file.

Comment out the jsp.jar, servlet.jar, jsf-api.jar, and jsf-ri.jar properties from the build.properties file.

Including the Classes, Pages, and Other Resources

All application classes and properties files should be copied into the `WEB-INF/classes` directory of the WAR file during the build process. JavaServer

Faces pages should be at the top level of the WAR file. The web.xml, faces-config.xml, and extra TLD files should be in the `WEB-INF` directory. Other resources, such as images can be at the top level or in a separate directory of the WAR file.

The `build` target of the example build file copies all of these files to a temporary `build` directory. This directory contains an exact image of the binary distribution for your JavaServer Faces application:

```
<target name="build" depends="prepare"
  description="Compile Java files and copy static files." >
  <javac srcdir="src"
       destdir="${build}/${example}/WEB-INF/classes">
     <include name="**/*.java" />
     <classpath refid="classpath"/>
  </javac>
  <copy todir="${build}/${example}/WEB-INF">
     <fileset dir="web/WEB-INF"    >
        <include name="web.xml" />
        <include name="*.tld" />
        <include name="*.xml" />
     </fileset>
  </copy>
  <copy todir="${build}">
     <fileset dir="web">
        <include name="*.html" />
        <include name="*.gif" />
        <include name="*.jpg" />
        <include name="*.jsp" />
        <include name="*.xml" />
        <include name="*.css" />
     </fileset>
  </copy>
  <copy
     todir="${build}/${example}/WEB-INF/classes/${example}" >
     <fileset dir="src/${example}" >
        <include name="*properties"/>
     </fileset>
     <fileset dir="src/${example}" >
     <include name="*.xml"/>
     </fileset>
  </copy>
</target>
```

The `build.war` target packages all the files from the `build` directory into the WAR file while preserving the directory structure contained in the `build` directory:

```
<target name="build.war" depends="build"
  <jar jarfile="${example}.war"
    basedir="${build}/${example}" />
  <copy todir=".." file="{example}.war" />
  <delete file="${example}.war" />
</target>
```

When writing a build file for your Web application, you can follow the build files included with each example.

Invoking the FacesServlet

Before a JavaServer Faces application can launch the first JSP page, the Web container must invoke the `FacesServlet` in order for the application lifecycle process to start. The application lifecycle is described in the section, The Lifecycle of a JavaServer Faces Page (page 13).

To make sure that the `FacesServlet` is invoked, you need to include the path to the `FacesServlet` in the URL to the first JSP page. You define the path in the `url-pattern` element nested inside the `servlet-mapping` element of the `web.xml` file. In the example `web.xml` file above, the path to the `FacesServlet` is `/faces`.

To include the path to the `FacesServlet` in the URL to the first JSP page, you must do one of two things:

- Include an HTML page in your application that has the URL to the first JSP page, and include the path to the `FacesServlet`:

  ```
  <a href="faces/First.jsp">
  ```

- Include the path to the `FacesServlet` in the URL to the first page when you enter it in your browser:

  ```
  http://localhost:8080/myApp/faces/First.jsp
  ```

The second method allows you to start your application from the first JSP page, rather than starting it from an HTML page. However, the second method requires your user to identify the first JSP page. When you use the first method, the user only has to enter:

```
http://localhost:8080/myApp
```

Setting Up The Application Configuration File

The Application Configuration File is new with this release. It is an XML file, named faces-config.xml, whose purpose is to configure resources for an application. These resources include: navigation rules, converters, validators, render kits, and others. For a complete description of the application configuration file, see Application Configuration (page 29). This section explains the basic requirements of for using file.

The Application Configuration file must be valid against the DTD located at http://java.sun.com/dtd/web-facesconfig_1_0.dtd. In addition, each file must include in this order:

- The XML version number:
  ```
  <?xml version="1.0"?>
  ```
- This DOCTYPE declaration at the top of the file:
  ```
  <!DOCTYPE faces-config PUBLIC
  “-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
  1.0//EN”
  “http://java.sun.com/dtd/web-facesconfig_1_0.dtd”>
  ```
- A faces-config tag enclosing all of the other declarations:
  ```
  <faces-config>
  ...
  </faces-config>
  ```

You can have more than one application configuration file, and there are three ways that you can make these files available to the application. The JavaServer Faces implementation finds the file or files by looking for:

- A resource named `/META-INF/faces-config.xml` in any of the JAR files in the Web application’s `/WEB-INF/lib` directory. If a resource with this name exists, it is loaded as a configuration resource. This method is practical for a packaged library containing some components and renderers.

The demo-components.jar, located in <JWSDP_HOME>jsf/samples uses this method.

- A context init parameter, `javax.faces.application.CONFIG_FILES` that specifies one or more (comma-delimited) paths to multiple configuration files for your Web application. This method will most likely be used for enterprise-scale applications that delegate the responsibility for maintaining the file for each portion of a big application to separate groups.
- A resource named faces-config.xml in the /WEB-INF/ directory of your application if you don't specify a context init parameter. This is the way most simple apps will make their configuration files available.

Related Information

For further information on the technologies discussed in this tutorial see the Web sites listed in Table 1–2. References to individual technology homes listed in some chapters map as follows:

Table 1–2 Related Information

Technology	**Web Site**
JavaServer Faces technology	`http://java.sun.com/j2ee/javaserverfaces/`
Java Servlets	`http://java.sun.com/products/servlet/`
JavaServer Pages technology	`http://java.sun.com/products/jsp/`
JSP Standard Tag Library	`http://java.sun.com/products/jsp/taglibraries.html#jstl`
Tomcat	`http://jakarta.apache.org/tomcat/`
`Ant`	`http://ant.apache.org`

Introduction to JavaServer™ Faces Technology

JAVASERVER Faces technology is a user interface framework for building Web applications that run on a Java server and render the UI back to the client.

The main components of JavaServer Faces technology are:

- An APIs and reference implementation for: representing UI components and managing their state; handling events, server side validation, and data conversion; defining page navigation; supporting internationalization and accessibility; and providing extensibility for all of these features.
- A JavaServer Pages™ (JSP™) custom tag library for expressing UI components within a JSP page.

This well-defined programming model and UI component tag library significantly ease the burden of building and maintaining Web applications with server-side UIs. With minimal effort, you can:

- Wire client-generated events to server-side application code
- Map UI components on a page to server-side data
- Construct a UI with reusable and extensible components.
- Save and restore UI state beyond the life of server requests

As shown in Figure 2–1, the user interface you create with JavaServer Faces technology (represented by `myUI` in the graphic) runs on the server and renders back to the client.

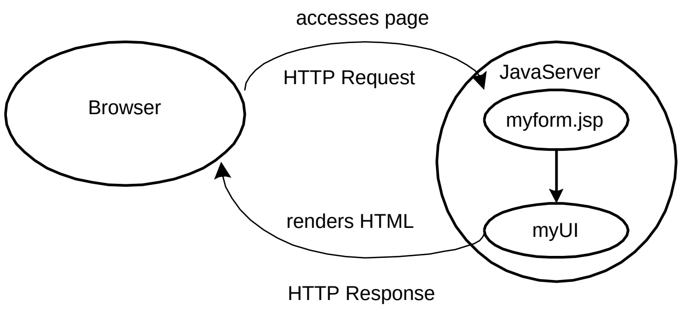


Figure 2–1 The UI Runs on the Server

The JSP page, `myform.jsp`, expresses the user interface components with custom tags defined by JavaServer Faces technology framework rather than hard-coding them with a markup language. The UI for the Web application (represented by `myUI` in the figure) manages the objects referenced by the JSP page. These objects include:

- The component objects that map to the tags on the JSP page
- The event listeners, validators, and converters that are registered on the components
- The model objects that encapsulate the data and application-specific functionality of the components

JavaServer Faces Technology Benefits

One of the greatest advantages of JavaServer Faces technology is that it offers a clean separation between behavior and presentation. Web applications built with JSP technology partially achieve this separation. However, a JSP application cannot map HTTP requests to component-specific event handling or manage UI elements as stateful objects on the server. JavaServer Faces technology allows you to build Web applications that implement finer-grained separation of behavior and presentation traditionally offered by client-side UI architectures.

The separation of logic from presentation also allows each member of a Web application development team to focus on their piece of the development process, and provides a simple programming model to link the pieces together. For example, Page Authors with no programming expertise can use JavaServer Faces technology UI component tags to link to application code from within a Web page without writing any scripts.

Another important goal of JavaServer Faces technology is to leverage familiar UI-component and Web-tier concepts without limiting you to a particular scripting technology or markup language. While JavaServer Faces technology includes a JSP custom tag library for representing components on a JSP page, the JavaServer Faces technology APIs are layered directly on top of the JavaServlet API. This allows you to do a few things: to use another presentation technology besides JSP, to create your own custom components directly from the component classes, and to generate output for different client devices.

Most importantly, JavaServer Faces technology provides a rich architecture for managing component state, processing component data, validating user input, and handling events.

What is a JavaServer Faces Application?

For the most part, JavaServer Faces applications are just like any other Java Web application. They run in a Java Servlet container, and they typically contain:

- JavaBeans^{TM} components (called model objects in JavaServer Faces technology) containing application-specific functionality and data
- Event listeners
- Pages, such as JSP pages
- Server-side helper classes, such as database-access beans

In addition to these items, a JavaServer Faces application also has:

- A custom tag library for rendering UI components on a page
- A custom tag library for representing event handlers, validators, and other actions.
- UI components represented as stateful objects on the server
- Validators, event handlers, and navigation handlers

Every JavaServer Faces application must include a custom tag library that defines the tags representing UI components and a custom tag library for representing other core actions, such as validators and event handlers. Both of these tag libraries are provided by the JavaServer Faces implementation.

The component tag library eliminates the need to hard-code UI components in HTML or another markup language, resulting in completely reusable components. And, the core tag library makes it easy to register events, validators, and other actions on the components.

The component tag library can be the `html_basic` tag library included with the JavaServer Faces technology reference implementation, or you can define your own tag library that renders custom components or renders output other than HTML.

Another important advantage of JavaServer Faces applications is that the UI components on the page are represented as stateful objects on the server. This allows the application to manipulate the component state and wire client-generated events to server-side code.

Finally, JavaServer Faces technology allows you to convert and validate data on individual components and report any errors before the server-side data is updated.

This tutorial provides more detail on each of these features. First, let's look at a JSP page and a JavaServer Faces page side-by side.

An Example JavaServer Faces Page

To see how much easier Web development is with JavaServer Faces technology, it helps to look at the differences between a JavaServer Faces page and a JSP page. The following JSP page comes from the *Web Applications* chapter of *The Java Web Services Tutorial*. This page asks you to type your name into a text field and click the button. It then displays your name on the page.

```
<html>
<head><title>Hello</title></head>
<body bgcolor="white">
<img src="duke.waving.gif">
<h2>My name is Duke. What is yours?</h2>

<form method="get">
<input type="text" name="username" size="25">
<p></p>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>
<%
String username = request.getParameter("username");
```

```
if ( username != null && username.length() > 0 ) {
%>
<%@include file="response.jsp" %>
<%
}
%>
</body>
</html>
```

Even for this very simple page, you need to know how to extract the user name from the request parameters, which requires some programming knowledge. An average page author might not know how to do this.

Now, let's look at the JavaServer Faces version of this page. Note that instead of including the response in the same page, the JavaServer Faces version displays the response on a second page. Here is the first page:

```
<HTML>
  <HEAD> <title>Hello</title> </HEAD>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <body bgcolor="white">
  <h2>My name is Duke.  What is yours?</h2>
  <jsp:useBean id="UserNameBean"
     class="helloDuke.UserNameBean" scope="session" />
  <f:use_faces>
     <h:form id="helloForm" formName="helloForm" >
        <h:graphic_image id="wave_img" url="/wave.med.gif" />
        <h:input_text id="username"
           valueRef="UserNameBean.userName"/>
        <h:command_button id="submit" label="Submit"
           commandName="submit" />
     </h:form>
  </f:use_faces>
</HTML>
```

Here is the second page:

```
<HTML>
  <HEAD> <title>Hello</title> </HEAD>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <body bgcolor="white">
  <h:graphic_image id="wave_img" url="/wave.med.gif" />
  <f:use_faces>
     <h:form id="responseform" formName="responseform">
        <h:graphic_image id="wave_img" url="/wave.med.gif" />
```

```
            <h2>Hi, <h:output_text id="userLabel"
              valueRef="UserNameBean.userName" /> </h2>
            <h:command_button id="back" label="Back"
              commandName="back" /><p>
          </h:form>
          </f:use_faces>
    </HTML>
```

The first difference to note is that these pages contain no Java code. Any logic that needs to be performed is done in model objects or helper classes, not in the pages.

The logic can be referenced from the component tags in the pages. The `h:input_text` tag represents the text field that takes the user's name. As the `valueRef` attribute of the `h:input_text` tag specifies, the user's name is saved to the `userName` property of the model object, `UserNameBean`. The `h:output_text` tag retrieves the user's name from `UserNameBean` and displays it on the following page.

While it's true that you can eliminate the script by using the JSTL tags, `c:set` and `c:out`, these tags cannot associate the data with a stateful UI component, like the `input_text` and `output_text` tags do. This will become even more important to you as you build more complicated applications.

By moving the code out of the pages and into model objects on the server, a Web development team will have a much easier time maintaining and scaling the application. With JavaServer Faces technology, the page author can easily write the entire page and simply reference the logic—written by the developer—from the component tags. The next section describes all of the roles of the Web development team and which part of a JavaServer Faces application they are responsible for.

Framework Roles

Because of the division of labor enabled by the JavaServer Faces technology design, JavaServer Faces application development and maintenance can proceed quickly and easily. The members of a typical development team are those listed below. In many teams, individual developers play more than one of these roles, however, it is still useful to consider JavaServer Faces technology from a variety of perspectives based on primary responsibility.

- **Page Authors**, who use a markup language, like HTML, to author pages for Web applications. When using the JavaServer Faces technology framework, page authors will most likely use the tag library exclusively.
- **Application Developers**, who program the model objects, the event handlers, the validators, and the page navigation. Application developers can also provide the extra helper classes.
- **Component Writers**, who have user-interface programming experience and prefer to create custom components using a programming language. These people can create their own components directly from the component classes, or they can extend the standard components provided by JavaServer Faces technology.
- **Tools Vendors**, who provide tools that leverage JavaServer Faces technology to make building server-side user interfaces even easier.

The primary users of JavaServer Faces technology will be page authors and application developers. This tutorial is written with these two customers in mind. The next section walks through a simple application, explaining which piece of the application the page author and the application developer develops.

The third chapter, Creating Custom UI Components (page 117) covers the responsibilities of a component writer.

A Simple JavaServer Faces Application

This section describes the process of developing a simple JavaServer Faces application. You'll see what features a typical JavaServer Faces application contains, and what part each role has in developing the application.

Steps in the Development Process

Developing a simple JavaServer Faces application requires performing these tasks:

- Develop the model objects, which will hold the data
- Add managed bean declarations to the Application Configuration File
- Create the Pages using the UI component and core tags

• Define Page Navigation

These tasks can be done simultaneously or in any order. However, the people performing the tasks will need to communicate during the development process. For example, the page author needs to know the names of the model objects in order to access them from the page.

The example used in this section is slightly more complicated than the example in An Example JavaServer Faces Page (page 4). This example asks you to guess a number between 0 and 10, inclusive. The second page tells you if you guessed correctly. The example also checks the validity of your input.

To deploy and execute this example, follow the instructions in Running the Examples Using the Pre-Installed XML Files (page ix).

Develop the Model Objects

Developing model objects is the responsibility of the application developer. The page author and the application developer might need to work in tandem to make sure that the component tags refer to the proper object properties, that the object properties have the proper types, and take care of other such details.

Here is the `UserNumberBean` class that holds the data entered in the text field on `greeting.jsp`:

```
package guessNumber;
import java.util.Random;

public class UserNumberBean {

Integer userNumber = null;
Integer randomInt = null;
String response = null;

public UserNumberBean () {
  Random randomGR = new Random();
  randomInt = new Integer(randomGR.nextInt(10));
  System.out.println("Duke’s Number: "+randomInt);
}

public void setUserNumber(Integer user_number) {
  userNumber = user_number;
  System.out.println("Set userNumber " + userNumber);
}
```

```
    public Integer getUserNumber() {
      System.out.println("get userNumber " + userNumber);
      return userNumber;
    }

    public String getResponse() {
      if(userNumber.compareTo(randomInt) == 0)
        return "Yay! You got it!";
      else
        return "Sorry, "+userNumber+" is incorrect.";
    }
```

As you can see, this bean is just like any other JavaBeans component: It has a set of accessor methods and a private data field for each property. This means that you can conceivably reference beans you've already written from your JavaServer Faces pages.

Depending on what kind of component references a particular model object property, the model object property can be any of the basic primitive and reference types. This includes any of the `Number` types, `String`, `int`, `double`, and `float`. JavaServer Faces technology will automatically convert the data to the type specified by the model object property. See Using the HTML Tags (page 53) and Writing a Model Object Class (page 75) for information on which types are accepted by which component tags.

You can also apply a converter to a component to convert the components value to a type not supported by the component. See Performing Data Conversions (page 92) for more information on applying a converter to a component.

In the `UserNumberBean`, the `userNumber` property has a type of `Integer`. The JavaServer Faces implementation can convert the `String` request parameters containing this value into an `Integer` before updating the model object property when you use an `input_number` tag. Although this example converts to an `Integer` type, in general, you should use the native types rather than the wrapper classes.

Adding Managed Bean Declarations

After developing the beans to be used in the application, you need to add declarations for them in the Application Configuration file. The task of adding managed bean declarations to the Application Configuration File can be done by any member of the development team. Here is a managed bean declaration for `UserNumberBean`:

```
<managed-bean>
  <managed-bean-name>UserNumberBean</managed-bean-name>
  <managed-bean-class>
    guessNumber.UserNumberBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The JavaServer Faces implementation processes this file on application startup time and initializes the UserNumberBean and stores it in session scope. The bean is then available for all pages in the application. For those familiar with previous releases, this managed bean facility replaces usage of the jsp:useBean tag. For more information, see the sections Managed Bean Creation (page 28) and Application Configuration (page 29).

Creating the Pages

Authoring the pages is the page author's responsibility. This task involves laying out UI components on the pages, mapping the components to model object data, and adding other core tags (such as validator tags) to the component tags.

Here is the new `greeting.jsp` page with the `validator` tags (minus the surrounding HTML):

```
<HTML>
  <HEAD> <title>Hello</title> </HEAD>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <body bgcolor="white">
  <h:graphic_image id="wave_img" url="/wave.med.gif" />
  <h2>Hi. My name is Duke.
    I'm thinking of a number from 0 to 10.
    Can you guess it?</h2>
  <f:use_faces>
    <h:form id="helloForm" formName="helloForm" >
      <h:graphic_image id="wave_img" url="/wave.med.gif" />
      <h:input_number id="userNo" numberStyle="NUMBER"
        valueRef="UserNumberBean.userNumber">
        <f:validate_longrange minimum="0" maximum="10" />
      </h:input_number>
      <h:command_button id="submit" action=”success”
```

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

This page demonstrates a few important features that you will use in most of your JavaServer Faces applications:

- The `form` Tag
  The `form` tag represents an input form, which allows the user to input some data and submit it to the server, usually by clicking a button. The tags representing the components that comprise the form are nested in the `form` tag. These tags are `h:input_number` and `h:command_button`.
- The `input_number` Tag
  The `input_number` tag represents a text field component, into which the user enters a number. This tag has two attributes: `id` and `valueRef`. The optional `id` attribute corresponds to the ID of the component object represented by this tag. The id attribute is optional. If you don't include one, the JavaServer Faces implementation will generate one for you. See Creating Model Objects (page 33) for more information.
  The `valueRef` uses a reference expression to refer to the model object property that holds the data entered into the text field. The part of the expression before the "." must match the name defined by the managed-bean-name element corresponding to the proper managed-bean declaration from the Application Configuration file. The part of the expression after the "." must match the name defined by the property-name element corresponding to the proper managed-bean declaration.
- The `validate_longrange` Tag
  The `input_number` tag also contains a `validate_longrange` tag, which is one of a set of standard validator tags included with JavaServer Faces technology. This validator checks if the local value of a component is within a certain range. The value must be anything that can be converted to a long. The `validate_longrange` tag has two attributes, one that specifies a minimum value and the other that specifies a maximum value. Here, the tag is used to ensure that the number entered in the text field is a number from 0 to 10. See Performing Validation (page 81) for more information on performing validation.
- The `command_button` Tag
  The `command_button` tag represents the button used to submit the data entered in the text field. The action attribute specifies an output that helps

the navigation mechanism to decide which page to open next. The next section discusses this further.

- The `output_errors` Tag
  The `output_errors` tag will display an error message if the data entered in the field does not comply with the rules specified by the validator. The error message displays wherever you place the `output_errors` tag on the page. The `for` attribute refers to the component whose value failed validation.

Creating Model Objects (page 33) discusses the tags in more detail and includes a table that lists all of the basic tags included with JavaServer Faces technology.

The next section discusses the navigation instructions used with this example.

Define Page Navigation

Another responsibility that the application developer has is to define page navigation for the application, such as which page to go to after the user clicks a button to submit a form. The JavaServer Faces navigation model, new for this release, is explained in Navigation Model (page 27). Navigating Between Pages (page 105) explains how to define the navigation rules for an entire application.

The application developer defines the navigation for the application in the application configuration file, the same file in which managed beans are declared.

Here are the navigation rules defined for the `guessNumber` example:

```
<navigation-rule>
  <from-tree-id>/greeting.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/response.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-tree-id>/response.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/greeting.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
```

Each `navigation-rule` defines how to get from one page (specified in the `from-tree-id` element) to the other pages of the application. The `navigation-rule` elements can contain any number of `navigation-case` elements, each of which defines the page to open next (defined by `to-tree-id`) based on a logical outcome (defined by `from-outcome`).

The outcome can be defined by the `action` attribute of the `UICommand` component that submits the form, as it is in the `guessNumber` example:

