Creating Custom UI Components

JavaServer Faces technology offers a basic set of standard, reusable UI components that enable quick and easy construction of user interfaces s for web applications.

But often an application requires a component that has additional functionality or requires a completely new component.

JavaServer Faces technology allows extension of standard components to enhance their functionality or to create custom components.

Using the Composite Components feature of Facelets, you can combine and reuse the standard components and provide extended functionality.

In some cases, however, it becomes necessary to create new components from scratch.

For example, you may want to change the appearance of a component, provide a different renderer, or even alter listener behavior.

For such cases, JavaServer Faces provides the ability to create custom components by extending the `UIComponent` class, the class that is the base class for all standard UI components.

This chapter explains how you can create simple custom components, custom renderers, custom converters, custom listeners, and associated custom tags, and take care of all the other details associated with using the components and renderers in an application.

The following topics are addressed here:

- Determining Whether You Need a Custom Component or Renderer
- Steps for Creating a Custom Component
- Creating Custom Component Classes
- Delegating Rendering to a Renderer
- Handling Events for Custom Components
- Creating the Component Tag Handler
- Defining the Custom Component Tag in a Tag Library Descriptor
- Creating a Custom Converter

- Implementing an Event Listener
- Creating a Custom Validator
- Using Custom Objects
- Binding Component Values and Instances to External Data Sources
- Binding Converters, Listeners, and Validators to Backing Bean Properties

Determining Whether You Need a Custom Component or Renderer

The JavaServer Faces implementation supports a rich set of components and associated renderers, which are suitable enough for most simple applications.

This section helps you to decide whether you can use standard components and renderers in your application or need a custom component or custom 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.

You need to create a custom component in the following situations:

- You need to add new behavior to a standard component, such as generating an additional type of event.
- 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, such as frames; however, because of the extensibility of the component architecture, you can use JavaServer Faces technology to create components like these.

- **You need to render to a non-HTML client that 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, some component architectures for wireless clients include 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 need only a custom renderer.

You do not need to create a custom component in these cases:

- **You need to aggregate components to create a new component that has its own unique behavior.**

For this case, you can use a composite component to combine existing standard components.

For more information on composite components, see Composite Components and Chapter 13, Advanced Composite Components.

- **You simply need to manipulate data on the component or add application-specific functionality to it.**

In this situation, you should create a backing bean for this purpose and bind it to the standard component rather than create a custom component.

See Backing Beans for more information on backing beans.

- You need to convert a component's data to a type not supported by its renderer.

See Using the Standard Converters for more information about converting a component's data.

- You need to perform validation on the component data.

Standard validators and custom validators can be added to a component by using the validator tags from the page.

See Using the Standard Validators and Creating a Custom Validator for more information about validating a component's data.

- **You need to register event listeners on components.**

You can either register event listeners on components using the `valueChangeListener` and `actionListener` tags, or you can point at an event-processing method on a backing bean using the component's `actionListener` or `valueChangeListener` attributes.

See Implementing an Event Listener and Writing Backing Bean Methods for more information