```
<h:command_button id="submit"
  action="success" label="Submit" />
```

The outcome can also come from the return value of the invoke method of an Action object. The invoke method performs some processing to determine the outcome. One example is that the invoke method can check if the password the user entered on the page matches the one on file. If it does, the invoke method could return "success"; otherwise, it might return "failure". An outcome of "failure" might result in the logon page being reloaded. An outcome of "success" might result in the page displaying the user's credit card activity opening.

To learn more about how navigation works and how to define navigation rules, see the sections Navigation Model (page 27) and Navigating Between Pages (page 105).

The Lifecycle of a JavaServer Faces Page

The lifecycle of a JavaServer Faces page is similar to that of a JSP page: The client makes an HTTP request for the page, and the server responds with the page translated to HTML. However, because of the extra features that JavaServer Faces technology offers, the lifecycle provides some additional services by executing some extra steps.

Which steps in the lifecycle are executed depends on whether or not the request originated from a JavaServer Faces application and whether or not the response is generated with the rendering phase of the JavaServer Faces lifecycle. This section first explains the different lifecycle scenarios. It then explains each of these lifecycle phases using the `guessNumber` example.

Request Processing Lifecycle Scenarios

A JavaServer Faces application supports two different kinds of responses and two different kinds of requests:

- Faces Response: A servlet response that was created by the execution of the Render Response (page 18) phase of the request processing lifecycle.
- Non-Faces Response: A servlet response that was not created by the execution of the Render Response phase. An example is a JSP page that does not incorporate JavaServer Faces components.
- Faces Request: A servlet request that was sent from a previously generated Faces Response. An example is a form submit from a JavaServer Faces user interface component, where the request URI identifies the JavaServer Faces component tree to use for processing the request.
- Non-Faces Request: A servlet request that was sent to an application component, such as a servlet or JSP page, rather than directed to a JavaServer Faces component tree.

These different requests and responses result in three possible lifecycle scenarios that can exist for a JavaServer Faces application:

- **Scenario 1: Non-Faces Request Generates Faces Response**
  An example of this scenario is when clicking a hyperlink on an HTML page opens a page containing JavaServer Faces components. To render a Faces Response from a Non-Faces Request, an application must provide a mapping to the `FacesServlet` in the URL to the page containing JavaServer Faces components. The `FacesServlet` accepts incoming requests and passes them to the lifecycle implementation for processing.
- **Scenario 2: Faces Request Generates Non-Faces Response**
  Sometimes a JavaServer Faces application might need to redirect to a different Web application resource or generate a response that does not contain any JavaServer Faces components. In these situations, the developer must skip to the rendering phase (Render Response (page 18)) by calling `FacesContext.responseComplete`. The `FacesContext` contains all of the information associated with a particular Faces Request. This method can be invoked during the Apply Request Values (page 16), Process Validations (page 17), or Update Model Values (page 17) phases.
- **Scenario 3: Faces Request Generates Faces Response**
  This is the most common scenario for the lifecycle of a JavaServer Faces application. It is also the scenario represented by the standard request processing lifecycle described in the next section. This scenario involves Jav-

aServer Faces components submitting a request to a JavaServer Faces application utilizing the `FacesServlet`. Because the request has been handled by the JavaServer Faces implementation, no additional steps are required by the application to generate the response. All listeners, validators and validators will automatically be invoked during the appropriate phase of the standard lifecycle, which the next section describes.

Standard Request Processing Lifecycle

The standard request processing lifecycle represents scenario 3, described in the previous section. Most users of JavaServer Faces technology won't need to concern themselves with the request processing lifecycle. However, knowing that JavaServer Faces technology properly performs the processing of a page, a developer of JavaServer Faces applications doesn't need to worry about rendering problems associated with other UI framework technologies. One example involves state changes on individual components. If the selection of a component such as a checkbox effects the appearance of another component on the page, JavaServer Faces technology will handle this event properly and will not allow the page to be rendered without reflecting this change.

Figure 2–2 illustrates the steps in the JavaServer Faces request-response lifecycle.

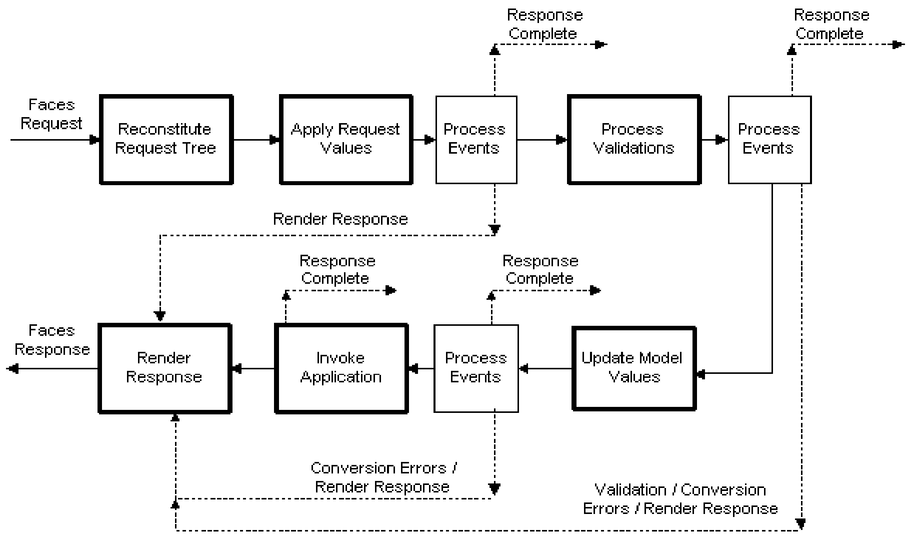


Figure 2–2 JavaServer Faces Request-Response Lifecycle

Reconstitute Component Tree

When a request for a JavaServer Faces page is made, such as when clicking on a link or a button, the JavaServer Faces implementation begins the Reconstitute Component Tree stage.

During this phase, the JavaServer Faces implementation builds the component tree of the JavaServer Faces page, wires up event handlers and validators, and saves the tree in the FacesContext. The component tree for the `greeting.jsp` page of the `guessNumber` example might conceptually look like this:

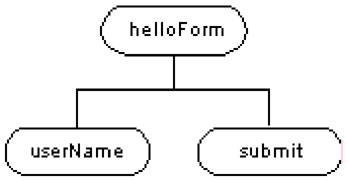


Figure 2–3 `guessNumber` Component Tree

Apply Request Values

Once the component tree is built, each component in the tree extracts its new value from the request parameters with its `decode` method. The value is then stored locally on the component. If the conversion of the value fails, an error message associated with the component is generated and queued on the `Faces-Context`. This message will be displayed during the Render Response phase, along with any validation errors resulting from the Process Validations phase.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners. See Implementing an Event Listener (page 100) for more information on how to specify which lifecycle processing phase the listener will process events.

In the case of the `userNumber` component on the `greeting.jsp` page, the value is whatever the user entered in the field. Since the model object property bound to the component has an `Integer` type, the JavaServer Faces implementation converts the value from a `String` to an `Integer`.

At this point, the components are set to their new values, and messages and events have been queued.

Process Validations

During this phase, the JavaServer Faces implementation processes all validations registered on the components in the tree. It examines the component attributes that specify the rules for the validation and compares these rules to the local value stored for the component. If the local value is invalid, the JavaServer Faces implementation adds an error message to the `FacesContext` and the lifecycle advances directly to the Render Response phase so that the page is rendered again with the error messages displayed. If there were conversion errors from Apply Request Values, the messages for these errors are displayed also.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners. See Implementing an Event Listener (page 100) for more information on how to specify in which lifecycle processing phase a listener will process events.

In the `greeting.jsp` page, the JavaServer Faces implementation processes the validator on the `userNumber` `input_number` tag. It verifies that the data the user entered in the text field is an integer from the range 0 to 10. If the data is invalid, or conversion errors occurred during the Apply Request Values phase, processing jumps to the Render Response phase, during which the `greeting.jsp` page is rendered again with the validation and conversion error messages displayed in the component associated with the `output_errors` tag.

Update Model Values

Once the JavaServer Faces implementation determines that the data is valid, it can walk the component tree and set the corresponding model object values to the components' local values. Only input components that have `valueRef` expressions will be updated. If the local data cannot be converted to the types specified by the model object properties, the lifecycle advances directly to Render Response so that the page is re-rendered with errors displayed, similar to what happens with validation errors.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners. See Implementing an Event Listener (page 100) for more information on how to specify in which lifecycle processing phase a listener will process events.

At this stage, the `userNumber` property of the `UserNumberBean` is set to the local value of the `userNumber` component.

Invoke Application

During this phase, the JavaServer Faces implementation handles any application-level events, such as submitting a form or linking to another page.

The `greeting.jsp` page from the `guessNumber` example has one application-level event associated with the `Command` component. When processing this event, a default `ActionListener` implementation retrieves the outcome, "success", from the component's `action` attribute. The listener passes the outcome to the default `NavigationHandler`. The `NavigationHandler` matches the outcome to the proper navigation rule defined in the application's application configuration file to determine what page needs to be displayed next. See Navigating Between Pages (page 105) for more information on managing page navigation. The JavaServer Faces implementation then sets the response component tree to that of the new page. Finally, the JavaServer Faces implementation transfers control to the Render Response phase.

Render Response

During the Render Response phase, the JavaServer Faces implementation invokes the components' encoding functionality and renders the components from the component tree saved in the `FacesContext`.

If errors were encountered during the Apply Request Values phase, Process Validations phase, or Update Model Values phase, the original page is rendered during this phase. If the pages contain `output_errors` tags, any queued error messages are displayed on the page.

New components can be added to the tree if the application includes custom renderers, which define how to render a component. After the content of the tree is rendered, the tree is saved so that subsequent requests can access it and it is available to the Reconstitute Component Tree phase. The Reconstitute Component Tree phase accesses the tree during a subsequent request.

User Interface Component Model

JavaServer Faces UI components are configurable, reusable elements that compose the user interfaces of JavaServer Faces applications. A component can be simple, like a button, or compound, like a table, which can be composed of multiple components.

JavaServer Faces technology provides a rich, flexible component architecture that includes:

- A set of `UIComponent` classes for specifying the state and behavior of UI components
- A rendering model that defines how to render the components in different ways.
- An event and listener model that defines how to handle component events
- A conversion model that defines how to plug in data converters onto a component
- A validation model that defines how to register validators onto a component

This section briefly describes each of these pieces of the component architecture.

The User-Interface Component Classes

JavaServer Faces technology provides a set of UI component classes, which specify all of the UI component functionality, such as holding component state, maintaining a reference to model objects, and driving event-handling and rendering for a set of standard components.

These classes are completely extensible, which means that component writers can extend the classes to create their own custom components. See Creating Custom UI Components (page 117) for an example of a custom image map component.

All JavaServer Faces UI component classes extend from `UIComponentBase`, which defines the default state and behavior of a `UIComponent`. The set of UI component classes included in this release of JavaServer Faces are:

- `UICommand`: Represents a control that fires actions when activated.
- `UIForm`: Encapsulates a group of controls that submit data to the application. This component is analogous to the form tag in HTML.
- `UIGraphic`: Displays an image.
- `UIInput`: Takes data input from a user. This class is a subclass of `UIOutput`.
- `UIOutput`: Displays data output on a page.
- `UIPanel`: Displays a table.
- `UIParameter`: Represents substitution parameters.
- `UISelectItem`: Represents a single item in a set of items.
- `UISelectItems`: Represents an entire set of items.
- `UISelectBoolean`: Allows a user to set a `boolean` value on a control by selecting or de-selecting it. This class is a subclass of `UIInput`.
- `UISelectMany`: Allows a user to select multiple items from a group of items. This class is a subclass of `UIInput`.
- `UISelectOne`: Allows a user to select one item out of a group of items.This class is a subclass of `UIInput`.

Most page authors and application developers will not have to use these classes directly. They will instead include the components on a page by using the component's corresponding tag. Most of these component tags can be rendered in different ways. For example, a `UICommand` can be rendered as a button or a hyperlink.

The next section explains how the rendering model works and how page authors choose how to render the components by selecting the appropriate tag.

The Component Rendering Model

The JavaServer Faces component architecture is designed such that the functionality of the components is defined by the component classes, whereas the component rendering can be defined by a separate renderer. This design has several benefits including:

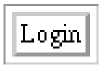
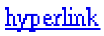
- Component writers can define the behavior of a component once, but create multiple renderers, each of which defines a different way to render the component to the same client or to different clients.
- Page authors and application developers can change the appearance of a component on the page by selecting the tag that represents the appropriate component/renderer combination.

A render kit defines how component classes map to component tags appropriate for a particular client. The JavaServer Faces implementation includes a standard `RenderKit` for rendering to an HTML client.

For every UI component that a `RenderKit` supports, the `RenderKit` defines a set of `Renderer` objects. Each `Renderer` defines a different way to render the particular component to the output defined by the `RenderKit`. For example, a `UISelectOne` component has three different renderers. One of them renders the component as a set of radio buttons. Another renders the component as a combo box. The third one renders the component as a list box.

Each JSP custom tag in the standard HTML RenderKit is composed of the component functionality, defined in the `UIComponent` class, and the rendering attributes, defined by the `Renderer`. For example, the two tags in Table 2–1 both represent a `UICommand` component, rendered in two different ways:

Table 2–1 `UICommand` tags

Tag	Rendered as
command_button	
command_hyperlink	

The command part of the tags corresponds to the `UICommand` class, specifying the functionality, which is to fire an action. The button and hyperlink parts of the tags each correspond to a separate `Renderer`, which defines how the component is rendered.

The JavaServer Faces reference implementation provides a custom tag library for rendering components in HTML. It supports all of the component tags listed in Table 2–2. To learn how to use the tags in an example, see Creating Model Objects (page 33).

Table 2–2 The Component Tags

Tag	Functions	Rendered as	Appearance
`command_button`	Submits a form to the application.	An HTML `<input type=`*type*`>` element, where the *type* value can be `submit`, `reset`, or `image`	A button
`command_hyperlink`	Links to another page or location on a page.	An HTML `<a href>` element	A Hyperlink
`form`	Represents an input form. The inner tags of the form receive the data that will be submitted with the form.	An HTML `<form>` element	No appearance
`graphic_image`	Displays an image.	An HTML `<img>` element	An image
`input_date`	Allows a user to enter a date.	An HTML `<input type= text>` element	A text string, formatted with a `java.text.DateFormat` date instance
`input_datetime`	Allows a user to enter a date and time.	An HTML `<input type=text>` element	A text string, formatted with a `java.text.SimpleDateFormat` datetime instance
`input_hidden`	Allows a page author to include a hidden variable in a page.	An HTML `<input type=hidden>` element	No appearance

Table 2–2 The Component Tags (Continued)

Tag	Functions	Rendered as	Appearance
`input_number`	Allows a user to enter a number.	An HTML `<input type=text>` element	A text string, formatted with a `java.text.NumberFormat` instance
`input_secret`	Allows a user to input a string without the actual string appearing in the field.	An HTML `<input type=password>` element	A text field, which displays a row of characters instead of the actual string entered
`input_text`	Allows a user to input a string.	An HTML `<input type=text>` element	A text field
`input_textarea`	Allows a user to enter a multi-line string.	An HTML `<textarea>` element	A multi-row text field
`input_time`	Allows a user to enter a time.	An HTML `<input type=text>` element	A text string, formatted with a `java.text.DateFormat` time instance
`output_date`	Displays a formatted date.	plain text	A text string, formatted with a `java.text.DateFormat` time instance
`output_datetime`	Displays a formatted date and time.	plain text	A text string, formatted with a `java.text.SimpleDateFormat` datetime instance
`output_errors`	Displays error messages.	plain text	plain text
`output_label`	Displays a nested component as a label for a specified input field.	An HTML `<label>` element	plain text
`output_message`	Displays a localized message.	plain text	plain text

Table 2–2 The Component Tags (Continued)

Tag	Functions	Rendered as	Appearance
`output_number`	Displays a formatted number.	plain text	A text string, formatted with a `java.text.NumberFormat` instance
`output_text`	Displays a line of text.	plain text	plain text
`output_time`	Displays a formatted time.	plain text	A text string, formatted with a `java.text.DateFormat` time instance
`panel_data`	Iterates over a collection of data.		A set of rows in a table
`panel_grid`	Displays a table.	An HTML `<table>` element with `<tr>` and `<td>` elements	A table
`panel_group`	Groups a set of components under one parent.		A row in a table
`panel_list`	Displays a table of data that comes from a collection, array, iterator, or map.	An HTML `<table>` element with `<tr>` and `<td>` elements	A table
`selectboolean_checkbox`	Allows a user to change the value of a boolean choice.	An HTML `<input type=checkbox>` element.	A checkbox
`selectitem`	Represents one item in a list of items in a `UISelectOne` component.	An HTML `<option>` element	No appearance
`selectitems`	Represents a list of items in a `UISelectOne` component.	A list of HTML `<option>` elements	No appearance

Table 2–2 The Component Tags (Continued)

Tag	Functions	Rendered as	Appearance
`selectmany _checkboxlist`	Displays a set of checkboxes, from which the user can select multiple values.	A set of HTML `<input>` elements of type checkbox	A set of checkboxes
`selectmany _listbox`	Allows a user to select multiple items from a set of items, all displayed at once.	A set of HTML `<select>` elements	A list box
`selectmany_menu`	Allows a user to select multiple items from a set of items.	A set of HTML `<select>` elements	A scrollable combo box
`selectone _listbox`	Allows a user to select one item from a set of items, all displayed at once.	A set of HTML `<select>` elements	A list box
`selectone_menu`	Allows a user to select one item from a set of items.	An HTML `<select>` element	A scrollable combo box
`selectone_radio`	Allows a user to select one item from a set of items.	An HTML `<input type=radio>` element	A set of radio buttons

Conversion Model

A JavaServer Faces application can optionally associate a component with server-side model object data. This model object is a JavaBeans component that encapsulates the data on a set of components. An application gets and sets the model object data for a component by calling the appropriate model object properties for that component.

When a component is bound to a model object, the application has two views of the component's data: the model view and the presentation view, which represents the data in a manner that can be viewed and modified by the user.

A JavaServer Faces application must ensure that the component's data can be converted between the model view and the presentation view. This conversion is usually performed automatically by the component's renderer.

In some situations, you might want to convert a component's data to a type not supported by the component's renderer. To facilitate this, JavaServer Faces technology includes a set of standard `Converter` implementations and also allows you to create your own custom `Converter` implementations. If you register the `Converter` implementation on a component, the `Converter` implementation converts the component's data between the two views. See Performing Data Conversions (page 92) for more details on the converter model, how to use the standard converters, and how to create and use your own custom converter.

Event and Listener Model

One goal of the JavaServer Faces specification is to leverage existing models and paradigms so that developers can quickly become familiar with using JavaServer Faces in their web applications. In this spirit, the JavaServer Faces event and listener model leverages the JavaBeans event model design, which is familiar to GUI developers and Web Application Developers.

Like the JavaBeans component architecture, JavaServer Faces technology defines `Listener` and `Event` classes that an application can use to handle events generated by UI components. An `Event` object identifies the component that generated the event and stores information about the event. To be notified of an event, an application must provide an implementation of the `Listener` class and register it on the component that generates the event. When the user activates a component, such as clicking a button, an event is fired. This causes the JavaServer Faces implementation to invoke the listener method that processes the event.

JavaServer Faces supports two kinds of events: value-changed events and action events.

A *value-changed* event occurs when the user changes a component value. An example is selecting a checkbox, which results in the component's value changing to true. The component types that generate these types of events are the `UIInput`, `UISelectOne`, `UISelectMany`, and `UISelectBoolean` components. Value-changed events are only fired if no validation errors were detected.

An *action event* occurs when the user clicks a button or a hyperlink. The `UICommand` component generates this event.

For more information on handling these different kinds of events, see Handling Events (page 99).

Validation Model

JavaServer Faces technology supports a mechanism for validating a component's local data during the Process Validations (page 17) phase, before model object data is updated.

Like the conversion model, the validation model defines a set of standard classes for performing common data validation checks. The `jsf-core` tag library also defines a set of tags that correspond to the standard `Validator` implementations.

Most of the tags have a set of attributes for configuring the validator's properties, such as the minimum and maximum allowable values for the component's data. The page author registers the validator on a component by nesting the validator's tag within the component's tag.

Also like the conversion model, the validation model allows you to create your own `Validator` implementation and corresponding tag to perform custom validation. See Performing Validation (page 81) for more information on the standard `Validator` implementations and how to create custom `Validator` implementation and validator tags.

Navigation Model

Virtually all web applications are made up of a set of pages. One of the primary concerns of a web application developer is managing the navigation between these pages.

The new JavaServer Faces navigation model makes it easy to define page navigation and to handle any additional processing needed to choose the sequence in which pages are loaded. In many cases, no code is required to define navigation. Instead, navigation can be completely defined in the application configuration resource file (see section Application Configuration (page 29)) using a small set of XML elements. The only situation in which you need to provide some code is if additional processing is required to determine which page to access next.

To load the next page in a web application, the user usually clicks a button. As explained in the section Define Page Navigation (page 12), a button click generates an action event. The JavaServer Faces implementation provides a new, default action event listener to handle this event. This listener determines the outcome of the action, such as success or failure. This outcome can be defined as a string property of the component that generated the event or as the result of extra processing performed in an `Action` object associated with the component. After the outcome is determined, the listener passes it to the `NavigationHandler` instance associated with the application. Based on which outcome is returned, the `NavigationHandler` selects the appropriate page by consulting the application configuration file.

For more information on how to perform page navigation, see section Navigating Between Pages (page 105).

Managed Bean Creation

Another critical function of web applications is proper management of resources. This includes separating the definition of UI component objects from data objects and storing and managing these object instances in the proper scope. Previous releases of JavaServer Faces technology enabled you to create model objects that encapsulated data and business logic separately from UI component objects and store them in a particular scope. This release fully specifies how these objects are created and managed.

This release introduces new APIs for:

- Evaluating an expression that refers to a model object, a model object property, or other primitive or data structure. This is done with the `ValueBinding` API.
- Retrieving the object from scope. This is done with the `VariableResolver` API.
- Creating an object and storing it in scope if it is not already there. This is done with the default `VariableResolver`, called the Managed Bean Facility, which is configured with the application configuration file, described in the next section.

Application Configuration

Previous sections of this chapter have discussed the various resources available to a JavaServer Faces application. These include: converters, validators, components, model objects, actions, navigation handlers, and others. In previous releases, these resources had to be configured programmatically. An `ApplicationHandler` was required to define page navigation, and a `ServletContextListener` was required to register converters, validators, renderers, render kits, and messages.

This release introduces a portable configuration resource format (as an XML document) for configuring resources required at application startup time. This new feature eliminates the need for an `ApplicatonHandler` and a `ServletContextListener`. This tutorial explains in separate sections how to configure resources in the XML document. See section Setting Up The Application Configuration File (page xvi) for information on requirements for setting up the application configuration file. See section Creating Model Objects (page 33) for an explanation of how to use the faces-config.xml file to create model objects. See section Navigating Between Pages (page 105) for information on how to define page navigation in the faces-config.xml file. See sections Performing Validation (page 81) and Performing Data Conversions (page 92) for how to register custom validators and converters. See sections Register the Component (page 140) and Register the Renderer with a Render Kit (page 139) for information on how to register components and renderers to an application.

Once these resources were created, the information for some of these resources used to be stored in and accessed from the `FacesContext`, which represents contextual information for a given request. These resources are typically available during the life of the application. Therefore, information for these resources is more appropriately retrieved from a single object that is instantiated for each application. This release of JavaServer Faces introduces the `Application` class, which is automatically created for each application.

The `Application` class acts as a centralized factory for resources such as converters and message resources that are defined in the faces-config.xml file. When an application needs to access some information about one of the resources defined in the faces-config.xml file, it first retrieves an `Application` instance from an `ApplicationFactory` and retrieves the resource instance from the `Application`.

Using JavaServer Faces Technology

This section shows you how to use JavaServer Faces technology in a Web application by demonstrating simple JavaServer Faces features using a working example. This example emulates on online car dealership, with features such as price updating, car option packaging, a custom converter, a custom validator, and an image map custom component.

The cardemo Example

Table 3–1 lists all of the files used in this example, except for the image and properties files.

Table 3–1 Example Files

File	Function
`ImageMap.jsp`	The first page that allows you to select a locale
`Storefront.jsp`	Shows the cars available
`more.jsp`	Allows you to choose the options for a particular car
`buy.jsp`	Shows the options currently chosen for a particular car
`Customer.jsp`	Allows you to enter your personal information so that you can order the car
`Thanks.jsp`	The final page that thanks you for ordering the car
`error.jsp`	A page that displays an error message

Table 3–1 Example Files (Continued)

File	Function
`CarActionListener.java`	The `ActionListener` that handles the car packaging dependencies on `more.jsp`
`CreditCardConverter.java`	Defines a custom `Converter`
`FormatValidator.java`	Defines a custom `Validator`
`CurrentOptionServerBean.java`	Represents the model for the currently-chosen car
`CustomerBean.java`	Represents the model for the customer information
`ImageMapEventHandler.java`	Handles the `ActionEvent` caused by clicking on the image map
`PackageValueChanged.java`	Handles the event of selecting options on `more.jsp` and updates the price of the car

The `cardemo` also uses a set of model objects, custom components, renderers, and tags, as shown in Table 3–2. These files are located in the `examples/components` directory of your download.

Table 3–2 Model Objects and Custom Components, Renderers, and Tags Used by `cardemo`

File	Function
`AreaRenderer`	This `Renderer` performs the delegated rendering for the `UIArea` component
`AreaTag`	The tag handler that implements the `area` custom tag
`ImageArea`	The model object that stores the shape and coordinates of the hot spots
`MapTag`	The tag handler that implements the `map` custom tag
`UIArea`	The class that defines the `UIArea` component, corresponding to the `area` custom tag

Table 3–2 Model Objects and Custom Components, Renderers, and Tags Used by `cardemo (Continued)`

File	Function
`UIMap`	The class that defines the `UIMap` component, corresponding to the `map` custom tag

Figure 3–1 illustrates the page flow for the `cardemo` application

.

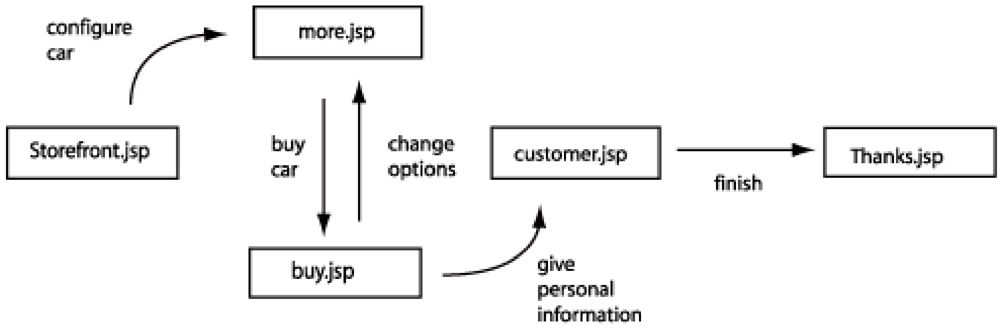


Figure 3–1 Page Flow for `cardemo`

How to Build and Run the Example

If you just want to run the example, simply follow the instructions in Running the Examples Using the Pre-Installed XML Files (page ix).

The `example/cardemo` directory also contains a `build.xml` that you can use to build and run the example in case you would like to make changes to any of the source files. Follow the directions in Building and Running the Sample Apps Manually (page x) to build and run the example.

Creating Model Objects

Previous releases of JavaServer Faces technology required the page author to create a model object by declaring it from the page using the `jsp:useBean` tag. This technique had its disadvantages, one of which was that if a user accessed the pages of an application out of order, the bean might not have been created before a particular page was referring to it.

The new way to create model objects and store them in scope is with the Managed Bean Creation facility. This facility is configured in the application configuration resource file (see section Application Configuration (page 29)using managed-bean XML elements to define each bean. This file is processed at application startup time, which means that the objects declared in it are available to the entire application before any of the pages are accessed.

The Managed Bean Creation facility has many advantages over the jsp:useBean tag, including:

- You can create model objects in one centralized file that is available to the entire application, rather than conditionally instantiating model objects throughout the application.
- You can make changes to the model object without any additional code
- When a managed bean is created, you can customize the bean's property values directly from within the configuration file.
- Using `value-ref` elements, you can set the property of one managed bean to be the result of evaluating another value reference expression.
- Managed beans can be created programmatically as well as from a JSP page. You'd do this by creating a `ValueBinding` for the value reference expression and then calling `getValue` on it.

This section will show you how to initialize model objects using the Managed Bean Creation Facility. The section Writing a Model Object Class (page 75) explains how to write a model object class.

Using the `managed-bean` Element

You create a model object using a `managed-bean` element. The `managed-bean` element represents an instance of a bean class that must exist in the application. At runtime, the JavaServer Faces implementation processes the `managed-bean` element and instantiates the bean as specified by the element configuration.

Most of the model objects used with cardemo are still created with `jsp:useBean`. The `Storefront.jsp` page uses the `useBean` tag to declare the `CurrentOptionServer` model object:

```
<jsp:useBean id="CurrentOptionServer"
    class="cardemo.CurrentOptionServer" scope="session"
  <jsp:setProperty name="CurrentOptionServer"
    property="carImage" value="current.gif"/>
  </jsp:useBean>
```

To instantiate this bean using the Managed Bean Creation facility, you would add this `managed-bean` element configuration to the application configuration file:

```
<managed-bean>
  <managed-bean-name> CurrentOptionServer </managed-bean-name>
  <managed-bean-class>
    cardemo.CurrentOptionServer
  </managed-bean-class>
  <managed-bean-scope> session </managed-bean-scope>
  <managed-property>
    <property-name>carImage</property-name>
    <value>current.gif</value>
  </managed-property>
</managed-bean>
```

The `managed-bean-name` element defines the key under which the bean will be stored in a scope. For a component to map to this bean, the component tag's `valueRef` must match the `managed-bean-name` up to the first period. For example, this `valueRef` refers maps to the `carImage` property:

```
valueRef="CurrentOptionServer.carImage"
```

The part before the "." matches the `managed-bean-name` of `CurrentOptionServer`. The section Using the HTML Tags (page 53) has more examples of using `valueRef` to bind components to bean properties.

The `managed-bean-class` element defines the fully-qualified name of the JavaBean-compliant class used to instantiate the bean. It is the application developer's responsibility to ensure that the class complies with the configuration of the bean in the application configuration resources file. For example, the property definitions must match those configured for the bean.

The `managed-bean-scope` element defines the scope in which the bean will be stored. The four acceptable scopes are: none, request, session or application. If you define the bean with a none scope, the bean is instantiated anew each time it is referenced, and so it does not get saved in any scope. One reason to use a scope of none is when a managed bean references another `managed-bean`. The second bean should be in none scope if it is only supposed to be created when it is referenced. See section Initializing Managed Bean Properties (page 40) for an example of initializing a managed-bean property.

The `managed-bean` element can contain zero or more `managed-property` elements, each corresponding to a property defined in the bean class. These elements are used to initialize the values of the bean properties. In the example above, the `carImage` property is initialized with the value `current.gif`. The next section explains in more detail how to use the `managed-property` element.

Initializing Properties using the managed-property Element

A `managed-property` element must contain a `property-name` element, which must match the name of the corresponding property in the bean. A `managed-property` element must also contain one of a set of elements (listed in Table 3–3 on page 36) that defines the value of the property. This value must be of the same type as that defined for the property in the corresponding bean. Which element you use to define the value depends on the type of the property defined in the bean. Table 3–3 on page 36 lists all of the elements used to initialize a value.

Table 3–3 subelements of managed-property that define property values

element	value that it defines
map-entries	defines the values of a map
null-value	explicitly sets the property to null.
value	defines a single value, such as a String or int
values	defines an aggregate value, such as an array or List
value-ref	references another object

The section Using the managed-bean Element (page 34) includes an example of initializing String properties using the `value` subelement. You also use the `value` `subelement to` initialize primitive and other reference types. The rest of this section describes how to use the `value` subelement and other subelements to initialize properties of type `java.util.Map`, `array` and `Collection`.

Referencing an Initialization Parameter

Another powerful feature of the Managed Bean Facility is the ability to reference implicit objects from a managed bean property.

Suppose that you have a page that accepts data from a customer, including the customer's address. Suppose also that most of your customers live in a particular zip code. You can make the zip code component render with this zip code by saving it in an implicit object and referencing it when the page is rendered.

You can save the zip code as an initial default value in the context initParam implicit object by setting the `context-param` element in your web.xml file:

```
<context-param>
  <param-name>defaultZipCode</param-name>
  <param-value>94018</param-name>
</context-param>
```

Next, you write a managed-bean declaration with a property that references the parameter:

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
     <managed-bean-class>CustomerBean</managed-bean-class>
     <managed-bean-scope>request</managed-bean-scope>
     <managed-property>
        <property-name>zipCode</property-name>
          <value-ref>initParam.defaultZipCode</value-ref>
        </managed-property>
        ...
</managed-bean>
```

To access the zip code at the time the page is rendered, refer to the property from the `zip` component tag's `valueRef` attribute:

```
<h:input_text id=zip valueRef="customer.zipCode"
```

Retrieving values from other implicit objects are done in a similar way. See Table 3–6 on page 44 for a list of implicit objects.

Initializing Map Properties

The map-entries element is used to initialize the values of a bean property with a type of java.util.Map. Here is the definition of map-entries from the DTD that defines the application configuration file:

```
<!ELEMENT map-entries (key-class?, value-class?, map-entry*) >
```

As this definition shows, a map-entries element contains an optional key-class element, an optional value-class element and zero or more map-entry elements.

Here is the definition of map-entry from the DTD:

```
<!ELEMENT map-entry (key, (null-value|value|value-ref)) >
```

According to this definition, each of the map-entry elements must contain a key element and either a null-value, value, or value-ref element. Here is an example that uses the map-entries element:

```
<managed-bean>
  ...
  <managed-property>
     <property-name>cars</property-name>
     <map-entries>
        <map-entry>
           <key>Jalopy</key>
           <value>50000.00</value>
        </map-entry>
        <map-entry>
           <key>Roadster</key>
           <value-ref>
              sportsCars.roadster
           </value-ref>
        </map-entry>
     </map-entries>
  </managed-property>
</managed-bean>
```

The map that is created from this map-entries tag contains two entries. By default, the keys and values are all converted to java.lang.String. If you want to specify a different type for the keys in the map, embed the `key-class` element just inside the `map-entries` element:

```
<map-entries>
  <key-class>java.math.BigDecimal</key-class>
  ...
</map-entries>
```

This declaration will convert all of the keys into `java.math.BigDecimal`. Of course, you need to make sure that the keys can be converted to the type that you specify. The key from the example in this section cannot be converted to a `java.math.BigDecimal` because it is a String.

If you also want to specify a different type for all of the values in the map, include the `value-class` element after the `key-class` element:

```
<map-entries>
  <key-class>int</key-class>
  <value-class>java.math.BigDecimal</value-class>
  ...
</map-entries>
```

Note that this tag only sets the type of all the `value` subelements.

The first `map-entry` in the example above includes a `value` subelement. The `value` subelement defines a single value, which will be converted to the type specified in the bean according to the rules defined in the JavaServer Pages Specification, 2.0.

The second `map-entry` defines a `value-ref` element, which references a property on another bean. Referencing another bean from within a bean property is useful for building a system out of fine-grained objects. For example, a request-scoped form-handling object might have a pointer to an application-scoped database mapping object, and together the two can perform a form handling task. Note that including a reference to another bean will initialize the bean if it does not exist already.

It is also possible to assign the entire map with a value-ref element that specifies a map-typed expression, instead of using a `map-entries` element.

Initializing Array and Collection Properties

The values element is used to initialize the values of an array or Collection property. Each individual value of the array or Collection is initialized using a value, null-value, or value-ref element. Here is an example:

```
<managed-bean>
  ...
  <managed-property>
     <property-name>cars</property-name>
     <values>
        <value-type>java.lang.Integer</value>
        <value>Jalopy</value>
        <value-ref>myCarsBean.luxuryCar</value-ref>
        <null-value/>
     </values>
  </managed-property>
</managed-bean>
```

This example initializes an array or a Collection. The type of the corresponding property in the bean determines which data structure is created. The values element defines the list of values in the array or Collection. The value element specifies a single value in the array or Collection. The value-ref element references a property in another bean. The null-value element will cause the property's set method to be called with an argument of null. A null property cannot be specified for a property whose data type is a Java primitive, such as int, or boolean.

Initializing Managed Bean Properties

Sometimes you might want to create a bean that also references other managed beans so that you can construct a graph or a tree of beans. For example, suppose that you want to create a bean representing a customer's information, including the mailing address and street address, each of which are also beans. The following managed-bean declarations create a CustomerBean instance that has two AddressBean properties, one representing the mailing address and the other representing the street address. This declaration results in a tree of beans with CustomerBean as its root and the two CustomerBean objects as children.

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>
     com.mycompany.mybeans.CustomerBean
```

```
      </managed-bean-class>
      <managed-bean-scope> request </managed-bean-scope>
      <managed-property>
        <property-name>mailingAddress</property-name>
        <value-ref>addressBean</value-ref>
      </managed-property>
      <managed-property>
        <property-name>streetAddress</property-name>
        <value-ref>addressBean</value-ref>
      </managed-property>
      <managed-property>
        <property-name>customerType</property-name>
        <value>New</value>
      </managed-property>
    </managed-bean>
    <managed-bean>
      <managed-bean-name>addressBean</managed-bean-name>
      <managed-bean-class>
        com.mycompany.mybeans.AddressBean
      </managed-bean-class>
      <managed-bean-scope> none </managed-bean-scope>
      <managed-property>
        <property-name>street</property-name>
        </null-value>
      <managed-property>
      ...
    </managed-bean>
```

The first CustomerBean declaration (with the managed-bean-name of customer) creates a CustomerBean in request scope. This bean has two properties, called `mailingAddress` and `shippingAddress`. These properties use the `value-ref` element to reference a bean, named `CustomerBean`.

The second managed bean declaration defines an `AddressBean`, but does not create it because its `managed-bean-scope` element defines a scope of none. Recall that a scope of none means that the bean is only created when something else references it. Since both the `mailingAddress` and `streetAddress` properties both reference `addressBean` using the `value-ref` element, two instances of `AddressBean` are created when `CustomerBean` is created.

When you create an object that points to other objects, do not try to point to an object with a shorter life span because it might be impossible to recover that scope's resources when it goes away. A session-scoped object, for example, cannot point to a request-scoped object. And objects with "none" scope have no effective life span managed by the framework, so they can only point to other "none" scoped objects. Table 3–4 on page 42 outlines all of the allowed connections:

Table 3–4

An object of this scope	**May point to a object of this scope**
none	none
application	none, application
session	none, application, session
request	none, application, session, request

Cycles are not permitted in forming these connections, in order to avoid issues involving order of initialization that would require a more complex implementation strategy.

Binding a Component to a Data Source

The `UIInput` and `UIOutput` components (and all components that extend these components) support storing a local value and referring to a value in another location with the optional `valueRef` attribute, which has replaced the `modelReference` attribute of previous releases. Like the `modelReference` attribute, the `valueRef` attribute is used to bind a component's data to data stored in another location.

Also like the `modelReference` attribute, the `valueRef` attribute can be used to bind a component's data to a JavaBeans component or one of its properties. What's different about the `valueRef` attribute is that it also allows you to map the component's data to any primitive (such as int), structure (such as an array), or collection (such as a list), independent of a JavaBeans component.

In addition to the `valueRef` attribute, this release also introduces the `actionRef` attribute, which binds an `Action` to a component. As explained in section Navigating Between Pages (page 105), an `Action` performs some logic and returns an outcome, which tells the navigation model what page to access next.

This section explains how the binding of a component to data works, and how to use `valueRef` to bind a component to a bean property and primitive, and how to combine the component data with an `Action`.

How Binding a Component to Data Works

Many of the standard components support storing local data, which is represented by the component's value property. They also support referencing data stored elsewhere, represented by the component's `valueRef` property.

Here is an example of using a `value` property to set an integer value:

```
value=”9”
```

Here is an example of using a `valueRef` property to refer to the bean property that stores the same integer:

```
valueRef=”order.quantity”
```

During the Apply Request Values phase of the standard request processing lifecycle, the component's local data is updated with the values from the current request. During this phase and the Process Validations phase, local values from the current request are checked against the converters and validators registered on the components

During the Update Model Values phase, the JavaServer Faces implementation copies the component's local data to the model data if the component has a valueRef property that points to a model object property.

During the Render Response phase, model data referred to by the component's `valueRef` property is accessed and rendered to the page.

The `valueRef` property uses an expression language syntax to reference the data bound to a component. Table 3–5 on page 44 shows a few examples of valid `valueRef` expressions.

Table 3–5 Example valueRef Expressions

Value	**valueRef Expression**
A property initialized from a context init parameter	initParam.quantity
A bean property	CarBean.engineOption
Value in an array	engines[3]
Value in a collection	CarPriceMap[“jalopy”]
Property of an object in an array of objects	cars[3].carPrice

The new `ValueBinding` API evaluates the `valueRef` expression that refers to a model object, a model object property, or other primitive or data structure.

A `ValueBinding` uses a `VariableResolver` to retrieve a value. The `VariableResolver` searches the scopes and implicit objects to retrieve the value. Implicit objects map parameters to values. For example, the integer literal, quantity, from Table 3–5 on page 44 is initialized as a property initialized from a context init parameter. The implicit objects that a `VariableResolver` searches are listed in Table 3–6 on page 44.

Table 3–6 Implicit Objects

Implicit object	**What it is**
applicationScope	A Map of the application scope attribute values, keyed by attribute name.
cookie	A Map of the cookie values for the current request, keyed by cookie name.
facesContext	The FacesContext instance for the current request.

Table 3–6 Implicit Objects

Implicit object	What it is
header	A Map of HTTP header values for the current request, keyed by header name.
headerValues	A Map of String arrays containing all of the header values for HTTP headers in the current request, keyed by header name.
initParam	A Map of the context initialization parameters for this web application.
param	A Map of the request parameters for this request, keyed by parameter name.
paramValues	A Map of String arrays containing all of the parameter values for request parameters in the current request, keyed by parameter name.
requestScope	A Map of the request attributes for this request, keyed by attribute name.
sessionScope	A Map of the session attributes for this request, keyed by attribute name.
tree	The root UIComponent in the current component tree stored in the Faces-Request for this request.

A `VariableResolver` also creates and stores objects in scope. The default `VariableResolver` resolves standard implicit variables and is the Managed Bean Facility, discussed in section Creating Model Objects (page 33). The Managed Bean Facility is configured with the application configuration resource file, faces-config.xml.

It's also possible to create a custom `VariableResolver`. There are many situations in which you would want to create a `VariableResolver`. One situation is if you don't want the web application to search a particular scope, or you want it to search only some of the scopes for performance purposes.

Binding a Component to a Bean Property

To bind a component to a bean or its property, you must first specify the name of the bean or property as the value of the `valueRef` attribute. You configured this bean in the application configuration file, as explained in section Creating Model

Objects (page 33). If you are binding the component to a bean or its property, the component tag's `valueRef` expression must match the corresponding `message-bean-name` element up to the first "." in the expression. Likewise, the part of the valueRef expression after the "." must match the name specified in the corresponding `property-name` element in the application configuration file. For example, consider this bean configuration:

```
<managed-bean>
  <managed-bean-name>CarBean</managed-bean-name>
  <managed-property>
    <property-name>carName</property-name>
    <value>Jalopy</value>
  </managed-property>
  ...
</managed-bean>
```

This example configures a bean called `CarBean`, which has a property called `carName` of type String. If there is already a matching instance of this bean in the specified scope, the JavaServer Faces implementation does not create it.

To bind a component to this bean property, you refer to the property using a reference expression from the `valueRef` attribute of the component's tag:

```
<h:output_text valueRef="CarBean.carName" />
```

See section Creating Model Objects (page 33) for information on how to configure beans in the application configuration file.

Writing Model Object Properties (page 76) explains in more detail how to write the model object properties for each of the component types.

Binding a Component to an Initial Default

As explained in How Binding a Component to Data Works (page 43), the `valueRef` property can refer to a value mapped in an implicit object.

Suppose that you have a set of pages that all display a version number in a `UIOutput` component. You can save this number in an implicit object. This way, all of the pages can reference it, rather than each page including it. To save `ver-`

sionNo as an initial default value in the context initParam implicit object set the context-param element in your web.xml file:

```
<context-param>
  <param-name>versionNo</param-name>
  <param-value>1.05</param-name>
</context-param>
```

To access the version number at the time the page is rendered, refer to the parameter from the version component tag's valueRef attribute:

```
<h:output_text id=version valueRef="initParam.versionNo"
```

Storing values to and retrieving values from other implicit objects are done in a similar way.

Combining Component Data and Action Objects

An Action is an object that performs application-specific processing when an ActionEvent occurs as a result of clicking a button or a hyperlink. The JavaServer Faces implementation automatically registers a default ActionListener to handle the Action Event.

The processing an Action object performs occurs in its invoke method, which returns a logical outcome as a result of the processing. For example, the invoke method can return "failure" after checking if a password a user enters does not match the password on file.

This outcome is returned to the default NavigationHandler by way of the default ActionListener implementation. The NavigationHandler selects the page to be accessed next by matching the outcome against those defined in a set of navigation rules specified in the application configuration file.

As the section Using an Action Object With a Navigation Rule (page 111) explains, the component that generated the ActionEvent maps to the Action object with its actionRef property. This property references a bean property that returns the Action object.

It is common practice to include the bean property and the Action implementation to which it refers within the same bean class. Additionally, this bean class should represent the model data for the entire form from which the ActionEvent originated. This is so that the Action object's invoke method has access to the form data and the bean's methods.

To illustrate how convenient it is to combine the form data and the Action object, consider the situation in which a user uses a form to log in to a Web site. This form's data is represented by LogonBean, which is configured in the application configuration file:

```
<managed-bean>
  <managed-bean-name>logonForm</managed-bean-name>
  <managed-bean-class>foo.LogonForm</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

This declaration creates the LogonForm bean in request scope for each individual request if the bean is not already in request scope. For more information on creating beans, see Creating Model Objects (page 33).

To logon, the user enters her username and password in the form. The following tags from the login.jsp page accept the username and password input:

```
<h:input_text id="username" size="16"
  valueRef="logonForm.username" />
<h:input_secret id="password" size="16"
  valueRef="logonForm.password"/>
```

The valueRef properties of these UIInput components refer to LogonForm bean properties. The data for these properties are updated when the user enters the username and password and submits the form by clicking the SUBMIT button. The button is rendered with this command_button tag:

```
<h:command_button id="submit" type="SUBMIT"
  label="Log On" actionRef="logonForm.logon" />
```

The actionRef property refers to the getLogon method of the LoginForm bean:

```
public Action getLogon() {
  return new Action() {
    public String invoke() {
      return (logon());
    }
  };
}
```

This method returns an `Action` (implemented here as an anonymous inner class), whose `invoke` method returns an outcome. This outcome is determined by the processing performed in the bean's `logon` method:

```
protected String logon() {
  // If the username is not found in the database, or the
     password does not match that stored for the username
        Add an error message to the FacesContext
        Return null to reload the current page.
  // else if the username and password are correct
     Save the username in the current session
     Return the outcome, “success”
}
```

The `logon` method must access the username and password that is stored in the username and password bean properties so that it can check them against the username and password stored in the database.

Using the JavaServer Faces Tag Libraries

JavaServer Faces technology provides two tag libraries: the `html_basic` tag library and the `jsf-core` tag library. The `html_basic` tag library defines tags for representing common HTML user interface components. The `jsf-core` tag library defines all of the other tags, including tags for registering listeners and validators on components. The tags in `jsf-core` are independent of any rendering technology and can therefore be used with any render kit. Using these tag libraries is similar to using any other custom tag library. This section assumes that you are familiar with the basics of *custom tag libraries*. If you are not, consult the *The Java Web Services Tutorial*.

Declaring the JavaServer Faces Tag Libraries

To use the JavaServer Faces tag libraries, you need to include these `taglib` directives at the top of each page that will contain the tags defined by these tag libraries:

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

The `uri` attribute value uniquely identifies the tag library. The `prefix` attribute value is used to distinguish tags belonging to the tag library. For example, the `form` tag must be referenced in the page with the `h` prefix, like this:

```
<h:form ...>
```

When you reference any of the JavaServer Faces tags from within a JSP page, you must enclose all of them in the `use_faces` tag, which is defined in the `jsf_core` library:

```
<f:use_faces>
  ... other faces tags, possibly mixed with other content ...
</f:use_faces>
```

You can enclose other content within the `use_faces` tag, including HTML and other JSP tags, but all JavaServer Faces tags must be enclosed within the `use_faces` tag.

Using the Core Tags

The tags defined by the `jsf-core` TLD represent a set of tags for performing core actions that are independent of a particular render kit. The `jsf-core` tags are listed in Table 3–7.

Table 3–7 The `jsf-core` Tags

	Tags	**Functions**
Event Handling Tags	`action_listener`	Registers an action listener on a parent component
	`valuechanged_listener`	Registers a value-changed listener on a parent component
Attribute Configuration Tag	`attribute`	Adds configurable attributes to a parent components
Facet Tag	`facet`	Signifies a nested component that has a special relationship to its enclosing tag
Parameter Substitution Tag	`parameter`	Substitutes parameters into a `MessageFormat` instance and to add query string name/value pairs to a URL.
Container For Form Tags	`use_faces`	Encloses all JavaServer Faces tags on this page.

Table 3–7 The `jsf-core` Tags (Continued)

	Tags	Functions
Validator Tags	`validate_doublerange`	Registers a `DoubleRangeValidator` on a component
	`validate_length`	Registers a `LengthValidator` on a component
	`validate_longrange`	Registers a `LongRangeValidator` on a component
	`validate_required`	Registers a `RequiredValidator` on a component
	`validate_stringrange`	Registers a `StringRangeValidator` on a component
	`validator`	Registers a custom `Validator` on a component

These tags are used in conjunction with component tags and are therefore explained in other sections of this tutorial. Table 3–8 lists which sections explain how to use which `jsf-core` tags.

Table 3–8 Where the `jsf-core` Tags are Explained

Tags	Where Explained
Event-Handling Tags	Handling Events (page 99)
`attribute` Tag	Using the Standard Converters (page 93)
`facet` Tag	Using the panel_grid Tag (page 64)
`parameter` Tag	Submitting ActionEvents (page 55), Linking to a URL (page 57), and Using the output_message Tag (page 63)
`use_faces` Tag	Declaring the JavaServer Faces Tag Libraries (page 50)
Validator Tags	Performing Validation (page 81)

Using the HTML Tags

The tags defined by `html_basic` represent HTML form controls and other basic HTML elements. These controls display data or accept data from the user. This data is collected as part of a form and is submitted to the server, usually when the user clicks a button. This section explains how to use each of the component tags shown in Table 2–2, and is organized according to the `UIComponent` classes from which the tags are derived.

This section does not explain every tag attribute, only the most commonly-used ones. Please refer to `html_basic.tld` file in the `lib` directory of your download for a complete list of tags and their attributes.

In general, most of the component tags have these attributes in common:

- `id`: uniquely identifies the component
- `valueRef`: identifies the data source mapped to the component
- `key`: identifies a key in a resource bundle.
- `bundle`: identifies a resource bundle

In this release, the `id` attribute is not required for a component tag except in these situations:

- Another component or a server-side class must refer to the component
- The component tag is impacted by a JSTL conditional or iterator tag (for more information, see *The Java Web Services Tutorial*).

If you don't include an `id` attribute, the JavaServer Faces implementation automatically generates a component ID.

UIOutput and subclasses of UIOutput have a `valueRef` attribute, which is always optional, except in the case of `SelectItems`. Using the value-ref attribute to bind to a data source is explained more in section Using the Core Tags (page 51).

The UIForm Component

A `UIForm` component is an input form with child components representing data that is either presented to the user or submitted with the form. The `form` tag encloses all of the controls that display or collect data from the user. Here is the `form` tag from the `ImageMap.jsp` page:

```
<h:form formName="imageMapForm"
... other faces tags and other content...
</h:form>
```

The `formName` attribute is passed to the application, where it is used to select the appropriate business logic.

The `form` tag can also include HTML markup to layout the controls on the page. The `form` tag itself does not perform any layout; its purpose is to collect data and to declare attributes that can be used by other components in the form.

The UICommand Component

The `UICommand` component performs an action when it is activated. The most common example of such a component is the button. This release supports `Button` and `Hyperlink` as `UICommand` component renderers.

Using the command_button Tag

Most pages in the `cardemo` example use the `command_button` tag. When the button is clicked, the data from the current page is processed, and the next page is opened. Here is the `buyButton` `command_button` tag from `buy.jsp`:

```
<h:command_button key="buy" bundle="carDemoBundle"
            commandName="customer" action="success" />
```

Clicking the button will cause `Customer.jsp` to open. This page allows you to fill in your name and shipping information.

The `key` attribute references the localized message for the button's label. The `bundle` attribute references the `ResourceBundle` that contains a set of localized messages. For more information on localizing JavaServer Faces applications, see Performing Localization (page 112).

The `commandName` attribute refers to the name of the command generated by the event of clicking the button. The `commandName` is used by the `ActionEventListener` to determine how to process the command. See Handling Events (page 99) for more information on how to implement event listeners to process the event generated by button components.

The `action` attribute represents a literal outcome value returned when the button is clicked. The outcome is passed to the default `NavigationHandler`, which matches the outcome against a set of navigation rules defined in the application configuration file.

A `command_button` tag can have an `actionRef` attribute as an alternative to the `action` attribute. The `actionRef` attribute is a value reference expression that points to an Action, whose invoke method performs some processing and returns the logical outcome.

See section Navigating Between Pages (page 105) for information on how to use the action and actionRef attributes.

The `cardemo` application uses the `commandName` and the `action` attributes together. This is because it uses the outcome from the `action` attribute to navigate between pages, but it also uses the `commandName` attribute to point to a listener that performs some other processing. In practice, this extra processing should be performed by the `Action` object, and the `actionRef` attribute should be used to point to the `Action` object. The `commandName` attribute and its associated listener should only be used to process UI changes that don't result in a page being loaded.

Using the command_hyperlink Tag

The command_hyperlink tag represents an HTML hyperlink and is rendered as an HTML `<a>` element. The `command_hyperlink` tag can be used for two purposes:

- To submit `ActionEvents` to the application. See Handling Events (page 99) and Navigating Between Pages (page 105) for more information.
- To link to a particular URL

Submitting ActionEvents

Like the command_button tag, the `command_hyperlink` tag can be used to submit `ActionEvents`. To submit a ActionEvent for the purpose of navigating between pages, the tag needs one of these attributes:

- `action`, which indicates a logical outcome for determining the next page to be accessed
- `actionRef`, which refers to the bean property that returns an Action in response to the event of clicking the hyperlink

The `action` attribute represents a literal outcome value returned when the hyperlink is clicked. The outcome is passed to the default `NavigationHandler`, which matches the outcome against a set of navigation rules defined in the application configuration file.

The `actionRef` attribute is a value reference expression that points to an `Action`, whose `invoke` method performs some processing and returns the logical outcome.

See section Navigating Between Pages (page 105) for information on how to use the `action` and `actionRef` attributes.

To submit an `ActionEvent` for the purpose of making UI changes, the tag needs both of these attributes:

- `commandName`: the logical name of the command
- `commandClass`: the name of the listener that handles the event

The `commandName` attribute refers to the name of the command generated by the event of clicking the hyperlink. The `commandName` is used by the `ActionEventListener` to determine how to process the command. See Handling Events (page 99) for more information on how to implement event listeners to process the event generated by button components.

The `commandName` attribute and its associated listener should only be used to process UI changes that don't result in a page being loaded. See Registering Listeners on Components (page 103) for more information on using the `commandName` attribute.

In addition to these attributes, the tag also needs a `label` attribute, which is the text that the user clicks to generate the event and either:

A `command_hyperlink` tag can contain `parameter` tags that will cause an HTML `<input type=hidden>` element to be rendered. This `input` tag represents a hidden control that stores the name and value specified in the `parameter` tags between client/server exchanges so that the server-side classes can retrieve the value. The following two tags show `command_hyperlink` tags that submit `ActionEvents`. The first tag does not use parameters; the second tag does use parameters.

```
<h:command_hyperlink id="commandParamLink" commandName="login"
  commandClass="LoginListener" label="link text"/>
<h:command_hyperlink id="commandParamLink" commandName="login"
  commandClass="LoginListener" label="Login">
  <f:parameter id="Param1" name="name"
```

```
        valueRef=”LoginBean.name”/>
      <f:parameter id="Param2" name="value"
        valueRef="LoginBean.password"/>
    </h:command_hyperlink>
```

The first tag renders this HTML:

```
    <a href="#"
    onmousedown="document.forms[0].commandParamLink.value='login';
    document.forms[0].submit()" class="hyperlinkClass">
      link text</a>
      <input type="hidden" name="commandParamLink"/>
```

The second tag renders this HTML, assuming that `LoginBean.name` is `duke` and `LoginBean.password` is `redNose`:

```
    <a href="#"
    onmousedown="document.forms[0].commandParamLink.value='login';
    document.forms[0].submit()" class="hyperlinkClass">
      link text</a>
      <input type="hidden" name="commandParamLink"/>
      <input type="hidden" name="name" value="duke"/>
      <input type="hidden" name="value" value="redNose"/>
```

Note: Notice that the `command_hyperlink` tag that submits `ActionEvents` will render JavaScript. If you use this tag, make sure your browser is JavaScript-enabled.

Linking to a URL

To use `command_hyperlink` to link to a URL, your `command_hyperlink` tag must specify the `href` attribute, indicating the page to which to link.

A `command_hyperlink` that links to a URL can also contain `parameter` tags. The `parameter` tags for this kind of `command_link` tag allow the page author to add query strings to the URL. The following two tags show `command_hyperlink` tags that link to a URL. The first tag does not use parameters; the second tag does use parameters.

```
    <h:command_hyperlink id="hrefLink" href="welcome.html"
      image="duke.gif"/>
    <h:command_hyperlink id="hrefParamLink" href="welcome.html"
      image="duke.gif">
      <f:parameter id="Param1" name="name"
```

```
        valueRef=”LoginBean.name”/>
      <f:parameter id="Param2" name="value"
        valueRef="LoginBean.password"/>
    </h:command_hyperlink>
```

The first tag renders this HTML:

```
    <a href="hello.html"><img src="duke.gif"></a>
```

The second tag renders this HTML, assuming that `LoginBean.name` is duke and `LoginBean.password` is redNose:

```
    <a href="hello.html?name=duke&value=redNose">
      <img src="duke.gif"></a>
```

The UIGraphic Component

The `UIGraphic` component displays an image. The `cardemo` application has many examples of `graphic_image` tags. Here is the `graphic_image` tag used with the image map on `ImageMap.jsp`:

```
    <h:graphic_image id="mapImage" url="/world.jpg"
      usemap="#worldMap" />
```

The `url` attribute specifies the path to the image. It also corresponds to the local value of the `UIGraphic` component so that the URL can be retrieved with the `currentValue` method or indirectly from a model object. The URL of the example tag begins with a “/”, which adds the relative context path of the Web application to the beginning of the path to the image.

The `usemap` attribute refers to the image map defined by the custom `UIMap` component on the same page. See Creating Custom UI Components (page 117) for more information on the image map.

The UIInput and UIOutput Components

The `UIInput` component displays a value to a user and allows the user to modify this data. The most common example is a text field. The `UIOutput` component displays data that cannot be modified. The most common example is a label.

Both `UIInput` and `UIOutput` components can be rendered in several different ways. Since the components have some common functionality, they share many of the same renderers.

Table 3–9 lists the common renderers of `UIInput` and `UIOutput`. Recall from The Component Rendering Model (page 20) that the tags are composed of the component and the renderer. For example, the `input_text` tag refers to a `UIInput` component that is rendered with the `Text` Renderer.

Table 3–9 UIInput and UIOutput Renderers

Renderer	Tag	Function
Date	`input_date`	Accepts a `java.util.Date` formatted with a `java.text.Date` instance
	`output_date`	Displays a `java.util.Date` formatted with a `java.text.Date` instance
DateTime	`input_datetime`	Accepts a `java.util.Date` formatted with a `java.text.DateTime` instance
	`output_datetime`	Displays a `java.util.Date` formatted with a `java.text.DateTime` instance
Number	`input_number`	Accepts a numeric data type (`java.lang.Number` or primitive), formatted with a `java.text.NumberFormat`
	`output_number`	Accepts a numeric data type (`java.lang.Number` or primitive), formatted with a `java.text.NumberFormat`
Text	`input_text`	Accepts a text string of one line.
	`output_text`	Displays a text string of one line.
Time	`input_time`	Accepts a `java.util.Date`, formatted with a `java.text.DateFormat` time instance
	`output_time`	Displays a `java.util.Date`, formatted with a `java.text.DateFormat` time instance

In addition to the renderers listed in Table 3–9, `UIInput` and `UIOutput` each support other renderers that the other component does not support. These are listed in Table 3–10.

Table 3–10 Additional UIInput and UIOutput Renderers

Component	Renderer	Tag	Function
`UIInput`	`Hidden`	`input_hidden`	Allows a page author to include a hidden variable in a page
	`Secret`	`input_secret`	Accepts one line of text with no spaces and displays it as a set of asterisks as it is typed
	`TextArea`	`input_textarea`	Accepts multiple lines of text
`UIOutput`	`Errors`	`output_errors`	Displays error messages for an entire page or error messages associated with a specified client identifier
	`Label`	`output_label`	Displays a nested component as a label for a specified input field
	`Message`	`output_message`	Displays a localized message

All of the tags listed in Table 3–9—except for the `input_text` and `output_text` tags—display or accept data of a particular format specified in the `java.text` or `java.util` packages. You can also apply the `Date`, `DateTime`, `Number`, and `Time` renderers associated with these tags to convert data associated with the `input_text`, `output_text`, `input_hidden`, and `input_secret` tags. See Performing Data Conversions (page 92) for more information on using these renderers as converters.

The rest of this section explains how to use selected tags listed in the two tables above. These tags are: `input_datetime`, `output_datetime`, `output_label`, `output_message`, `input_secret`, `output_text`, and `input_text`.

The `output_errors` tag is explained in Performing Validation (page 81). The tags associated with the `Date`, `Number`, and `Time` renderers are defined in a similar way to those tags associated with the `DateTime` renderer. The `input_hidden` and `input_textarea` tags are similar to the `input_text` tag. Refer to the html_basic TLD in your download to see what attributes are supported for these extra tags.

Using the input_datetime and output_datetime Tags

The DateTime renderer can render both UIInput and UIOutput components. The input_datetime tag displays and accepts data in a java.text.SimpleDateFormat. The output_datetime tag displays data in a java.text.SimpleDateFormat. This section shows you how to use the output_datetime tag. The input_datetime tag is written in a similar way.

The output_datetime and input_datetime tags have the following attributes and values for formatting data:

- dateStyle: short(default), medium, long, full
- timeStyle: short(default), medium, long, full
- timezone: short(default), long
- formatPattern: a String specifying the format of the data

See java.text.SimpleDateFormat and java.util.TimeZone for information on specifying the style of dateStyle, timeStyle, and timezone. You can use the first three attributes in the same tag simultaneously or separately. Or, you can simply use formatPattern to specify a String pattern to format the data. The following tag is an example of using the formatPattern attribute:

```
<h:output_datetime
  formatPattern="EEEEEEEE, MMM d, yyyy hh:mm:ss a z"
  valueRef="LoginBean.date"/>
```

One example of a date and time that this tag can display is:

```
Saturday, Feb 22, 2003 18:10:15 pm PDT
```

You can also display the same date and time with this tag:

```
<h: output_datetime dateStyle=”full” timeStyle=”long”
  valueRef=”LoginBean.date” />
```

The application developer is responsible for ensuring that the LoginBean.date property is the proper type to handle these formats.

The tags corresponding to the Date, Number, and Time renderers are written in a similar way. See the html_basic TLD in the lib directory of your installation to look up the attributes supported by the tags corresponding to these renderers.

Using the output_text and input_text Tags

The Text renderer can render both UIInput and UIOutput components. The input_text tag displays and accepts a single-line string. The output_text tag displays a single-line string. This section shows you how to use the input_text tag. The output_text tag is written in a similar way.

The following attributes, supported by both output_text and input_text, are likely to be the most commonly used:

- id: Identifies the component associated with this tag
- valueRef: Identifies the model object property bound to the component
- converter: Identifies one of the renderers that will be used to convert the component's local data to the model object property data specified in the valueRef attribute. See Performing Data Conversions (page 92) for more information on how to use this attribute.
- value: Allows the page author to specify the local value of the component.

The output_text tag also supports the key and bundle attributes, which are used to fetch the localized version of the component's local value. See Performing Localization (page 112) for more information on how to use these attributes.

Here is an example of an input_text tag from the Customer.jsp page:

```
<h:input_text valueRef="CustomerBean.firstName" />
```

The valueRef value refers to the firstName property on the CustomerBean model object. After the user submits the form, the value of the firstName property in CustomerBean will be set to the text entered in this field.

Using the output_label Tag

The output_label tag is used to attach a label to a specified input field for accessibility purposes. Here is an example of an output_label tag:

```
<h:output_label for="firstName">
  <h:output_text id="firstNameLabel" value="First Name"/>
</h:output_label>
...
<h:input_text id="firstName" />
```

The for attribute maps to the id of the input field to which the label is attached. The output_text tag nested inside the output_label tag represents the actual label. The `value` attribute on the `output_text` tag indicates the label that is displayed next to the input field.

Using the output_message Tag

The `output_message` tag allows a page author to display concatenated messages as a `MessageFormat` pattern. Here is an example of an `output_message` tag:

```
<h:output_message
  value="Goodbye, {0}. Thanks for ordering your {1} " >
  <f:parameter id="param1" valueRef="LoginBean.name"/>
  <f:parameter id="param2" valueRef="OrderBean.item" />
</h:output_message>
```

The `value` attribute specifies the `MessageFormat` pattern. The `parameter` tags specify the substitution parameters for the message. The `valueRef` for `param1` maps to the user's name in the `LoginBean`. This value replaces `{0}` in the message. The `valueRef` for `param2` maps to the item the user ordered in the `OrderBean`. This value replaces `{1}` in the message. Make sure you put the `parameter` tags in the proper order so that the data is inserted in the correct place in the message.

Instead of using `valueRef`, a page author can hardcode the data to be substituted in the message by using the `value` attribute on the `parameter` tag.

Using the input_secret Tag

The `input_secret` tag renders an `<input type="password">` HTML tag. When the user types a string in this field, a row of asterisks is displayed instead of the string the user types. Here is an example of an `input_secret` tag:

```
<h:input_secret redisplay="false"
  valueRef="LoginBean.password" />
```

In this example, the `redisplay` attribute is set to false. This will prevent the password from being displayed in a query string or in the source file of the resulting HTML page.

The UIPanel Component

The `UIPanel` class extends `UIOutput`. A `UIPanel` component is used as a layout container for its children. When using the renderers from the HTML render kit, a

UIPanel is rendered as an HTML table. Table 3–11 lists all of the renderers and tags corresponding to the UIPanel component.

Table 3–11 UIPanel Renderers and Tags

Renderer	**Tag**	**Renderer Attributes**	**Function**
Data	panel_data	var	Iterates over a collection of data, rendered as a set of rows
Grid	panel_grid	columnClasses, columns, footerClass, headerClass, panelClass, rowClasses	Displays a table
Group	panel_group		Groups a set of components under one parent
List	panel_list	columnClasses, footerClass, headerClass, panelClass, rowClasses	Displays a table of data that comes from a Collection, array, Iterator, or Map

The panel_grid and panel_list tags are used to represent entire tables. The panel_data tags and panel_group tags are used to represent rows in the tables. To represent individual cells in the rows, the output_text tag is usually used, but any output component tag can be used to represent a cell.

A panel_data tag can only be used in a panel_list. A panel_group can be used in both panel_grid tags and panel_list tags. The next two sections show you how to create tables with panel_grid and panel_list, and how to use the panel_data and panel_group tags to generate rows for the tables.

Using the panel_grid Tag

The panel_grid tag has a set of attributes that specify CSS stylesheet classes: the columnClasses, footerClass, headerClass, panelClass, and rowClasses. These stylesheet attributes are not required.

The panel_grid tag also has a columns attribute. The columns attribute is required if you want your table to have more than one column because the columns attribute tells the renderer how to group the data in the table.

If a headerClass is specified, the panel_grid must have a header as its first child. Similarly, if a footerClass is specified, the panel_grid must have a footer as its last child.

The cardemo application includes one panel_grid tag on the buy.jsp page:

```
<h:panel_grid id="choicesPanel" columns="2"
  footerClass="subtitle" headerClass="subtitlebig"
  panelClass="medium"
  columnClasses="subtitle,medium">
  <f:facet name="header">
     <h:panel_group>
        <h:output_text  key="buyTitle" bundle="carDemoBundle"/>
     </h:panel_group>
  </f:facet>
  <h:output_text key="Engine" bundle="carDemoBundle" />
  <h:output_text
     valueRef=
        "CurrentOptionServer.currentEngineOption"/>
  ...
  <h:output_text  key="gpsLabel" bundle="carDemoBundle"  />
  <h:output_text  valueRef="CurrentOptionServer.gps" />
  <f:facet name="footer">
     <h:panel_group>
        <h:output_text  key="yourPriceLabel"
           bundle="carDemoBundle"  />
        <h:output_text
           valueRef="CurrentOptionServer.packagePrice" />
     </h:panel_group>
  </f:facet>
</h:panel_grid>
```

This panel_grid is rendered to a table that lists all of the options that the user chose on the previous page, more.jsp. This panel_grid uses stylesheet classes to format the table. The CSS classes are defined in the stylesheet.css file in the example/cardemo/web directory of your download. The subtitlebig definition is:

```
.subtitlebig {
  font-family: Arial, Helvetica, sans-serif;
  font-size: 14px;
  color: #93B629;
  padding-top: 10;
  padding-bottom: 10;
}
```

Since the panel_grid tag specifies a headerClass and a footerClass, the panel_grid must contain a header and footer. Usually, a facet tag is used to represent headers and footers. This is because header and footer data is usually static.

A facet is used to represent a component that is independent of the parent-child relationship of the page's component tree. Since header and footer data is static, the elements representing headers and footers should not be updated like the rest of the components in the tree.

This panel_grid uses a facet tag for both the headers and footers. Facets can only have one child, and so a panel_group tag is needed to group more than one element within a facet. In the case of the header facet, a panel_group tag is not really needed. This tag could be written like this:

```
<f:facet name="header">
  <h:output_text key="buyTitle" bundle="carDemoBundle"/>
</f:facet>
```

The panel_group tag is needed within the footer facet tag because the footer requires two cells of data, represented by the two output_text tags within the panel_group tag:

```
<f:facet name="footer">
  <h:panel_group>
    <h:output_text  key="yourPriceLabel"
      bundle="carDemoBundle"  />
    <h:output_text
      valueRef="CurrentOptionServer.packagePrice" />
  </h:panel_group>
</f:facet>
```

A panel_group tag can also be used to encapsulate a nested tree of components so that the parent thinks of it as a single component.

In between the header and footer facet tags, are the output_text tags, each of which represents a cell of data in the table:

```
<h:output_text key="Engine" bundle="carDemoBundle" />
<h:output_text
  valueRef=
    "CurrentOptionServer.currentEngineOption"/>
...
<h:output_text  key="gpsLabel" bundle="carDemoBundle"  />
<h:output_text  valueRef="CurrentOptionServer.gps" />
```

Again, the data represented by the output_text tags is grouped into rows according to the value of the columns attribute of the output_text tag. The columns attribute in the example is set to “2”. So from the list of output_text tags representing the table data, the data from the odd output_text tags is rendered in the first column and the data from the even output_text tags is rendered in the second column.

Using the panel_list Tag

The panel_list tag has the same set of stylesheet attributes as panel_grid, but it does not have a columns attribute. The number of columns in the table equals the number of output_text (or other component tag) elements within the panel_data tag, which is nested inside the panel_list tag. The panel_data tag iterates over a Collection, array, Iterator, or Map of model objects. Each output_text tag nested in a panel_data tag maps to a particular property of each of the model objects in the list. Here is an example of a panel_list tag:

```
<h:panel_list id="Accounts" >
  <f:facet name="header">
    <h:panel_group>
      <h:output_text id="acctHead" value="Account Id"/>
      <h:output_text id="nameHead" value="Customer Name"/>
      <h:output_text id="symbolHead" value="Symbol"/>
      <h:output_text id="tlSlsHead" value="Total Sales"/>
    </h:panel_group>
  </f:facet>
  <h:panel_data id="tblData" var="customer"
    valueRef="CustomerBean">
    <h:output_text id="acctId"
      valueRef="customer.acctId"/>
    <h:output_text id="name" valueRef="customer.name"/>
    <h:output_text id="symbol"
      valueRef="customer.symbol"/>
```

```
        <h:output_text id="tlSls"
          valueRef="customer.totalSales"/>
      </h:panel_data>
    </h:panel_list>
```

This example uses a `facet` tag, and a set of `output_text` tags nested inside a `panel_group` tag to represent a header row. See the previous section for a description of using facets and `panel_group` tags.

The component represented by the `panel_data` tag maps to a bean that is a `Collection`, `array`, `Iterator`, or `Map` of beans. The `valueRef` attribute refers to this bean, called `CustomerBean`. The `var` attribute refers to the current bean in the `CustomerBean` list. In this example, the current bean is called `customer`. Each component represented by an `output_text` tag maps to a property on the `customer` bean.

The `panel_data` tag's purpose is to iterate over the model objects and allow the `output_text` tags to render the data from each bean in the list. Each iteration over the list of beans will produce one row of data.

One example table that can be produced by this `panel_list` tag is:

Table 3–12 Example Accounts Table

Account Id	**Customer Name**	**Symbol**	**Total Sales**
123456	Sun Microsystems, Inc.	SUNW	2345.60
789101	ABC Company	ABC	458.21

The UISelectBoolean Component

The `UISelectBoolean` class defines components that have a `boolean` value. The `selectboolean_checkbox` tag is the only tag that JavaServer Faces technology provides for representing boolean state. The `more.jsp` page has a set of selectboolean_checkbox tags. Here is the one representing the cruisecontrol component:

```
<h:selectboolean_checkbox id="cruisecontrol"
  title="Cruise Control"
  valueRef="CurrentOptionServer.cruiseControlSelected" >
  <f:valuechanged_listener
    type="cardemo.PackageValueChanged"/>
</h:selectboolean_checkbox>
```

The id attribute value refers to the component object. The label attribute value is what is displayed next to the checkbox. The valueRef attribute refers to the model object property associated with the component. The property that a selectboolean_checkbox tag maps to should be of type boolean, since a checkbox represents a boolean value.

The UISelectMany Component

The UISelectMany class defines components that allow the user to select zero or more values from a set of values. This component can be rendered as a checkboxlist, a listbox, or a menu. This section explains the selectmany_checkboxlist and selectmany_menu tags. The selectmany_listbox tag is similar to the selectmany_menu tag, except selectmany_listbox does not have a size attribute since a listbox displays all items at once.

Using the selectmany_checkboxlist Tag

The selectmany_checkboxlist tag renders a set of checkboxes, one for each value that can be selected. The cardemo does not have an example of a selectmany_checkboxlist tag, but this tag can be used to render the checkboxes on the more.jsp page:

```
<h:selectmany_checkboxlist
  valueRef="CurrentOptionServer.currentOptions">
  <h:selectitem itemLabel="Sunroof"
    valueRef="CurrentOptionServer.sunRoofSelected">
  <f:valuechanged_listener
    type="cardemo.PackageValueChanged" />
  </h:selectitem>
  <h:selectitem itemLabel="Cruise Control"
    valueRef=
      "CurrentOptionServer.cruiseControlSelected" >
    <f:valuechanged_listener
      type="cardemo.PackageValueChanged" />
    </h:selectitem>
  </h:selectmany_checkboxlist>
```

The `valueRef` attribute identifies the model object property, `currentOptions`, for the current set of options. This property holds the values of the currently selected items from the set of checkboxes.

The `selectmany_checkboxlist` tag must also contain a tag or set of tags representing the set of checkboxes. To represent a set of items, you use the `selec-titems` tag. To represent each item individually, use a `selectitem` tag for each item. The UISelectItem and UISelectItems Classes (page 72) section explains these two tags in more detail.

Using the selectmany_menu Tag

The `selectmany_menu` tag represents a component that contains a list of items, from which a user can choose one or more items. The menu is also commonly known as a drop-down list or a combo box. The tag representing the entire list is the `selectmany_menu` tag. Here is an example of a `selectmany_menu` tag:

```
<h:selectmany_menu  id="fruitOptions"
  valueRef="FruitOptionBean.chosenFruits">
  <h:selectitems
    valueRef="FruitOptionBean.allFruits"/>
</h:selectmany_menu>
```

The attributes of the `selectmany_menu` tag are the same as those of the `selectmany_checkboxlist` tag. Again, the `valueRef` of the `selectmany_menu` tag maps to the property that holds the currently selected items' values. A `selectmany_menu` tag can also have a `size` attribute, whose value specifies how many items will display at one time in the menu. When the `size` attribute is set, the menu will render with a scrollbar for scrolling through the displayed items.

Like the `selectmany_checkboxlist` tag, the `selectmany_menu` tag must contain either a `selectitems` tag or a set of `selectitem` tags for representing the items in the list. The `valueRef` attribute of the `selectitems` tag in the example maps to the property that holds all of the items in the menu. The UISelectItem and UISelectItems Classes (page 72) explains these two tags.

The UISelectOne Component

The UISelectOne class defines components that allow the user to select one value from a set of values. This component can be rendered as a listbox, a radio button, or a menu. The cardemo example uses the selectone_radio and selectone_menu tags. The selectone_listbox tag is similar to the selectone_menu tag, except selectone_listbox does not have a size attribute since a listbox displays all items at once. This section explains how to use the selectone_radio and selectone_menu tags.

Using the selectone_radio Tag

The selectone_radio tag renders a set of radio buttons, one for each value that can be selected. Here is a selectone_radio tag from more.jsp that allows you to select a brake option:

```
<h:selectone_radio  id="currentBrake"
  valueRef="CurrentOptionServer.currentBrakeOption">
  <f:valuechanged_listener
    type="cardemo.PackageValueChanged"/>
  <h:selectitems
    valueRef="CurrentOptionServer.brakeOption"/>
</h:selectone_radio>
```

The id attribute of the selectone_radio tag uniquely identifies the radio group. The id is only required if another component, model object, or listener must refer to this component; otherwise, the JavaServer Faces implementation will generate a component id for you.

The valueRef attribute identifies the model object property for brakeOption, which is currentBrakeOption. This property holds the value of the currently selected item from the set of radio buttons. The currentBrakeOption property can be any of the types supported by JavaServer Faces technology.

The selectone_radio tag must also contain a tag or set of tags representing the list of items contained in the radio group. To represent a set of tags, you use the selectitems tag. To represent each item individually, use a selectitem tag for each item. The UISelectItem and UISelectItems Classes (page 72) explains these two tags in more detail.

Using the selectone_menu Tag

The selectone_menu tag represents a component that contains a list of items, from which a user can choose one item. The menu is also commonly known as a drop-down list or a combo box. An option list is a little different from a radio group because all selectable items are contained in one component; whereas a radio group consists of a set of distinct components. The tag representing the entire list is the `selectone_menu` tag. Here is the `selectone_menu` tag from the `more.jsp` page:

```
<h:selectone_menu  id="currentEngine"
  valueRef="CurrentOptionServer.currentEngineOption">
  <f:valuechanged_listener
    type="cardemo.PackageValueChanged" />
  <h:selectitems
    valueRef="CurrentOptionServer.engineOption"/>
</h:selectone_menu>
```

The attributes of the `selectone_menu` tag are the same as those of the `selectone_radio` tag. Again, the `valueRef` of the `selectone_menu` tag maps to the property that holds the currently selected item's value. A `selectone_menu` tag can also have a size attribute, whose value specifies how many items will display at one time in the menu. When the `size` attribute is set, the menu will render with a scrollbar for scrolling through the displayed items.

Like the `selectone_radio` tag, the `selectone_menu` tag must contain either a `selectitems` tag or a set of `selectitem` tags for representing the items in the list. The UISelectItem and UISelectItems Classes (page 72) section explains these two tags.

The UISelectItem and UISelectItems Classes

The `UISelectItem` and the `UISelectItems` classes represent components that can be nested inside a `UISelectOne` or a `UISelectMany` component. The `UISelectItem` is associated with a `SelectItem` instance, which contains the value, label, and description of a single item in the `UISelectOne` or `UISelectMany` component. The `UISelectItems` class represents a set of `SelectItem` instances, containing the values, labels, and descriptions of the entire list of items.

The `selectitem` tag represents a `UISelectItem` component. The `selectitems` tag represents a `UISelectItems` component. You can use either a set of `selectitem` tags or a single `selectitems` tag within your `selectone` or `selectmany` tags.

The advantages of using `selectitems` are

- You can represent the items using different data structures, including `Array`, `Map`, `List`, and `Collection`. The data structure is composed of `SelectItem` instances.
- You can dynamically generate a list of values at runtime.

The advantages of using `selectitem` are:

- The page author can define the items in the list from the page.
- You have less code to write in the model object for the `selectitem` properties.

For more information on writing model object properties for the UI`SelectItems` components, see Writing Model Object Properties (page 76). The rest of this section shows you how to use the `selectitems` and `selectitem` tags.

The selectitems Tag

Here is the `selectone_menu` tag from The UISelectOne Component (page 71):

```
<h:selectone_menu  id="currentEngine"
  valueRef="CurrentOptionServer.currentEngineOption">
  <f:valuechanged_listener
    type="cardemo.PackageValueChanged" />
  <h:selectitems
    valueRef="CurrentOptionServer.engineOption"/>
</h:selectone_menu>
```

The `id` attribute of the `selectitems` tag refers to the `UISelectItems` component object.

The `valueRef` attribute binds the `selectitems` tag to the `engineOption` property of `CurrentOptionServer`.

In the `CurrentOptionServer`, the `engineOption` property has a type of `ArrayList`:

```
engineOption = new ArrayList(engines.length);
```

`UISelectItems` is a collection of `SelectItem` instances. You can see this by noting how the `engineOption` `ArrayList` is populated:

```
for (i = 0; i < engines.length; i++) {
  engineOption.add(new SelectItem(engines[i], engines[i],
              engines[i]));
}
```

The arguments to the `SelectItem` are:

- An `Object` representing the value of the item
- A `String` representing the label that displays in the `UISelectOne` component on the page
- A `String` representing the description of the item

UISelectItems Properties (page 80) describes in more detail how to write a model object property for a `UISelectItems` component

The selectitem Tag

The `cardemo` application contains a few examples of `selectitem` tags, but let's see how the `engineOption` tag would look if you used `selectitem` instead of `selectitems`:

```
<h:selectone_menu id="engineOption"
valueRef="CurrentOptionServer.currentEngineOption">
  <h:selectitem
    itemValue=”v4” itemLabel=”v4”/>
  <h:selectitem
    itemValue=”v6” itemLabel=”v6”/>
  <h:selectitem
    itemValue=”v8” itemLabel=”v8”/>
</h:selectone_menu>
```

The `selectone_menu` tag is exactly the same and maps to the same property, representing the currently selected item.

The `itemValue` attribute represents the default value of the `SelectItem` instance. The `itemLabel` attribute represents the `String` that appears in the dropdown list component on the page.

You can also use a `valueRef` attribute instead of the `itemValue` attribute to represent the value of the item.

Writing a Model Object Class

A model object is a JavaBeans component that encapsulates the data on a set of components. It might also perform the application-specific functionality associated with the component data. For example, a model object might perform a currency conversion using a value that the user enters into `UIInput` component and then output the conversion to an `UIOutput` component. The model object follows JavaBeans component conventions in that it must contain an empty constructor and a set of properties for setting and getting the data, like this:

```
...
String myBeanProperty = null;
...
public MyBean() {}
String getMyBeanProperty{
  return myBeanProperty;
}
void setMyBeanProperty(String beanProperty){
  myBeanProperty = beanProperty;
}
```

You can bind most of the component classes to model object properties, but you are not required to do so.

In order to bind a component to a model object property, the type of the property must match the type of the component object to which it is bound. In other words, if a model object property is bound to a `UISelectBoolean` component, the property should accept and return a `boolean` value. The rest of this section explains how to write properties that can be bound to the component classes described in Using the HTML Tags (page 53).

Writing Model Object Properties

Table 3–13 lists all the component classes described in Using the HTML Tags (page 53) and the acceptable types of their values.

Table 3–13 Acceptable Component Types

Component	**Renderer**	**Types**
`UIInput/ UIOutput`	`Date`	`java.util.Date`
	`DateTime`	`java.util.Date`
	`Number`	`java.lang.Number`
	`Time`	`java.util.Date`
	`Text`	`java.lang.String` With a standard converter: `Date` and `Number`
`UIInput`	`Hidden`	`java.lang.String` With a standard converter: `Date` and `Number`
	`Secret`	`java.lang.String` With a standard converter: `Date` and `Number`
`UIOutput`	`Message`	`java.lang.String`
`UIPanel`	`Data`	`array,java.util.Collection, java.util.Iterator, java.util.Map`
`UISelectBoolean`	`Checkbox`	`boolean`
`UISelectItem`		`java.lang.String`
`UISelectItems`		`java.lang.String, Collection, Array, Map`
`UISelectMany`	`CheckboxList, Listbox, Menu`	`Collection, Array`
`UISelectOne`	`Listbox, Menu, Radio`	`java.lang.String,int, double, long`

Make sure to use the `valueRef` attribute in the tags of the components mapped to properties. Also, be sure to use the proper names of the properties. For example, if a `valueRef` tag has a value of `CurrentOptionServer.currentOption`, the corresponding `String` property should be:

```
String currentOption = null;
String getCurrentOption(){...}
void setCurrentOption(String option){...}
```

For more information on JavaBeans conventions, see *JavaBeans Components in JSP Pages* in *The Java Web Services Tutorial*.

UIInput and UIOutput Properties

Properties for `UIInput` and `UIOutput` objects accept the same types and are the most flexible in terms of the number of types they accept, as shown in Table 3–13.

Most of the `UIInput` and `UIOutput` properties in the `cardemo` application are of type `String`. The `zip UIInput` component is mapped to an `int` property in `CustomerBean.java` because the `zip` component is rendered with the `Number` renderer:

```
<h:input_number id="zip"  formatPattern="#####"
  valueRef="CustomerBean.zip" size="5">
  ...
</h:input_number>
```

Here is the property mapped to the `zip` component tag:

```
int zip = 0;
...
public void setZip(int zipCode) {
  zip = zipCode;
}
public int getZip() {
  return zip;
}
```

The components represented by the `input_text`, `output_text`, `input_hidden`, and `input_secret` tags can also be bound to the `Date`, `Number` and custom types in addition to `java.lang.String` when a `Converter` is applied to the component. See Performing Data Conversions (page 92) for more information.

UIPanel Properties

Only `UIPanel` components rendered with a `Data` renderer can be mapped to a model object. These `UIPanel` components must be mapped to a JavaBean component of type `array`, `java.util.Collection`, `java.util.Iterator`, or `java.util.Map`. Here is a bean that maps to the `panel_data` component from Using the panel_list Tag (page 67):

```
public class ListBean extends java.util.ArrayList{
  public ListBean() {
     add(new CustomerBean("123456", "Sun Microsystems, Inc.",
        "SUNW", 2345.60));
     add(new CustomerBean("789101", "ABC Company, Inc.",
        "ABC", 458.21));
  }
}
```

UISelectBoolean Properties

Properties that hold this component's data must be of `boolean` type. Here is the property for the `sunRoof` `UISelectBoolean` component:

```
protected boolean sunRoof = false;
   ...
public void setSunRoof(boolean roof) {
   sunRoof = roof;
}
public boolean getSunRoof() {
   return sunRoof;
}
```

UISelectMany Properties

Since a `UISelectMany` component allows a user to select one or more items from a list of items, this component must map to a model object property of type `java.util.Collection` or `array`. This model object property represents the set of currently selected items from the list of available items.

Here is the model object property that maps to the valueRef of the selectmany_checkboxlist from Using the selectmany_checkboxlist Tag (page 69):

```
protected ArrayList currentOptions = null;

public Object[] getCurrentOptions() {
  return currentOptions.toArray();
}
public void setCurrentOptions(Object []newCurrentOptions) {
  int len = 0;
  if (null == newCurrentOptions ||
    (len = newCurrentOptions.length) == 0) {
      return;
  }
  currentOptions.clear();
  currentOptions = new ArrayList(len);
  for (int i = 0; i < len; i++) {
    currentOptions.add(newCurrentOptions[i]);
  }
}
```

Note that the setCurrentOptions(Object) method must clear the Collection and rebuild it with the new set of values that the user selected.

As explained in The UISelectMany Component (page 69), the UISelectItem and UISelectItems components are used to represent all the values in a UISelectMany component. See UISelectItem Properties (page 80) and UISelectItems Properties (page 80) for information on how to write the model object properties for the UISelectItem and UISelectItems components.

UISelectOne Properties

The UISelectOne properties accept the same types as UIInput and UIOutput properties. This is because a UISelectOne component represents the single selected item from a set of items. This item could be a String, int, long, or double. Here is the property corresponding to the engineOption UISelectOne component from more.jsp:

```
protected Object currentEngineOption = engines[0];
...
public void setCurrentEngineOption(Object eng) {
  currentEngineOption = eng;
}
```

```
public Object getCurrentEngineOption() {
  return currentEngineOption;
}
```

Note that `currentEngineOption` is one of the objects in an array of objects, representing the list of items in the `UISelectOne` component.

As explained in The UISelectOne Component (page 71), the `UISelectItem` and `UISelectItems` components are used to represent all the values in a `UISelectOne` component. See UISelectItem Properties (page 80) and UISelectItems Properties (page 80) for information on how to write the model object properties for the `UISelectItem` and `UISelectItems` components.

UISelectItem Properties

A `UISelectItem` component represents one value in a set of values in a `UISelectMany` or `UISelectOne` component. A `UISelectItem` property must be mapped to property of type `SelectItem`. A `SelectItem` object is composed of: an `Object` representing the value, and two `Strings` representing the label and description of the `SelectItem`.

Here is an example model object property for a `SelectItem` component:

```
SelectItem itemOne = null;

SelectItem getItemOne(){
  return SelectItem(String value, String label, String
     description);
}

void setItemOne(SelectItem item) {
  itemOne = item;
}
```

UISelectItems Properties

The `UISelectItems` properties are the most difficult to write and require the most code. The `UISelectItems` components are used as children of `UISelect-Many` and `UISelectOne` components. Each `UISelectItems` component is composed of a set of `SelectItem` instances. In your model object, you must define a set of `SelectItem` objects, set their values, and populate the `UISelectItems` object with the `SelectItem` objects. The following code snippet from `Curren-`

tOptionServer shows how to create the engineOption UISelectItems property.

```
import javax.faces.component.SelectItem;
...
protected ArrayList engineOption;
...
public CurrentOptionServer() {
  protected String engines[] = {
     "V4", "V6", "V8"
  };
  engineOption = new ArrayList(engines.length);
  ...
  for (i = 0; i < engines.length; i++) {
     engineOption.add(new SelectItem(engines[i],
                 engines[i], engines[i]));
  }
}
...
public void setEngineOption(Collection eng) {
  engineOption = new ArrayList(eng);
}
public Collection getEngineOption() {
  return engineOption;
}
```

The code first initializes engineOption as an ArrayList. The for loop creates a set of SelectItem objects with values, labels and descriptions for each of the engine types. Finally, the code includes the obligatory setEngineOption and getEngineOption accessor methods.

Performing Validation

JavaServer Faces technology provides a set of standard classes and associated tags that page authors and application developers can use to validate a component's data. Table 3–14 lists all of the standard validator classes and the tags that allow you to use the validators from the page.

Table 3–14 The Validator Classes

Validator Class	Tag	Function
`DoubleRangeValidator`	`validate_doublerange`	Checks if the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point.
`LengthValidator`	`validate_length`	Checks if the length of a component's local value is within a certain range. The value must be a `java.lang.String`.
`LongRangeValidator`	`validate_longrange`	Checks if the local value of a component is within a certain range. The value must be anything that can be converted to a long.
`RequiredValidator`	`validate_required`	Checks if the local value of a component is not null. In addition, if the local value is a `String`, ensures that it is not empty.
`StringRangeValidator`	`validate_stringrange`	Checks if the local value of a component is within a certain range. The value must be a `java.lang.String`.

All of these validator classes implement the `Validator` interface. Component writers and application developers can also implement this interface to define their own set of constraints for a component's value.

This section shows you how to use the standard `Validator` implementations, how to write your own custom validator by implementing the `Validator` interface, and how to display error messages resulting from validation failures.

Displaying Validation Error Messages

A page author can output error messages resulting from both standard and custom validation failures using the `output_errors` tag. Here is an example of an `output_errors` tag:

```
<h:output_errors for="ccno" />
```

The `output_errors` tag causes validation error messages to be displayed wherever the tag is located on the page. The `for` attribute of the tag must match the id of the component whose data requires validation checking. This means that you must provide an ID for the component by specifying a value for the component tag's `id` attribute. If the `for` attribute is specified, the errors resulting from all failed validations on the page will display wherever the tag is located on the page. The next two sections show examples of using the `output_errors` tag with the validation tags.

Using the Standard Validators

When using the standard `Validator` implementations, you don't need to write any code to perform validation. You simply nest the standard validator tag of your choice inside a tag that represents a component of type `UIInput` (or a subclass of `UIInput`) and provide the necessary constraints, if the tag requires it. Validation can only be performed on components whose classes extend `UIInput` since these components accept values that can be validated.

The `Customer.jsp` page of the `cardemo` application uses two of the standard validators: `StringRangeValidator` and `RequiredValidator`. This section explains how to use these validators. The other standard validators are used in a similar way.

Using the Required Validator

The `zip input_text` tag on `Customer.jsp` uses a `RequiredValidator`, which checks if the value of the component is `null` or is an empty `String`. If your component must have a non-`null` value or a `String` value at least one character in length, you should register this validator on the component. If you don't register a `RequiredValidator`, any other validators you have registered on the component will not be executed. This is because the other validators can only validate a non-null value or a String value of at least one character. Here is the zip input_text tag from Customer.jsp:

```
<h:input_text id=”zip” valueRef=”CustomerBean.zip” size=”10”>
  <f:validate_required />
  <cd:format_validator
     formatPatterns=”99999|99999-9999|### ###” />
</h:input_text>
<h:output_errors for=”zip” />
```

The zip component tag contains a custom validator tag besides the validate_required tag. This custom validator is discussed in section Creating a Custom Validator (page 85). In order for other validators to be processed, the validate_required tag is needed to first check if the value is null or a String value of at least one character. However, you can register the validator tags in any order; you don’t have to register the RequiredValidator first.

Because of the output_errors tag, an error will display on the page if the value is null or an empty String. When the user enters a value in response to seeing the error message, the other validators can check the validity of the value.

Using the StringRangeValidator

The middleInitial component on the Customer.jsp page uses a StringRangeValidator, which checks if the user only enters an alphabetic character in the middleInitial component. Here is the middleInitial input_text tag from Customer.jsp:

```
<h:input_text id="middleInitial" size="1"
  maxLength="1" valueRef=”CustomerBean.middleInitial” >
  <f:validate_stringrange minimum="A" maximum="z"/>
</h:input_text>
<h:output_errors  clientId="middleInitial"/>
```

The middleInitial tag uses the size attribute and the maxLength attribute. These attributes restrict the input to one character.

The validator tag uses a StringRangeValidator whose attributes restrict the value entered to a single alphabetic character from the range A to Z, ignoring case.

Creating a Custom Validator

If the standard validators don't perform the validation checking you need, you can easily create a custom validator for this purpose. To create and use a custom validator, you need to:

1. Implement the `Validator` interface
2. Register the error messages
3. Register the `Validator` class
4. Create a custom tag or use the validator tag

The cardemo application uses a general-purpose custom validator that validates input data against a format pattern that is specified in the custom validator tag. This validator is used with the Credit Card Number field and the Zip code field. Here is the custom validator tag used with the Zip code field:

```
<cd:format_validator
  formatPatterns="99999|99999-9999|### ###" />
```

According to this validator, the data entered in the Zip code field must be either:

- A 5-digit number
- A 9-digit number, with a hyphen between the 5th and 6th digits
- A 6-character string, consisting of numbers or letters, with a space between the 3rd and 4th character

The rest of this section describe how this validator is implemented, how it works, and how to use it in a page.

Implement the Validator Interface

All custom validators must implement the `Validator` interface. This implementation must contain a constructor, a set of accessor methods for any attributes on the tag, and a `validate` method, which overrides the `validate` method of the `Validator` interface.

The `FormatValidator` class implements `Validator` and validates the data on the Credit Card Number field and the Zip code field. This class defines accessor methods for setting the attribute `formatPatterns`, which specifies the acceptable format patterns for input into the fields.

In addition to the constructor and the accessor methods, the class overrides `Validator.validate` and provides a method called `getMessageResources`, which gets the custom error messages to be displayed when the `String` is invalid.

All custom `Validator` implementations must override the `validate` method, which takes the `FacesContext` and the component whose data needs to be validated. This method performs the actual validation of the data. Here is the validate method from `FormatValidator`:

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

  if ((context == null) || (component == null)) {
    throw new NullPointerException();
  }
  if (!(component instanceof UIOutput)) {
    return;
  }
  if ( formatPatternsList == null ) {
    component.setValid(true);
    return;
  }
  String value =
    (((UIOutput)component).getValue()).toString();
  Iterator patternIt = formatPatternsList.iterator();
  while (patternIt.hasNext()) {
    valid = isFormatValid(((String)patternIt.next()), value);
    if (valid) {
      break;
    }
  }
  if ( valid ) {
    component.setValid(true);
  } else {
    component.setValid(false);
    Message errMsg =
      getMessageResources().getMessage(context,
        FORMAT_INVALID_MESSAGE_ID,
        (new Object[] {formatPatterns}));
    context.addMessage(component, errMsg);
  }
}
```

This method gets the local value of the component and converts it to a String. It then iterates over the `formatPatternsList` list, which is the list of acceptable patterns as specified in the `formatPatterns` attribute of the validator tag. While iterating over the list, this method checks the pattern of the local value against the patterns in the list. If the value's pattern matches one of the acceptable patterns, this method stops iterating over the list and marks the components value as valid by calling the component's setValid method with the value `true`. If the pattern of the local value does not match any pattern in the list, this method: marks the component's local value invalid by calling `component.setValid(false)`, generates an error message, and queues the error message to the `FacesContext` so that the message is displayed on the page during the Render Response phase.

The `FormatValidator` class also provides the `getMessageResources` method, which returns the error message to display when the data is invalid:

```
public synchronized MessageResources getMessageResources() {
  MessageResources carResources = null;
  ApplicationFactory aFactory = (ApplicationFactory)
    FactoryFinder.getFactory(
    FactoryFinder.APPLICATION_FACTORY);
  Application application =
    aFactory.getApplication();
  carResources =
    application.getMessageResources("carDemoResources");
  return (carResources);
}
```

This method first gets an `ApplicationFactory`, which returns `Application` instances. The Application instance supports the `getMessageResources(String)` method, which returns the `MessageResources` instance identified by `carResources`. This `MessageResources` instance is registered in the application configuration file. This is explained in the next section.

Register the Error Messages

If you create custom error messages, you need to make them available at application startup time. You do this by registering them with the application configuration file, faces-config.xml.

Note: This technique for registering messages is not utilized in the version of cardemo shipped with this release. The cardemo application will be updated to use this technique in future releases.

Here is the part of the file that registers the error messages:

```
<message-resources>
  <message-resources-id>
    carDemoResources
  </message-resources-id>
  <message>
    <message-id>cardemo.Format_Invalid</message-id>
    <summary xml:lang="en">
      Input must match one of the following patterns
      {0}
    </summary>
    <summary xml:lang="de">
      Eingang muß eins der folgenden Muster
      zusammenbringen {0}
    </summary>
    <summary xml:lang="es">
      La entrada debe emparejar uno de los
      patrones siguientes {0}
    </summary>
    <summary lang="fr">
      L'entrée doit assortir un des modèles
      suivants {0}
    </summary>
  </message>
</message-resources>
```

The message-resources element represents a set of localizable messages, which are all related to a unique message-resources-id. This message-resources-id is the identifier under which the MessageResources class must be registered. It corresponds to a static message ID in the FormatValidator class:

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

The message element can contain any number of summary elements, each of which defines the localized messages. The lang attribute specifies the language code.

This is all it takes to register message resources. Prior to this release, you had to write an implementation of the MessageResources class, create separate XML files for each locale, and add code to a ServletContextListener implementation. Now, all you need are a few simple lines in the faces-config.xml file to register message resources.

Register the Custom Validator

Just as the message resources need to be made available at application startup time, so does the custom validator. You register the custom validator in the faces-config.xml file with the validator XML tag:

```
<validator>
  <description>FormatValidator Description</description>
  <validator-id>FormatValidator</validator-id>
  <validator-class>cardemo.FormatValidator</validator-class>
  <attribute>
    <description>
      List of format patterns separated by '|'
    </description>
    <attribute-name>formatPatterns</attribute-name>
    <attribute-class>java.lang.String</attribute-class>
  </attribute>
</validator>
```

The validator-id and validator-class are required subelements. The validator-id represents the identifier under which the Validator class should be registered. This ID is used by the tag class corresponding to the custom validator tag.

The validator-class element represents the fully-qualified class name of the Validator class.

The attribute element identifies an attribute associated with the Validator. It has required attribute-name and attribute-class subelements. The attribute-name element refers to the name of the attribute as it appears in the validator tag. The attribute-class element identifies the Java type of the value associated with the attribute.

Create a Custom Tag or Use the validator Tag

There are two ways to register a `Validator` instance on a component from the page:

- Specify which validator class to use with the `validator` tag. The `Validator` implementation defines its own properties
- Create a custom tag that provides attributes for configuring the properties of the validator from the page

If you want to configure the attributes in the `Validator` implementation rather than from the page, the page author only needs to nest a `f:validator` tag inside the tag of the component whose data needs to be validated and set the `validator` tag's type attribute to the name of the `Validator` implementation:

```
<h:input_text id="zip" valueRef="CustomerBean.zip"
      size="10" ... >
  <f:validator type=”cardemo.FormatValidator” />
  ...
</h:input_text>
```

If you want to use a custom tag, you need to: write a tag handler to create and register the `Validator` instance on the component, write a TLD to define the tag and its attributes, and add the custom tag to the page.

Writing the Tag Handler

The tag handler associated with a custom validator tag must extend the `ValidatorTag` class. This class is the base class for all custom tag handlers that create `Validator` instances and register them on a UI component. The `FormatValidatorTag` is the class that registers the `FomatValidator` instance.

The `CreditCardValidator` tag handler class:

- Sets the ID of the Validator by calling `super.setId("FormatValidator")`.
- Provides a set of accessor methods for each attribute defined on the tag.
- Implements `createValidator` method of the `ValidatorTag` class. This method creates an instance of the `Validator` and sets the range of values accepted by the validator.

Here is the `createValidator` method from `FormatValidator`:

```
protected Validator createValidator() throws JspException {
  FormatValidator result = null;
  result = (FormatValidator) super.createValidator();
  Assert.assert_it(null != result);
  result.setFormatPatterns(formatPatterns);
  return result;
}
```

This method first calls `super.createValidator` to get a new `Validator` and casts it to `FormatValidator`.

Next, the tag handler sets the `Validator` instance's attribute values to those supplied as tag attributes in the page. The handler gets the attribute values from the page via the accessor methods that correspond to the attributes.

Writing the Tag Library Descriptor

To define a tag, you need to declare it in a tag library descriptor (TLD), which is an XML document that describes a tag library. A TLD contains information about a library and each tag contained in the library.

The custom validator tag for the Credit Card Number and Zip Code fields is defined in the `cardemo.tld`, located in `../cardemo/web/WEB-INF` directory of your download bundle. It contains only one tag definition, for `format_validator`:

```
<tag>
  <name>format_validator</name>
  <tag-class>cardemo.FormatValidatorTag</tag-class>
  <attribute>
     <name>formatPatterns</name>
     <required>true</required>
     <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
```

The `name` element defines the name of the tag as it must be used in the page. The `tag-class` element defines the tag handler class. The attribute elements define each of the tag’s attributes. For more information on defining tags in a TLD, please consult the Defining Tags section of *The Java Web Services Tutorial*.

Adding the Custom Tag to the Page

To use the custom validator in the JSP page, you need to declare the custom tag library that defines the custom tag corresponding to the custom component.

To declare the custom tag library, include a `taglib` directive at the top of each page that will contain the custom validator tags included in the tag library. Here is the `taglib` directive that declares the cardemo tag library:

```
<%@ taglib uri="/WEB-INF/cardemo.tld" prefix="cd" %>
```

The `uri` attribute value uniquely identifies the tag library. The `prefix` attribute value is used to distinguish tags belonging to the tag library. Here is the `format_validator` tag from the zip tag on `Customer.jsp`:

```
<cd:format_validator
     formatPatterns="99999|99999-9999|## ###" />
```

To register this validator on the zip component (corresponding to the Zip Code-field) you need to nest the format_validator tag within the zip component tag:

```
<h:input_text id="zip" valueRef="CustomerBean.zip" size="10" >
  ...
  <cd:format_validator
     formatPatterns=”99999|99999-9999|### ###" />
</h:input_text>
<h:output_errors for="zip" />
```

The output_errors tag following the zip input_text tag will cause the error messages to display next to the component on the page. The for refers to the component whose value is being validated.

This way, a page author can use the same custom validator for any similar component by simply nesting the custom validator tag within the component tag.

Performing Data Conversions

A typical Web application must deal with two different viewpoints of the underlying data being manipulated by the user interface:

- The model view, in which data is represented as native Java types, such as java.util.Date or java.util.Number.
- The presentation view, in which data is represented in a manner that can be read or modified by the user. For example, a java.util.Date might be represented as a text string in the format mm/dd/yy or as a set of three text strings.

The JavaServer Faces implementation automatically converts component data between these two views through the component’s renderer. For example, a UIInput component is automatically converted to a Number when it is rendered with the Number renderer. Similarly, a UIInput component that is rendered with the Date renderer is automatically converted to a Date.

The page author selects the component/renderer combination by choosing the appropriate tag: input_number for a UIInput/Number combination and input_date for a UIInput/Date combination. It is the application developer’s responsibility to ensure that the model object property associated with the component is of the same type as that generated by the renderer.

Sometimes you might want to convert a component’s data to a type not supported by the component’s renderer, or you might want to convert the format of the data. To facilitate this, JavaServer Faces technology allows you to register a Converter implementation on certain component/renderer combinations. These combinations are: UIInput/Text, UIInput/Secret, UIInput/Hidden, and UIOutput/Text.

Note: In a future release, the mechanism of using component/renderer combinations to perform conversions might be removed. Instead, the page author would register a converter on a component associated with an input_text, input_secret, input_hidden, or output_text tag to perform conversions.

The Converter converts the data between the two views. You can either use the standard converters supplied with the JavaServer Faces implementation or create your own custom Converter. This section describes how to use the standard Converter implementations and explains an example of a custom Converter.

Using the Standard Converters

The JavaServer Faces implementation provides a set of Converter implementations that you can use to convert your component data to a type not supported by its renderer. The page author can apply a Converter to a component's value by setting the component tag's converter attribute to the identifier of the Converter. In addition, the page author can customize the behavior of the Converter with an attribute tag, which specifies the format of the converted value. The following tag is an example of applying a Number converter to a component and specifying the format of the Number:

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

As shown in the tag above, the salePrice component's value is converted to a Number with a currency format. Table 3–15 lists all of the standard Converter identifiers, the attributes you can use to customize the behavior of the converter, and the acceptable values for the format of the data.

Table 3–15 Standard Converter Implementations

Converter Identifier	Configuration Attributes	Pattern Defined by	Valid Values for Attributes
`Boolean`	`none`		
`Date`	`dateStyle`	`java.text.DateFormat`	`short, medium, long, full.` Default: `short`
	`timezone`	`java.util.TimeZone`	See `java.util.Time-Zone`
`DateFormat`	`formatPattern`	`java.text.DateFormat`	See the `Formatting` lesson in *The Java Tutorial*
	`timezone`	`java.util.TimeZone`	See `java.util.Time-Zone`
`DateTime`	`dateStyle`	`java.text.DateFormat`	`short, medium, long, full` Default: `short`
	`timeStyle`	`java.text.DateFormat`	`short, medium, long, full.` Default: `short`
	`timezone`	`java.util.TimeZone`	See `java.util.Time-Zone`
`Number`	`numberStyle`	`java.text.NumberFor-mat`	`currency, inte-ger, number, percent` Default: `integer`
`NumberFor-mat`	`formatPattern`	`java.text.NumberFor-mat`	See the Formatting les-son in *The Java Tuto-rial*.

Table 3–15 Standard Converter Implementations (Continued)

Converter Identifier	Configuration Attributes	Pattern Defined by	Valid Values for Attributes
`Time`	`timeStyle`	`java.text.DateFormat`	`short, medium, long, full.` Default: `short`
	`timezone`	`java.util.TimeZone`	See `java.util.TimeZone`

Creating and Using a Custom Converter

If the standard `Converter` implementations don't perform the kind of data conversion you need to perform, you can easily create a custom `Converter` implementation for this purpose. To create and use a custom `Converter`, you need to perform these steps:

1. Implement the `Converter` interface
2. Register the `Converter` with application
3. Use the `Converter` in the page

The `cardemo` application uses a custom `Converter`, called `CreditCardConverter`, to convert the data entered in the Credit Card Number field. It strips blanks and dashes from the text string and formats the text string so that a blank space separates every four characters. This section explains how this converter works.

Implement the Converter Interface

All custom converters must implement the `Converter` interface. This implementation—at a minimum—must define how to convert data both ways between the two views of the data.

To define how the data is converted from the presentation view to the model view, the `Converter` implementation must implement the `getAsObject(Faces-`

Context, UIComponent, String) method from the Converter interface. Here is the implementation of this method from CreditCardConverter:

```
public Object getAsObject(FacesContext context,
     UIComponent component, String newValue)
        throws ConverterException {
  String convertedValue = null;
  if ( newValue == null ) {
     return newValue;
  }
  convertedValue = newValue.trim();
  if ( ((convertedValue.indexOf("-")) != -1) ||
     ((convertedValue.indexOf(" ")) != -1)) {
     char[] input = convertedValue.toCharArray();
     StringBuffer buffer = new StringBuffer(50);
     for ( int i = 0; i < input.length; ++i ) {
        if ( input[i] == '-' || input[i] == ' '  ) {
           continue;
        } else {
           buffer.append(input[i]);
        }
     }
     convertedValue = buffer.toString();
  }
  return convertedValue;
}
```

During the Apply Request Values phase, when the components' decode methods are processed, the JavaServer Faces implementation looks up the component's local value in the request and calls the getAsObject method. When calling this method, the JavaServer Faces implementation passes in the current FacesContext, the component whose data needs conversion, and the local value as a String. The method then writes the local value to a character array, trims the dashes and blanks, adds the rest of the characters to a String, and returns the String.

To define how the data is converted from the model view to the presentation view, the Converter implementation must implement the getAsString(FacesContext, UIComponent, Object) method from the Converter interface. Here is the implementation of this method from CreditCardConverter:

```
public String getAsString(FacesContext context,
  UIComponent component,Object value)
     throws ConverterException {
  String inputVal = null;
  if ( value == null ) {
         return null;
      }
      try {
         inputVal = (String)value;
      } catch (ClassCastException ce) {
         throw new ConverterException(Util.getExceptionMessage(
            Util.CONVERSION_ERROR_MESSAGE_ID));
         }
         char[] input = inputVal.toCharArray();
         StringBuffer buffer = new StringBuffer(50);
         for ( int i = 0; i < input.length; ++i ) {
            if ( (i % 4) == 0 && i != 0) {
               if (input[i] != ' ' || input[i] != '-'){
                  buffer.append(" ");
               } else if (input[i] == '-') {
                     buffer.append(" ");
               }
            }
            buffer.append(input[i]);
         }
         String convertedValue = buffer.toString();
         return convertedValue;
      }
```

During the Render Response phase, in which the components' `encode` methods are called, the JavaServer Faces implementation calls the `getAsString` method in order to generate the appropriate output. When the JavaServer Faces implementation calls this method, it passes in the current `FacesContext`, the `UIComponent` whose value needs to be converted, and the model object value to be converted. Since this `Converter` does a `String`-to-`String` conversion, this method can cast the model object value to a `String`. It then reads the `String` to a character array and loops through the array, adding a space after every four characters.

Register the Converter

When you create a custom `Converter`, you need to register it with the application. Here is the `converter` declaration from faces_config.xml:

```
<converter>
  <description>CreditCard Converter</description>
  <converter-id>creditcard</converter-id>
  <converter-class>
    cardemo.CreditCardConverter
  </converter-class>
</converter>
```

The `converter` element represents a `Converter` implementation. The converter element contains required converter-id and converter-class elements.

The converter-id element identifies an ID that is used by the converter attribute of a UI component tag to apply the converter to the component's data.

The converter-class element identifies the Converter implementation.

Use the Converter in the Page

To apply the data conversion performed by your `Converter` to a particular component's value, you need to set the `converter` attribute of the component's tag to the `Converter` implementation's identifier. You provided this identifier when you registered the `Converter` with the application, as explained in the previous section.

The identifier for the `CreditCardConverter` is `creditcard`. The `CreditCardConverter` is attached to the `ccno` component, as shown in this tag from the `Customer.jsp` page:

```
<h:input_text id="ccno" size="16"
  converter="creditcard" >
  ...
</h:input_text>
```

By setting the `converter` attribute of a component's tag to the identifier of a `Converter`, you cause that component's local value to be automatically converted according to the rules specified in the `Converter`.

This way, a page author can use the same custom `Converter` for any similar component by simply supplying the `Converter` implementation's identifier to the `converter` attribute of the component's tag.

Handling Events

As explained in Event and Listener Model (page 26), the JavaServer Faces event and listener model is similar to the JavaBeans event model in that it has strongly typed event classes and listener interfaces. JavaServer Faces technology supports two different kinds of component events: action events and value-changed events.

Action events occur when the user activates a component represented by `UICommand`. These components include buttons and hyperlinks. These events are represented by the `javax.faces.event.ActionEvent` class. An implementation of the `javax.faces.event.ActionListener` handles action events.

Value-changed events result in a change to the local value of a component represented by `UIInput` or one of its subclasses. One example of a value-changed event is that generated by entering a value in a text field. These events are represented by the `javax.faces.event.ValueChangedEvent` class. An implementation of the `javax.faces.event.ValueChangedListener` handles value-changed events.

Both action events and value-changed events can be processed at any stage during the request processing lifecycle. Both `ActionListener` and `ValueChangedListener` extend from the common `FacesListener` interface.

To cause your application to react to action events or value-changed events emitted by a standard component, you need to:

- Implement an event listener to handle the event
- Register the event listener on the component

When emitting events from custom components, you need to manually queue the event on the `FacesContext`. Handling Events for Custom Components (page 141) explains how to do this. The `UIInput` and `UICommand` components automatically queue events on the `FacesContext`.

The rest of this section explains how to implement a `ValueChangedListener` and an `ActionListener` and how to register the listeners on components.

Implementing an Event Listener

For each kind of event generated by components in your application, you need to implement a corresponding listener interface. Listeners that handle the action events in an application must implement `javax.faces.event.ActionListener`. Similarly, listeners that handle the value-changed events must implement `javax.faces.event.ValueChangedListener`. The cardemo application includes implementations of both of these listeners.

Note: You should not create an `ActionListener` to handle an event that results in navigating to a page. You should write an `Action` class to handle events associated with navigation. SeeNavigating Between Pages (page 105) for more information. `ActionListeners` should only be used to handle UI changes, such as tree expansion.

By virtue of extending from `FacesListener`, both listener implementations must implement the `getPhaseId` method. This method returns an identifier from `javax.event.PhaseId` that refers to a phase in the request processing lifecycle. The listener must not process the event until after this phase has passed. For example, a listener implementation that updates a component's model object value in response to a value-changed event should return a `PhaseId` of `PhaseId.PROCESS_VALIDATIONS` so that the local values pass validation checks before the model object is updated. The phases during which events can be handled are Apply Request Events, Process Validations, and Update Model Values. If your listener implementation returns a `PhaseID` of `PhaseId.ANY_PHASE` then the listener will process events during the Apply Request Values phase if possible.

Implementing a Value-Changed Listener

In addition to the `getPhaseId` method, a `ValueChangedListener` implementation must include a `processValueChanged(ValueChangedEvent)` method.

The `processValueChanged(ValueChangedEvent)` method processes the specified `ValueChangedEvent` and is invoked by the JavaServer Faces implementation when the `ValueChangedEvent` occurs. The `ValueChangedEvent` instance stores the old and the new values of the component that fired the event.

The `cardemo` application has a new feature that updates the price of the chosen car after an extra option is selected for the car. When the user selects an option, a

`ValueChangedEvent` is generated, and the `processValueChanged` method of the `PackageValueChanged` listener implementation is invoked. Here is the `processValueChanged` method from `PackageValueChanged`:

```
public void processValueChanged(ValueChangedEvent vEvent) {
  try {
    String componentId =
      vEvent.getComponent().getComponentId();
    FacesContext context = FacesContext.getCurrentInstance();
    String currentPrice;
    int cPrice = 0;
    currentPrice =
      (String)context.getModelValue(
        "CurrentOptionServer.carCurrentPrice");
    cPrice = Integer.parseInt(currentPrice);
    if ((componentId.equals("currentEngine")) ||
      (componentId.equals("currentBrake")) ||
      (componentId.equals("currentSuspension")) ||
      (componentId.equals("currentSpeaker")) ||
      (componentId.equals("currentAudio")) ||
      (componentId.equals("currentTransmission"))) {
        cPrice = cPrice -
          (this.getPriceFor((String)vEvent.getOldValue()));
        cPrice = cPrice +
          (this.getPriceFor((String)vEvent.getNewValue()));
    } else {
      Boolean optionSet = (Boolean)vEvent.getNewValue();
      cPrice =
        calculatePrice(componentId, optionSet, cPrice);
    }
    currentPrice = Integer.toString(cPrice);
    context.setModelValue(
      "CurrentOptionServer.carCurrentPrice", currentPrice);
  } catch (NumberFormatException ignored) {}
}
```

This method first gets the ID of the component that fired the event from `ValueChangeEvent`. Next, it gets the current price of the car from the `CurrentOptionServer` bean.

The if statement checks if the component that fired the event is one of the `SelectItems` components. If it is, it subtracts the old value of the selected option from the current price and adds the new value of the selected option to the current price. The `getPriceFor(String)` method contains the prices for each option.

If the component that fired the event is a SelectBoolean, the new value is retrieved from the event. The calculatePrice(String, Boolean, int) method checks if the value is true. If it is, the price returned from getPriceFor(String) for the selected option is added to the current price; otherwise it is subtracted from the current price.

Finally the method updates the current price in the CurrentOptionServer bean.

Implementing Action Listeners

In addition to the getPhaseId method, a ActionListener implementation must include a processAction(ActionEvent) method.

The processAction(ActionEvent) processes the specified ActionEvent and is invoked by the JavaServer Faces implementation when the ActionEvent occurs. The ActionEvent instance stores the value of commandName, which identifies the command or action that should be executed when the component associated with the commandName is activated.

The cardemo application has a another new feature that allows a user to select a package, which contains a set of options for their chosen car. These packages are called Custom, Deluxe, Performance, and Standard.

The user selects a package by clicking on one of the buttons representing a package. When the user clicks one of the buttons, an ActionEvent is generated, and the processAction(ActionEvent) method of the CarActionListener listener implementation is invoked. Here is a piece of the processAction(ActionEvent) method from CarActionListener:

```
public void processAction(ActionEvent event) {
  String actionCommand = event.getActionCommand();
  ResourceBundle rb =
    ResourceBundle.getBundle("cardemo/Resources",
    (FacesContext.getCurrentInstance().getLocale()));
  if (actionCommand.equals("custom")) {
    processCustom(event, rb);
  } else if (actionCommand.equals("standard")) {
    processStandard(event, rb);
  ...
  } else if (actionCommand.equals("recalculate")) {
    FacesContext context = FacesContext.getCurrentInstance();
    String currentPackage =
      (String)context.getModelValue(
      CurrentOptionServer.currentPackage");
    if (currentPackage.equals("custom")) {
            processCustom(event, rb);
        } else if (currentPackage.equals("standard")) {
            processStandard(event, rb);
        }
        ...
    }else if (actionCommand.equals("buy")) {
        FacesContext context = FacesContext.getCurrentInstance();
        context.setModelValue("CurrentOptionServer.packagePrice",
        context.getModelValue(
            "CurrentOptionServer.carCurrentPrice"));
    }
}
```

This method gets the `commandName` from the specified `ActionEvent`. Each of the `UICommand` components on `more.jsp` has its own unique `commandName`, but more than one component is allowed to use the same `commandName`. If one of the package buttons is clicked, this method calls another method to process the event according to the specified `commandName`. For example, `processStandard(ActionEvent, ResourceBundle)` sets each component's model value in `CurrentOptionServer` according to the options included in the Standard package. Since the engine options allowed in the Standard package are only V4 and V6, the `processStandard(ActionEvent, ResourceBundle)` method sets the `engineOption` property to an array containing V4 and V6.

If the `Recalculate` button is clicked, this method gets the value of `currentPackage` from the `CurrentOptionServer` bean. This value corresponds to the `commandName` associated with one of the package buttons. The method then calls the appropriate method to process the event associated with the current package.

If the `Buy` button is clicked, this method updates the `packagePrice` property of `CurrentOptionServer` with the current price.

Registering Listeners on Components

A page author can register a listener implementation on a component by nesting either a `valuechanged_listener` tag or an `action_listener` tag within the component's tag on the page.

Custom components and renderers also have the option of registering listeners themselves, rather than requiring the page author to be aware of registering listeners. See Handling Events for Custom Components (page 141) for more information.

This section explains how to register the PackageValueChanged listener and the CarActionListener implementations on components.

Registering a ValueChangedListener on a Component

A page author can register a ValueChangedListener on a UIInput component or a component that extends from UIInput by nesting a valuechanged_listener tag within the component's tag on the page. Several components on the more.jsp page have the PackageValueChanged listener registered on them. One of these components is currentEngine:

```
<h:selectone_menu  id="currentEngine"
  valueRef="CurrentOptionServer.currentEngineOption">
  <f:valuechanged_listener
    type="cardemo.PackageValueChanged" />
  <h:selectitems
    valueRef="CurrentOptionServer.engineOption"/>
</h:selectone_menu>
```

The type attribute of the valuechanged_listener tag specifies the fully-qualified class name of the ValueChangedListener implementation.

After this component tag is processed and local values have been validated, the component instance represented by this tag will automatically queue the ValueChangeEvent associated with the specified ValueChangedListener to the FacesContext. This listener processes the event after the phase specified by the getPhaseID method of the listener implementation.

Registering an ActionListener on a Component

A page author can register an ActionListener on a UICommand component by nesting an action_listener tag within the component's tag on the page. Several components on the more.jsp page have the CarActionListener listener implementation registered on them, as shown by the custom tag:

```
<h:command_button id="custom" commandName="custom"
  commandClass="package-selected"
  key="Custom" bundle="carDemoBundle">
  <f:action_listener type="cardemo.CarActionListener" />
</h:command_button>
```

The component tag must specify a `commandName` that specifies what action should be performed when the component is activated. The `ActionEvent` is constructed with the component ID and the `commandName`. More than one component in a component tree can have the same `commandName` if the same command is executed for those components.

The `type` attribute of the `action_listener` tag specifies the fully-qualified class name of the `ActionListener` implementation.

When the component associated with this tag is activated, the component's `decode` method (or its associated `Renderer`) automatically queues the `ActionEvent` associated with the specified `ActionListener` to the `FacesContext`. This listener processes the event after the phase specified by the `getPhaseID` method of the listener implementation.

Navigating Between Pages

As explained in section Navigation Model (page 27), this release of JavaServer Faces technology includes a new navigation model that eliminates the need to define navigation rules programmatically with an `ApplicationHandler`.

Now you define page navigation rules in a centralized XML file called the application configuration resource file. See Application Configuration (page 29) for more information on this file.

Any additional processing associated with navigation that you might have included in an `ApplicationHandler` you now include in an `Action` class. An `Action` object is referenced by the `UICommand` component that triggers navigation. The `Action` object returns a logical outcome based on the results of its processing. This outcome describes what happened during the processing. The Action that was invoked and the outcome that is returned are two criteria a navigation rule uses for choosing which page to navigate to.

This rest of this section explains:

- What navigation is
- How an application navigates between pages
- How to define navigation rules in the application configuration file
- How to include any processing associated with page navigation in an `Action` class
- How to reference an `Action` class from a component tag

What is Navigation?

Navigation is a set of rules for choosing a page to be displayed. The selection of the next page is determined by:

- The page that is currently displayed
- The `Action` that was invoked by a `UICommand` component's `actionRef` property
- An outcome string that was returned by the `Action` or passed from the component.

A single navigation rule defines how to navigate from one particular page to any number of other pages in an application. The JavaServer Faces implementation chooses the proper navigation rule according to what page is currently displayed.

Once the proper navigation rule is selected, the choice of which page to access next from the current page depends on the `Action` that was invoked and the outcome that was returned.

The `UICommand` component either specifies an outcome from its action property or refers to an `Action` object with its `actionRef` property. The `Action` object performs some processing and returns a particular outcome string.

The outcome can be anything the developer chooses, but Table 3–16 on page 106 lists some outcomes commonly used in Web applications.

Table 3–16 Common cutcome strings

Outcome	**What it means**
"success"	Everything worked. Go on to the next page
"error"	Something is wrong. Go on to an error page
"logon"	The user needs to log on first. Go on to the logon page.
"no results"	The search did not find anything. Go to the search page again.

Usually, the `Action` class performs some processing on the form data of the current page. For example, the `Action` class might check if the username and password entered in the form match the username and password on file. If they match, the `Action` returns the outcome "success". Otherwise, it returns the outcome “failure”. As this example demonstrates, both the Action and the outcome are necessary to determine the proper page to access.

Here is a navigation rule that could be used with the example Action class processing described in the previous paragraph:

```
<navigation-rule>
  <from-tree-id>logon.jsp</from-tree-id>
  <navigation-case>
     <from-action-ref>LogonForm.logon</from-action-ref>
     <from-outcome>success</from-outcome>
     <to-tree-id>/storefront.jsp</to-tree-id>
  </navigation-case>
  <navigation-case>
     <from-action-ref>LogonForm.logon</from-action-ref>
     <from-outcome>failure</from-outcome>
     <to-tree-id>/logon.jsp</to-tree-id>
     </navigation-case>
</navigation-rule>
```

This navigation rule defines the possible ways to navigate from logon.jsp. Each navigation-case element defines one possible navigation path from logon.jsp. The first navigation-case says that if LogonForm.logon returns an outcome of “success”, storefront.jsp will be accessed. The second navigation-case says that logon.jsp will be re-rendered if LogonForm.logon returns “failure”.

For a complete description of how to define navigation rules, see Configuring Navigation Rules in faces-config.xml (page 108).

The next section describes what happens behind the scenes when navigation occurs.

How Navigation Works

As section The Lifecycle of a JavaServer Faces Page (page 13) explains, a JavaServer Faces page is represented by a component tree, which is comprised of all of the components on a page. To load another page, the JavaServer Faces implementation accesses a component tree identifier and stores the tree in the FacesContext. The new navigation model determines how this tree is selected.

Any UICommand components in the tree are automatically registered with the default ActionListenerImpl. When one of the components is activated--such as by a button click--an ActionEvent is emitted. If the Invoke Application phase is reached, the default ActionListenerImpl handles this event.

The ActionListenerImpl retrieves an outcome--such as “success” or “failure”--from the component generating the event. The UICommand component either literally specifies an outcome with its action property or refers to a JavaBean component property of type Action with its actionRef property. The invoke method of the Action object performs some processing and returns a particular outcome string.

After receiving the outcome string, the ActionListenerImpl passes it to the default NavigationHandler. Based on the outcome, the currently displayed page, and the Action object that was invoked, the NavigationHandler selects the appropriate component tree by consulting the application configuration file (faces-config.xml).

The next section explains how to define navigation rules for your application in the faces-config.xml file.

Configuring Navigation Rules in faces-config.xml

An application’s navigation configuration consists of a set of navigation rules. Each rule is defined by the navigation-rule element in the faces-config.xml file. See Setting Up The Application Configuration File (page xvi) for information on how to set up the faces-config.xml file for use in your application.

Here are two example navigation rules:

```
<navigation-rule>
  <from-tree-id>/more.jsp</from-tree-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-tree-id>/buy.jsp</to-tree-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>out of stock</from-outcome>
    <from-action-ref>
      CarOptionServer.carBuyAction
    </from-action-ref>
    <to-tree-id>/outofstock.jsp</to-tree-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
```

```
    <navigation-case>
      <from-outcome>error</from-outcome>
      <to-tree-id>/error.jsp</to-tree-id>
    </navigation-case>
  </navigation-case>
```

The first navigation rule in this example says that the application will navigate from more.jsp to:

- buy.jsp if the item ordered is in stock.
- outofstock.jsp if the item is out of stock.

The second navigation rule says that the application will navigate from any page to error.jsp if the application encountered an error.

Each navigation-rule element corresponds to one component tree identifier, defined by the optional from-tree-id element. This means that each rule defines all the possible ways to navigate from one particular page in the application. If there is no from-tree-id element, the navigation rules defined in the navigation-rule element apply to all the pages in the application. The from-tree-id element also allows wildcard matching patterns. For example, this from-tree-id element says the navigation rule applies to all the pages in the cars directory:

```
<from-tree-id>/cars/*</from-tree-id>
```

As shown in the example navigation rule, a navigation-rule element can contain zero or more navigation-case elements. The navigation-case element defines a set of matching criteria. When these criteria are satisfied, the application will navigate to the page defined by the to-tree-id element contained in the same navigation-case element.

The navigation criteria are defined by optional from-outcome and from-action-ref elements.

The from-outcome element defines a logical outcome, such as “success”. The from-action-ref element refers to a bean property that returns an Action object. The Action object’s invoke method performs some logic to determine the outcome and returns the outcome.

The navigation-case elements are checked against the outcome and the Action parameters in this order:

- Cases specifying both a from-outcome value and a from-action-ref value. Both of these elements can be used if the Action’s invoke method returns different outcomes depending on the result of the processing it performs.

- Cases specifying only a `from-outcome` value. The `from-outcome` element must match either the outcome defined by the `action` attribute of the `UICommand` component or the outcome returned by the `Action` object referred to by the `UICommand` component.
- Cases specifying only a `from-action-ref` value. This value must match the `Action` instance returned by the `UICommand` component.

Once any of these cases are matched, the component tree defined by the `to-tree-id` element will be selected for rendering.

The section Referencing An Action From a Component (page 110) explains how to write the tag corresponding to the `UICommand` component to return an outcome.

Referencing An Action From a Component

The `command_button` and `command_hyperlink` tags have two attributes used to specify an outcome, which is matched against the `from-outcome` elements in the `faces-config.xml` file in order to select the next page to be rendered:

- `action`: This attribute defines a literal outcome value
- `actionRef`: This attribute identifies a bean property that returns an `Action`, whose `invoke` method is executed when this button is clicked. This `invoke` method returns an outcome string.

This `command_button` tag could be used with the example navigation rule from the previous section:

```
<h:command_button id="buy2" key="buy" bundle="carDemoBundle"
     commandName="buy" actionRef="CarServer.carBuyAction">
```

The `actionRef` attribute refers to `CarOptionServer.carBuyAction`, a bean property that returns an `Action` object, whose `invoke` method returns the logical outcome.

If the outcome matches an outcome defined by a `from-outcome` element in `faces-config.xml`, the component tree specified in that navigation case is selected for rendering if:

- No `from-action-ref` is also defined for that navigation case
- There is a `from-action-ref` also defined for that navigation case, and the `Action` it identifies matches the `Action` identified by the command component's `actionRef` attribute.

Suppose that the buy2 `command_button` tag used the `action` attribute instead of the `actionRef` attribute:

```
<h:command_button id="buy2" key="buy" bundle="carDemoBundle"
    commandName="buy" action="out-of-stock">
```

If this outcome matches an outcome defined by a `from-outcome` element in the `faces-config.xml` file, the component tree corresponding to this navigation case is selected for rendering, regardless of whether or not the same navigation case also contains a `from-action-ref` element.

The next section explains how to write the bean and the `Action` class.

Using an Action Object With a Navigation Rule

It's common for applications to have a choice of pages to navigate to from a given page. You usually need to have some application-specific processing that determines which page to access in a certain situation. The processing code goes into the `invoke` method of an `Action` object. Here is the `Action` bean property and the `Action` implementation used with the examples in the previous two sections:

```
import javax.faces.application.Action;
...
public class CurrentOptionServer extends Object{
...
  public Action getCarBuyAction() {
         if (carBuyAction == null) {
            carBuyAction = new CarBuyAction();
         return carBuyAction;
      }

      class CarBuyAction extends Action {
         public String invoke() {
            if (carId == 1 && currentPackageName.equals("Custom") &&
               currentPackage.getSunRoofSelected()) {
                  currentPackage.setSunRoofSelected(false);
               return "out of stock";
            } else {
               return "success"
            }
      }
   }
```

The `CarBuyAction.invoke` method checks if the first car is chosen, the Custom package is chosen and the sunroof option is selected. If this is true, the `sunroof` checkbox component value is set to false, and the method returns the outcome, "out of stock". Otherwise, the outcome, "success" is returned.

As shown in the example in section Configuring Navigation Rules in faces-config.xml (page 108), when the NavigationHandler receives the "out-of-stock" outcome, it selects the `/outofstock.jsp` component tree.

As shown in the example code in this section, it's a good idea to include your `Action` class inside the same bean class that defines the property returning an instance of the `Action`. This is because the `Action` class will often need to access the bean's data to determine what outcome to return. Section Combining Component Data and Action Objects (page 47) discusses this concept in more detail.

Performing Localization

For this release, all data and messages in the `cardemo` application have been completely localized for French, German, Latin-American Spanish, and American English.

The image map on the first page allows you to select your preferred locale. See Creating Custom UI Components (page 117) for information on how the image map custom component was created.

This section explains how to localize static and dynamic data and messages for JavaServer Faces applications. If you are not familiar with the basics of localizing Web applications, see *Internationalizing and Localizing Web Applications* in *The Java Web Services Tutorial*.

Localizing Static Data

Static data can be localized using the JSTL Internationalization tags by following these steps:

1. After you declare the `html_basic` and `jsf-core` tag libraries in your JavaServer Faces page, add a declaration for the JSTL `fmt` tag library:

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
```

2. Create a `Properties` file containing the localized messages.
3. Add an `fmt:setBundle` tag:

```
<fmt:setBundle
    basename="cardemo.Resources"
    scope="session" var="cardemoBundle"/>
```

   The `basename` attribute value refers to the `Properties` file, located in the `cardemo` package. Make sure the `basename` attribute specifies the fully qualified classname of your Resources file. This file contains the localized messages.

   The `scope` attribute indicates the scope—either application, session, or page—for which this bundle can be used.

   The `var` attribute is an alias to the `Resources` file. This alias can be used by other tags in the page in order to access the localized messages.

4. Add a `key` attribute to a component tag to access the particular localized message and add the `bundle` attribute to refer to the file containing the localized message. The `bundle` attribute must exactly match the `var` attribute in the `fmt:setBundle` tag. Here is an example from `more.jsp`:

```
<h:output_text
  key="OptionsPackages" bundle="carDemoBundle" />
```

For more information on using the JSTL Internationalization functionality, please refer to the *JavaServer Pages Standard Tag Library* topic in *The Java Web Services Tutorial*.

Localizing Dynamic Data

The `cardemo` application has some data that is set dynamically in JavaBean classes. Because of this, the beans must load this localized data themselves; the data can't be loaded from the page.

One example of dynamically-loaded data includes the data associated with a `UISelectOne` component. Another example is the car description that appears on the `more.jsp` page. This description corresponds to the car the user chose from the `Storefront.jsp` page. Since the chosen car is not known to the application prior to startup time, the localized description cannot be loaded from the page. Instead, the `CurrentOptionServer` bean must load the localized car description.

In the `CurrentOptionServer` bean, the localized car title and description is loaded with the `setCarId(int)` method, which is called when the user selects a car from `Storefront.jsp`. Here is a piece of the `setCarId(int)` method:

```
public void setCarId(int id) {
  try {
    ResourceBundle rb;
    switch (id) {
      case 1:
        // load car 1 data
        String optionsOne = "cardemo/CarOptions1";
        rb = ResourceBundle.getBundle(
          optionsOne,
          (FacesContext.getCurrentInstance().getLocale()));
        setCarImage("/200x168_Jalopy.jpg");
        break;
  ...
  this.setCarTitle((String)rb.getObject("CarTitle"));
  this.setCarDesc((String)rb.getObject("CarDesc"));
  this.setCarBasePrice((String)rb.getObject("CarBasePrice"));
  this.setCarCurrentPrice((String)rb.getObject(
    "CarCurrentPrice"));
  loadOptions();
}
```

This method loads the localized data for the chosen car from the `ResourceBundle` associated with the car by calling

ResourceBundle.getBundle, passing in the path to the resource file and the current locale, which is retrieved from the FacesContext. This method then calls the appropriate setter methods of the CurrentOptionServer, passing the locale-specific object representing the localized data associated with the given key.

The localized data for the UISelectOne components is loaded with the loadOptions method, which is called when the CurrentOptionServer is initialized and at the end of the setCarId(int) method. Here is a piece of the loadOptions method:

```
public void loadOptions() {
  ResourceBundle rb =
    ResourceBundle.getBundle("cardemo/Resources",
    (FacesContext.getCurrentInstance().getLocale()));
  brakes = new String[2];
  brakes[0] = (String)rb.getObject("Disc");
  brakes[1] = (String)rb.getObject("Drum");
  ...
  brakeOption = new ArrayList(brakes.length);
  ...
  for (i = 0; i < brakes.length; i++) {
    brakeOption.add(new SelectItem(brakes[i], brakes[i],
      brakes[i]));
}
```

Just like in setCarId(int), the loadOptions method loads the localized data from the ResourceBundle. As shown in the code snippet, the localized data for the brakes component is loaded into an array. This array is used to create a Collection of SelectItem instances.

Localizing Messages

The JavaServer Faces API provides a set of classes for associating a set of localized messages with a component. The Message class corresponds to a single message. A set of Message instances compose a MessageResources, which is analogous to a ResourceBundle. A MessageResourceFactory creates and returns MessageResources instances.

MessageResources instances will most commonly comprise a list of validation error messages. Performing Validation (page 81) includes an example of registering and using a MessageResources for validation error messages.

To make a `MessageResources` bundle available to an application, you need to register the `MessageResources` instance with the application. This is explained in Register the Error Messages (page 87).

After registering the `MessageResources`, you can access the messages from your application (as explained in Implement the Validator Interface, page 85) by:

1. Calling the `getMessageResources(String)` method, passing in the `MessageResources` identifier
2. Calling `getMessage` on the `MessageResources` instance, passing in the `FacesContext`, the message identifier, and the substitution parameters. The substitution parameters are usually used to embed the `Validator` properties' values in the message. For example, the custom validator described in Implement the Validator Interface (page 85) will substitute the format pattern for the {0} in this error message:

```
Input must match one of the following patterns {0}
```

Creating Custom UI Components

If you've read through the first two chapters of this tutorial, you've noticed that JavaServer Faces technology offers a rich set of standard, reusable UI components that enable you to quickly and easily construct UIs for Web applications. But often you need a component with some additional functionality or a completely new component, like a client-side image map. Although JavaServer Faces technology doesn't furnish these components in its implementation, its component architecture allows you to extend the standard components to enhance their functionality or create your own unique components.

In addition to extending the functionality of standard components, you might also want to change their appearance on the page or render them to a different client. Enabled by the flexible JavaServer Faces architecture, you can separate the definition of the component behavior from its rendering by delegating the rendering to a separate renderer. This way, you can define the behavior of a custom component once, but create multiple renderers, each of which defines a different way to render the component.

In addition to providing a means to easily create custom components and renderers, the JavaServer Faces design also makes it easy to reference them from the page through JSP custom tag library technology.

This chapter uses an image map custom component to explain all you need to know to create simple custom components, custom renderers, and associated custom tags, and to take care of all the other details associated with using the components and renderers in an application.

Determining if You Need a Custom Component or Renderer

The JavaServer Faces implementation already supports a rich set of components and associated renderers, which are enough for most simple applications. This section will help you decide if you need a custom component or custom renderer or if you can use a standard component and renderer.

When to Use a Custom Component

A component class defines the state and behavior of a UI component. This behavior includes: converting the value of a component to the appropriate markup, queuing events on components, performing validation, and other functionality.

Situations in which you need to create a custom component include:

- If you need to add new behavior to a standard component, such as generating an additional type of event.
- If you need to aggregate components to create a new component that has its own unique behavior. The new component must be a custom component. One example is a datechooser component consisting of three drop-down lists.
- If you need a component that is supported by an HTML client, but is not currently implemented by JavaServer Faces technology. The current release does not contain standard components for complex HTML components, like frames; however, because of the extensibility of the component architecture, you can easily create components like this.
- If you need to render to a non-HTML client, which requires extra components not supported by HTML. Eventually, the standard HTML render kit will provide support for all standard HTML components. However, if you are rendering to a different client—such as a phone—you might need to create custom components to represent the controls uniquely supported by the client. For example, the MIDP component architecture includes support for tickers and progress bars, which are not available on an HTML client. In this case, you might also need a custom renderer along with the component; or, you might just need a custom renderer.

You do not need to create a custom component if:

- You need to simply manipulate data on the component or add application-specific functionality to it. In this situation, you should create a model object for this purpose and bind it to the standard component rather than create a custom component. See Writing a Model Object Class (page 75) for more information on creating a model object.
- You need to convert a component's data to a type not supported by its renderer. See Performing Data Conversions (page 92) for more information about converting a component's data.
- You need to perform validation on the component data. Both standard validators and custom validators can be added to a component by using the validator tags from the page. See Performing Validation (page 81) for more information about validating a component's data.
- You need to register event listeners on components. The EA3 release eliminated the need to create a custom component in order to register an event listener on it. Now you can register event listeners on components with the `valuechanged_event` and `action_listener` tags. See Handling Events (page 99) for more information on using these tags.

When to Use a Custom Renderer

If you are creating a custom component, you need to ensure—among other things—that your component class performs these operations:

- Decoding: converting the incoming request parameters to the local value of the component.
- Encoding: converting the current local value of the component into the corresponding markup that represents it in the response.

The JavaServer Faces specification supports two programming models for handling encoding and decoding:

- Direct implementation: The component class itself implements the decoding and encoding.
- Delegated implementation: The component class delegates the implementation of encoding and decoding to a separate renderer

By delegating the operations to the renderer, you have the option of associating your custom component with different renderers so that you can represent the component in different ways on the page. If you don't plan to render a particular component in different ways, it's simpler to let the component class handle the rendering.

If you aren't sure if you will need the flexibility offered by separate renderers, but want to use the simpler direct implementation approach, you can actually use both models. Your component class can include some default rendering code, but it can delegate rendering to a renderer if there is one.

Component, Renderer, and Tag Combinations

When you decide to create a custom component, you will usually create a custom renderer to go with it. You will also need a custom tag to associate the component with the renderer and to reference the component from the page.

In rare situations, however, you might use a custom renderer with a standard component rather than a custom component. Or, you might use a custom tag without a renderer or a component. This section gives examples of these situations and provides a summary of what's required for a custom component, renderer, and tag.

One example of using a custom renderer without a custom component is when you want to add some client-side validation on a standard component. You would implement the validation code with a client-side scripting language, such as JavaScript. You render the JavaScript with the custom renderer. In this situation, you will need a custom tag to go with the renderer so that its tag handler can register the renderer on the standard component.

Both custom components and custom renderers need custom tags associated with them. However, you can have a custom tag without a custom renderer or custom component. One example is when you need to create a custom validator that requires extra attributes on the validator tag. In this case, the custom tag corresponds to a custom validator, not to a custom component or custom renderer. In any case, you still need to associate the custom tag with a server-side object.

The following table summarizes what you must or can associate with a custom component, custom renderer, or custom tag.

Table 4–1 Requirements for Custom Components, Custom Renderers, and Custom Tags

	Must have	**Can have**
custom component	custom tag	custom renderer
custom renderer	custom tag	custom component or standard component
custom JavaServer Faces tag	some server-side object, like a component, a custom renderer, or custom validator	custom component or standard component associated with a custom renderer

Understanding the Image Map Example

The `cardemo` application now includes a custom image map component on the `ImageMap.jsp` page. This image map displays a map of the world. When the user clicks on one of a particular set of regions in the map, the application sets the locale in the `FacesContext` to the language spoken in the selected region. The hot spots of the map are: the United States, Spanish-speaking Central and South America, France, and Germany.

Why Use JavaServer Faces Technology to Implement an Image Map?

JavaServer Faces technology is an ideal framework to use for implementing this kind of image map because it can perform the work that must be done on the server without requiring you to create a server-side image map.

In general, client-side image maps are preferred over server-side image maps for a few reasons. One reason is that the client-side image map allows the browser to provide immediate feedback when a user positions her mouse over a hot spot. Another reason is that client-side image maps perform better because they don't require round-trips to the server. However, in some situations, your image map might need to access the server to retrieve some data or to change the appearance of non-form controls, which a client-side image map cannot do.

The image map custom component—because it uses JavaServer Faces technology—has the best of both style of image maps: It can handle the parts of the application that need to be performed on the server, while allowing the other parts of the application to be performed on the client side.

Understanding the Rendered HTML

Here is an abbreviated version of the form part of the HTML page that the application needs to render:

```
<form METHOD="post" ACTION="/cardemo/faces/...">
  <table> <tr> <td> Welcome to JavaServer Faces</td></tr>
     <tr><td>
        <img id="mapImage" src="world.jpg" usemap="#worldMap">
           <map name="worldMap">
              <area shape="poly"
                 coords="6,15,6,28,2,30,6,34,13,28,17,..."
                 onclick="document.forms[0].selectedArea.value=
                    'NAmericas';
                    document.forms[0].submit()"
                 onmouseover="document.forms[0].mapImage.src=
                    'world_namer.jpg';"
                 onmouseout="document.forms[0].mapImage.src=
                    'world.jpg';"
                 alt="NAmericas">
                 ...
              <input type="hidden" name="selectedArea"></map>
     </td></tr>
  </table>
</form>
```

The `img` tag associates an image (`world.jpg`) with an image map, referenced in the `usemap` attribute value.

The `map` tag specifies the image map and contains a set of `area` tags.

Each `area` tag specifies a region of the image map. The `onmouseover`, `onmouse-out`, and `onmouseclick` attributes define which JavaScript code is executed when these events occur. When the user moves her mouse over a region, the `onmouseover` function associated with the region displays the map with that region highlighted. When the user moves her mouse out of a region, the `onmou-`

seout function redisplays the original image. If the user clicks on a region, the onclick function sets the value of the input tag to the id of the selected area and submits the page.

The input tag represents a hidden control that stores the value of the currently-selected area between client/server exchanges so that the server-side component classes can retrieve the value.

The server side objects retrieve the value of selectedArea and set the locale in the FacesContext according to what region was selected.

Understanding the JSP Page

Here is an abbreviated form of the JSP page that the image map component will use to generate the HTML page shown in the previous section:

```
<f:use_faces>
  <h:form formName="imageMapForm" >
  ...
     <h:graphic_image id="mapImage" url="/world.jpg"
        usemap="#worldMap" />
     <d:map id="worldMap" currentArea="NAmericas" >
        <f:action_listener
           type="cardemo.ImageMapEventHandler" />
        <d:area id="NAmericas" valueRef="NA"
           onmouseover="/cardemo/world_namer.jpg"
           onmouseout="/cardemo/world.jpg" />
        ...
     </d:map>
     ...
  </h:form>
</f:use_faces>
```

The action_listener tag nested inside the map tag causes the ImageMapEventHandler to be registered on the component corresponding to map. This handler changes the locale according to the area selected from the image map. The way this event is handled is explained more in Handling Events for Custom Components (page 141).

Notice that the area tags do not contain any of the JavaScript, coordinate, or shape data that is displayed on the HTML page. The JavaScript is generated by the AreaRenderer class. The onmouseover and onmouseout attribute values indicate the image to be loaded when these events occur. How the JavaScript is generated is explained more in Performing Encoding (page 132).

The coordinate, shape, and alt data are obtained through the `valueRef` attribute, whose value refers to an attribute in application scope. The value of this attribute is a model object, which stores the coordinate, shape, and alt data. How these model objects are stored in the application scope is explained more in Simplifying the JSP Page (page 124).

Simplifying the JSP Page

One of the primary goals of JavaServer Faces technology is ease-of-use. This includes separating out the code from the page so that a wider range of page authors can easily contribute to the Web development process. For this reason, all JavaScript is rendered by the component classes rather than being included in the page.

Ease-of-use also includes compartmentalizing the tasks of developing a Web application. For example, rather than requiring the page author to hardcode the coordinates of the hot spots in the page, the application should allow the coordinates to be retrieved from a database or generated by one of the many image map tools available.

In a JavaServer Faces application, data such as coordinates would be retrieved via a model object from the `valueRef` attribute. However, the shape and coordinates of a hotspot should be defined together because the coordinates are interpreted differently depending on what shape the hotspot is. Since a component's `valueRef` can only be bound to one property, the `valueRef` attribute cannot refer to both the shape and the coordinates.

To solve this problem, the application encapsulates all of this information in a set of `ImageArea` objects. These objects are initialized into application scope by the Managed Bean Facility (Managed Bean Creation (page 28)). Here is part of the managed-bean declaration for the `ImageArea` bean corresponding to the South America hotspot:

```
<managed-bean>
  ...
  <managed-bean-name>SA</managed-bean-name>
  <managed-bean-class>
    components.model.ImageArea
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <managed-property>
    <property-name>shape</property-name>
    <value>poly</value>
    </managed-property>
    <managed-property>
      <property-name>alt</property-name>
    <value>SAmerica</value>
    </managed-property>
    <managed-property>
      <property-name>coords</property-name>
      <value>89,217,95,100...</value>
    </managed-property>
  </managed-bean>
```

For more information on initializing managed beans with the Managed Bean Facility, see section Creating Model Objects (page 33).

The `valueRef` attributes of the `area` tags refer to the beans in the application scope, as shown in this `area` tag from `ImageMap.jsp`:

```
<d:area id="NAmericas"
    valueRef="NA"
    onmouseover="/cardemo/world_namer.jpg"
    onmouseout="/cardemo/world.jpg" />
```

To reference the `ImageArea` model object values from the component class, you need to call `getvalueRef` from your component class. This returns the name of the attribute that stores the `ImageArea` object associated with the tag being processed. Next, you need to pass the attribute to the `getValueRef` method of the `Util` class, which is a reference implementation helper class that contains various factories for resources. This will return a `ValueBinding`, which uses the expression from the `valueRef` attribute to locate the `ImageArea` object containing the values associated with the current `UIArea` component. Here is the line from `AreaRenderer` that does all of this:

```
ImageArea ia = (ImageArea)
  ((Util.getValueBinding(
  uiArea.getValueRef())).getValue(context));
```

`ImageArea` is just a simple bean, so you can access the shape, coordinates, and alt values by calling the appropriate accessor methods. Performing Encoding (page 132) explains how to do this in the `AreaRenderer` class.

Summary of the Application Classes

The following table summarizes all of the classes needed to implement the image map component.

Table 4–2 Image Map Classes

Class	**Function**
`AreaTag`	The tag handler that implements the area custom tag
`MapTag`	The tag handler that implements the map custom tag
`UIArea`	The class that defines the `UIArea` component, corresponding to the area custom tag
`UIMap`	The class that defines the `UIMap` component, corresponding to the map custom tag
`AreaRenderer`	This `Renderer` performs the delegated rendering for the `UIArea` component
`ImageArea`	The model object that stores the shape and coordinates of the hot spots
`ImageMapEventHandler`	The listener interface for handling the action event generated by the `map` component

Steps for Creating a Custom Component

Before describing how the image map works, it helps to summarize the basic steps needed to create an application that uses custom components. You can apply the following steps while developing your own custom component example.

1. Write a tag handler class that extends `javax.faces.webapp.FacesTag`. In this class, you need:

- A `getRendererType` method, which returns the type of your custom renderer, if you are using one (explained in step 4).
- A `getComponentType` method, which returns the type of the custom component.
- An `overrideProperties` method, in which you set all of the new attributes of your component.

2. Create a tag library descriptor (TLD) that defines the custom tag.
3. Create a custom component class
4. Include the rendering code in the component class or delegate it to a renderer (explained in step 6).
5. If your component generates events, queue the event on the `FacesContext`.
6. Delegate rendering to a renderer if your component does not handle the rendering.
   a. Create a custom renderer class by extending `javax.faces.render.Renderer.`
   b. Register the renderer to a render kit.
   c. Identify the renderer type in the component tag handler.
7. Register the component
8. Create an event handler if your component generates events.
9. Declare your new TLD in your JSP page and use the tag in the page.

Creating the Component Tag Handler

If you've created your own JSP custom tags before, creating a component tag and tag handler should be easy for you.

In JavaServer Faces applications, the tag handler class associated with a component drives the Render Response phase of the JavaServer Faces lifecycle. For more information on the JavaServer Faces lifecycle, see The Lifecycle of a JavaServer Faces Page (page 13). The first thing that the tag handler does is retrieve the type of the component associated with the tag. Next, it sets the component's attributes to the values given in the page. Finally, it returns the type of the renderer (if there is one) to the JavaServer Faces implementation so that the component's encoding can be performed when the tag is processed.

The image map custom component includes two tag handlers: `AreaTag` and `MapTag`. To see how the operations on a JavaServer Faces tag handler are implemented, let's take a look at `MapTag`:

```
public class MapTag extends FacesTag {
   public String currentArea = null;
   public MapTag(){
      super();
   }
   public String getCurrentArea() {
      return currentArea;
   }
   public void setCurrentArea(String area) {
      currentArea = area;
   }
   public void overrideProperties(UIComponent component) {
      super.overrideProperties(component);
      UIMap map = (UIMap) component;
      if(map.getAttribute("currentArea") == null)
         map.setAttribute("currentArea", getCurrentArea());
      }
      public String getRendererType() { return null; }
      public UIComponent createComponent() {
         return (new UIMap());
      }
   } // end of class
```

The first thing to notice is that `MapTag` extends `FacesTag`, which supports `jsp.tagext.Tag` functionality as well as JavaServer Faces-specific functionality. `FacesTag` is the base class for all JavaServer Faces tags that correspond to a component. Tags that need to process their tag bodies should subclass `FacesBodyTag` instead.

As explained above, the first thing `MapTag` does is to retrieve the type of the component. This is done with the `getComponentType` operation,:

```
public String getComponentType() {
   return (“Map”);
}
```

Next, the tag handler sets the component's attribute values to those supplied as tag attributes in the page. The `MapTag` handler gets the attribute values from the page via JavaBean properties that correspond to the attributes. `UIMap` only has one attribute, currentArea. Here is the property used to access the value of currentArea:

```
public String currentArea = null;
...
public String getCurrentArea() {return currentArea;}
public void setCurrentArea(String area) {
   currentArea = area;
}
```

To pass the value of currentArea to the UIMap component, the tag handler implements the overrideProperties method, which calls the UIMap.setAttribute method with the name and value of currentArea attribute:

```
public void overrideProperties(UIComponent component) {
   super.overrideProperties(component);
   UIMap map = (UIMap) component;
   if(map.getAttribute("currentArea") == null)
      map.setAttribute("currentArea", getCurrentArea());
}
```

Finally, the tag handler provides a renderer type—if there is a renderer associated with the component—to the JavaServer Faces implementation. It does this with the getRendererType method:

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

Since UIMap does not have a renderer associated with it, this method returns null. In this case, the JavaServer Faces implementation will invoke the encoding methods of UIMap to perform the rendering.

Delegating Rendering to a Renderer (page 136) provides an example of returning a renderer from this method.

Defining the Custom Component Tag in a Tag Library Descriptor

To define a tag, you need to declare it in a tag library descriptor (TLD), which is an XML document that describes a tag library. A TLD contains information about a library and each tag contained in the library. TLDs are used by a Web container to validate the tags. The set of tags that are part of the HTML render kit are defined in the html_basic TLD.

The custom tags `image`, `area`, and `map`, are defined in `components.tld`, which is stored in the `components/src/components/taglib` directory of your installation. The `components.tld` defines tags for all of the custom components included in this release.

All tag definitions must be nested inside the `taglib` element in the TLD. Each tag is defined by a `tag` element. Here is the tag definition of the `map` tag:

```
<tag>
  <name>map</name>
  <tag-class>cardemo.MapTag</tag-class>
  <attribute>
    <name>id</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>currentArea</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
```

At a minimum, each tag must have a `name` (the name of the tag) and a `tag-class` (the tag handler) attribute. For more information on defining tags in a TLD, please consult the Defining Tags section of *The Java Web Services Tutorial*.

Creating Custom Component Classes

As explained in When to Use a Custom Component (page 118), a component class defines the state and behavior of a UI component. Some of the state information includes the component's type, identifier, and local value. Some of the behavior defined by the component class includes:

- Decoding (converting the request parameter to the component's local value)
- Encoding (converting the local value into the corresponding markup)
- Updating the model object value with the local value
- Processing validation on the local value
- Queueing events

The UIComponentBase class defines the default behavior of a component class. All of the classes representing the standard components extend from UIComponentBase. These classes add their own behavior definitions, as your custom component class will do.

Your custom component class needs to either extend UIComponentBase directly or extend a class representing one of the standard components. These classes are located in the javax.faces.component package and their names begin with UI.

To decide whether you need to extend directly from UIComponentBase or from one of the standard component classes, consider what behavior you want your component to have. If one of the standard component classes defines most of the functionality you need, you should extend that class rather than UIComponentBase. For example, suppose you want to create an editable menu component. It makes sense to have this component extend UISelectOne rather than UIComponentBase because you can reuse the behavior already defined in UISelectOne. The only new functionality you need to define is that which makes the menu editable.

The image map example has two component classes: UIArea and UIMap. The UIMap component class extends the standard component, UICommand. The UIArea class extends UIOutput.

This following sections explain how to extend a standard component and how to implement the behavior for a component.

Extending From a Standard Component

Both UIMap and UIArea extend from standard components. The UIMap class represents the component corresponding to the map tag:

```
<d:map id="worldMap" currentArea="NAmericas" />
```

The UIArea class represents the component corresponding to the area tag:

```
<d:area id="NAmericas" valueRef="NA"
  onmouseover="/world_namer.jpg" onmouseout="/world.jpg" />
```

The `UIMap` component has one or more `UIArea` components as children. Its behavior consists of:

- Retrieving the value of the currently-selected area.
- Rendering the `map` tag and the `input` tag
- Generating an event when the user clicks on the image map
- Queuing the event on the `FacesContext`

The `UIMap` class extends from `UICommand` because `UIMap` generates an `ActionEvent` when a user clicks on the map. Since `UICommand` components already have the ability to generate this kind of event, it makes sense to extend `UICommand` rather than redefining this functionality in a custom component extending from `UIComponentBase`.

The `UIArea` component class extends `UIOutput` because `UIArea` requires a value and `valueRef` attribute, which are already defined by `UIOutput`.

The `UIArea` component is bound to a model object that stores the shape and coordinates of the region of the image map. You'll see how all of this data is accessed through the `valueRef` expression in Performing Encoding (page 132). The behavior of the `UIArea` component consists of:

- Retrieving the shape and coordinate data from the model object
- Setting the value of the `selectedArea` tag to the `id` of this component
- Rendering the `area` tag, including the JavaScript for the `onmouseover`, `onmouseout`, and `onclick` functions

Although these tasks are actually performed by `AreaRenderer`, the `UIArea` component class must delegate the tasks to `AreaRenderer`. See Delegating Rendering to a Renderer (page 136) for more information.

The rest of these ccomponents' behavior is performed in its encoding and decoding methods. Performing Encoding (page 132) and Performing Decoding (page 135) explain how this behavior is implemented.

Performing Encoding

During the Render Response phase, the JavaServer Faces implementation processes the encoding methods of all components and their associated renderers in the tree. The encoding methods convert the current local value of the component into the corresponding markup that represents it in the response.

The UIComponentBase class defines a set of methods for rendering markup: encodeBegin, encodeChildren, encodeEnd. If the component has child components, you might need to use more than one of these methods to render the component; otherwise, all rendering should be done in encodeEnd.

The UIArea class defines the component corresponding to the area tags:

```
...
<d:area id="SAmericas" valueRef="SA"
       onmouseover="/cardemo/world_samer.gif"
       onmouseout="/cardemo/world.gif" />
...
```

The UIArea component is bound to a model object that stores the shape and coordinates of the region of the image map. You'll see how all of this data is accessed through the valueRef expression in Performing Encoding (page 132). The UIArea component delegates its rendering to a renderer, as explained in Delegating Rendering to a Renderer (page 136). Therefore, UIArea has no rendering behavior.

Since UIMap is a parent component of UIArea, the area tags must be rendered after the beginning map tag and before the ending map tag. To accomplish this, the UIMap class renders the beginning map tag in encodeBegin and the rest of the map tag in encodeEnd.

The JavaServer Faces implementation will automatically invoke the encodeEnd method of the UIArea component's renderer after it invokes UIMap's encodeBegin method and before it invokes UIMap's encodeEnd method. If a component needs to perform the rendering for its children, it does this in the encodeChildren method.

Here are the encodeBegin and encodeEnd methods of UIMap:

```
public void encodeBegin(FacesContext context) throws
IOException {
  if (context == null) {
     System.out.println("Map: context is null");
     throw new NullPointerException();
  }
  ResponseWriter writer = context.getResponseWriter();
```

```
    writer.write("<Map name=\"");
    writer.write(getComponentId());
    writer.write("\">");
  }

  public void encodeEnd(FacesContext context) throws IOException
  {
    if (context == null) {
      throw new NullPointerException();
    }
    ResponseWriter writer = context.getResponseWriter();
    writer.write(
      "<input type=\"hidden\" name=\"selectedArea\"");
    writer.write("\">");
    writer.write("</Map>");
  }
```

Notice that `encodeBegin` renders only the beginning `map` tag. The `encodeEnd` method renders the `input` tag and the ending `map` tag.

These methods first check if the `FacesContext` is null. The `FacesContext` contains all of the information associated with the current request.

You also need a `ResponseWriter`, which you get from the `FacesContext`. The `ResponseWriter` writes out the markup to the current response.

The rest of the method renders the markup to the `ResponseWriter`. This basically involves passing the HTML tags and attributes to the `ResponseWriter` as strings, retrieving the values of the component attributes, and passing these values to the `ResponseWriter`.

The `id` attribute value is retrieved with the `getComponentId` method, which returns the component's unique identifier. The other attribute values are retrieved with the `getAttribute` method, which takes the name of the attribute.

If you want your component to perform its own rendering but delegate to a `Renderer` if there is one, include the following lines in the encode method to check if there is a renderer associated with this component.

```
if (getRendererType() != null) {
  super.encodeEnd(context);
  return;
}
```

If there is a Renderer available, this method invokes the superclass' encodeEnd method, which does the work of finding the renderer. The UIMap class performs its own rendering so does not need to check for available renderers.

In some custom component classes that extend standard components, you might need to implement additional methods besides encodeEnd. For example, if you need to retrieve the component's value from the request parameters—such as to update a model object—you also have to implement the decode method.

Performing Decoding

During the Apply Request Values phase, the JavaServer Faces implementation processes the decode methods of all components in the tree. The decode method extracts a component's local value from incoming request parameters and converts the value to a type acceptable to the component class.

A custom component class needs to implement the decode method only if it must retrieve the local value, or it needs to queue events onto the FacesContext. The UIMap component must do both of the tasks. Here is the decode method of UIMap:

```
public void decode(FacesContext context) throws IOException {
  if (context == null) {
     throw new NullPointerException();
  }
  String value =
     context.getServletRequest().getParameter("selectedArea");
  if (value != null)
     setAttribute("currentArea", value);
     context.addFacesEvent(
        new ActionEvent(this, commandName));
     setValid(true);
}
```

The decode method first extracts the value of selectedArea from the request parameters. Then, it sets the value of UIMap's currentArea attribute to the value of selectedArea. The currentArea attribute value indicates the currently-selected area.

The decode method queues an action event onto the FacesContext. In the JSP page, the action_listener tag nested inside the map tag causes the ImageMapEventHandler to be registered on the map component. This event handler will handle the queued event during the Apply Request Values phase, as explained in Handling Events for Custom Components (page 141).

Finally, the decode method calls setValid(true) to confirm that the local values are valid.

Delegating Rendering to a Renderer

For the purpose of illustrating delegated rendering, the image map example includes an AreaRenderer, which performs the rendering for the UIArea component.

To delegate rendering, you need to perform these tasks:

- Create the renderer class
- Register the renderer with a render kit
- Identify the renderer type in the component's tag handler

Create the Renderer Class

When delegating rendering to a renderer, you can delegate all encoding and decoding to the renderer, or you can choose to do part of it in the component class. The UIArea component class only requires encoding.

To delegate the encoding to AreaRenderer, the AreaRenderer needs to implement an encodeEnd method.

The encoding methods in a Renderer are just like those in a UIComponent class except that they accept a UIComponent argument as well as a FacesContext argument, whereas the encodeEnd method defined by UIComponentBase only takes a FacesContext. The UIComponent argument is the component that needs to be rendered. In the case of non-delegated rendering, the component is rendering itself. In the case of delegated rendering, the renderer needs to be told what component it is rendering. So you need to pass the component to the encodeEnd method of AreaRenderer:

```
public void encodeEnd(FacesContext context,
  UIComponent component) { ... }
```

The encodeEnd method of AreaRenderer must retrieve the shape, coordinates, and alt values stored in the ImageArea model object that is bound to the UIArea component. Suppose that the `area` tag currently being rendered has a `valueRef` attribute value of "`fraA`". The following line from `encodeEnd` gets the `valueRef` value of "`fraA`" and uses it to get the value of the attribute "`fraA`" from the `FacesContext`.

```
ImageArea ia = (ImageArea)
  context.getModelValue(component.getvalueRef());
```

The attribute value is the `ImageArea` model object instance, which contains the shape, coordinates, and alt values associated with the `fraA` `UIArea` component instance.

Simplifying the JSP Page (page 124) describes how the application stores these values.

After retrieving the `ImageArea` object, you render the values for shape, coords, and alt by simply calling the associated accessor methods and passing the returned values to the `ResponseWriter`, as shown by these lines of code, which write out the shape and coordinates:

```
writer.write("<area shape=\"");
writer.write(ia.getShape());
writer.write("\"" );
writer.write(" coords=\"");
writer.write(ia.getCoords());
```

The `encodeEnd` method also renders the JavaScript for the `onmouseout`, `onmouseover`, and `onclick` attributes. The page author only needs to provide the path to the images that are to be loaded during an `onmouseover` or `onmouseout` action:

```
<d:area id="France" valueRef="fraA"
  onmouseover="/cardemo/world_france.jpg"
  onmouseout="/cardemo/world.jpg"  />
```

The `AreaRenderer` class takes care of generating the JavaScript for these actions, as shown in this code from `encodeEnd`:

```
writer.write(" onmouseover=\"");
writer.write("document.forms[0].mapImage.src='");
imagePath = (String) component.getAttribute("onmouseover");
if ('/' == imagePath.charAt(0)) {
  writer.write(imagePath);
} else {
  writer.write(contextPath + imagePath);
}
writer.write("';\"");
writer.write(" onmouseout=\"");
writer.write("document.forms[0].mapImage.src='");
imagePath = (String) component.getAttribute("onmouseout");
if ('/' == imagePath.charAt(0)) {
  writer.write(imagePath);
} else {
  writer.write(contextPath + imagePath);
}
```

The JavaScript that `AreaRenderer` generates for the `onclick` action sets the value of the hidden variable, `selectedArea`, to the value of the current area's component ID and submits the page:

```
writer.write("\"
  onclick=\"document.forms[0].selectedArea.value='");
writer.write(component.getComponentId());
writer.write("'; document.forms[0].submit()\"");
writer.write(" onmouseover=\"");
writer.write("document.forms[0].mapImage.src='");
```

By submitting the page, this code causes the JavaServer Faces lifecycle to return back to the Reconstitute Component Tree phase. This phase saves any state information—including the value of the `selectedArea` hidden variable—so that a new request component tree is constructed. This value is retrieved by the `decode` method of the `UIMap` component class. This `decode` method is called by the JavaServer Faces implementation during the Apply Request Values phase, which follows the Reconstitute Request Tree Phase.

In addition to the `encodeEnd` method, `AreaRenderer` also contains an empty constructor. This will be used to create an instance of `AreaRenderer` in order to add it to the render kit.

`AreaRenderer` also must implement the `decode` method and the other encoding methods, whether or not they are needed.

Finally, `AreaRenderer` requires an implementation of `supportsComponentType`:

```
public boolean supportsComponentType(String componentType) {
  if ( componentType == null ) {
     throw new NullPointerException();
  }
  return (componentType.equals(UIArea.TYPE));
}
```

This method returns `true` when `componentType` equals `UIArea`'s component type, indicating that `AreaRenderer` supports the `UIArea` component.

Note that `AreaRenderer` extends `BaseRenderer`, which in turn extends `Renderer`. The `BaseRenderer` class is included in the RI of JavaServer Faces technology. It contains definitions of the `Renderer` class methods so that you don't have to include them in your renderer class.

Register the Renderer with a Render Kit

For every UI component that a render kit supports, the render kit defines a set of `Renderer` objects that can render the component in different ways to the client supported by the render kit. For example, the standard `UISelectOne` component class defines a component that allows a user to select one item out of a group of items. This component can be rendered with the `Listbox` renderer, the `Menu` renderer, or the `Radio` renderer. Each renderer produces a different appearance for the component. The `Listbox` renderer renders a menu that displays all possible values. The `Menu` renderer renders a subset of all possible values. The `Radio` renderer renders a set of radio buttons.

When you create a custom renderer, you need to register it with the appropriate render kit. Since the image map application implements an HTML image map, `AreaRenderer` should be registered with the HTML render kit.

You register the renderer using the application configuration file (see Application Configuration (page 29)):

```
<render-kit>
  <renderer>
    <renderer-type>Area</renderer-type>
    <renderer-class>
      components.renderkit.AreaRenderer
    </renderer-class>
  </renderer>
</render-kit>
```

The `render-kit` element represents a `RenderKit` implementation. If no `render-kit-id` is specified, the default HTML render kit is assumed. The renderer element represents a `Renderer` implementation. By nesting the `renderer` element inside the `render-kit` element, you are registering the renderer with the `RenderKit` associated with the `render-kit` element.

The renderer-type will be used by the tag handler, as explained in the next section. The renderer-class is the fully-qualified classname of the Renderer.

Identify the Renderer Type

During the Render Response phase, the JavaServer Faces implementation calls the getRendererType method of the component's tag to determine which renderer to invoke, if there is one.

The getRendererType method of AreaTag must return the type associated with AreaRenderer. Recall that you identified this type when you registered AreaRenderer with the render kit. Here is the getRendererType method from the cardemo application's AreaTag class:

```
public String getRendererType() { return “Area”;}
```

Register the Component

After writing your component classes, you need to register them with the application using the application configuration file (see Application Configuration (page 29))

Here are the declarations that register the UIMap and UIArea components:

```
<component>
  <component-type>Area</component-type>
  <component-class>
    components.components.UIArea
  </component-class>
</component>
<component>
  <component-type>Map</component-type>
  <component-class>
    components.components.UIMap
  </component-class>
</component>
```

The component-type element indicates the name under which the component should be registered. Other objects referring to this component use this name. The component-class element indicates the fully-qualified class name of the component.

Handling Events for Custom Components

As explained in Handling Events (page 99), a standard component queues events automatically on the FacesContext. Custom components on the other hand must manually queue the event from the decode method.

Performing Decoding (page 135) explained how to write the decode method of UIMap to queue an event on the FacesContext component. This section explains how to write an event handler to handle this event and to register the event handler on the component.

The JavaServer Faces implementation calls the processing methods of any event handlers registered on components and queued on the FacesContext. The UIMap component queues an event on the FacesContext. In the JSP page, the ImageMapEventHandler was registered on map by nesting the action_listener tag within the map tag:

```
<d:map id="worldMap" currentArea=”NAmericas” >
  <f:action_listener type="cardemo.ImageMapEventHandler"/>
  ...
</d:map>
```

Since ImageMapEventHandler is registered on the map component, the JavaServer Faces implementation calls the ImageMapEventHandler’s processAction method when the user clicks on the image map:

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

When the JavaServer Faces implementation calls this method, it passes in an ActionEvent, representing the event generated by clicking on the image map.

This method first gets the `UIMap` component that generated the event by calling `event.getSource`. From this component, this method gets the `currentArea` attribute value, which is the ID of the currently-selected area. With this value, this method gets the locale corresponding to the selected area and sets the locale in the `FacesContext`. The rest of the code sets the component tree in `FacesContext` to that corresponding to `Storefront.jsp`, causing `Storefront.jsp` to load after the user clicks the image map.

It is possible to implement event-handling code in the custom component class instead of in an event handler if the component receives application events. This component class must subclass `UIComponentBase`. It must also implement the appropriate listener interface. This scenario allows an application developer to create a component that registers itself as a listener so that the page author doesn't need to register it.

Using the Custom Component in the Page

After you've created your custom component and written all the accompanying code, you are ready to use the component from the page.

To use the custom component in the JSP page, you need to declare the custom tag library that defines the custom tag corresponding to the custom component. The tag library is described in Defining the Custom Component Tag in a Tag Library Descriptor (page 129).

To declare the custom tag library, include a `taglib` directive at the top of each page that will contain the tags included in the tag library. Here is the `taglib` directive that declares the JavaServer Faces components tag library:

```
<%@ taglib uri="http://java.sun.com/jsf/demo/components"
  prefix=”d” %>
```

The `uri` attribute value uniquely identifies the tag library. The `prefix` attribute value is used to distinguish tags belonging to the tag library. For example, the `map` tag must be referenced in the page with the `d` prefix, like this:

```
<d:map ...>
```

Don't forget to also include the `taglib` directive for the standard tags included with the RI:

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

When you reference any JavaServer Faces tags—custom or standard—from within a JSP page, you must enclose all of them in the `use_faces` tag:

```
<f:use_faces>
  ... other faces tags, custom tags, and possibly mixed with
  other content
</f:use_faces>
```

All form elements must also be enclosed within the `form` tag, which is also nested within the `use_faces` tag:

```
<f:use_faces>
  <h:form formName="imageMapForm" >
    ... other faces tags, custom tags, and possibly mixed with
    other content
  </h:form>
<f:use_faces>
```

The `form` tag encloses all of the controls that display or collect data from the user. The `formName` attribute is passed to the application, where it is used to select the appropriate business logic.

Now that you've set up your page, you can add the custom tags in between the `form` tags, as shown here in the `ImageMap.jsp` page:

```
...
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jsf/demo/components"
          prefix="d" %>
<f:use_faces>
  <h:form formName="imageMapForm" >
    <table> <tr> <td>
    ...
      <tr> <TD>
      <h:graphic_image url="/world.jpg" usemap="#worldMap" />
      <d:map id="worldMap" currentArea="NAmericas" >
        <d:area id="NAmericas" valueRef="NA"
          onmouseover="/cardemo/world_namer.jpg"
```

```
                onmouseout=”/cardemo/world.jpg” />
            ...
         </d:map>
      </TD></tr></table>
      </h:form>
   </f:use_faces>
```

Conclusion

JavaServer Faces technology provides a rich, flexible architecture that makes it easy to build Web applications with server-side UI functionality.

You have seen how to use this technology to extend the functionality of standard components and create new components, to perform data conversions and validation, and to handle component events. You have also seen how to specify the rendering of components and how to use them in a Web application.

You have gained this knowledge by learning about the various examples included in the release and explained in this tutorial. You now have the means to create your own Web applications using JavaServer Faces technology.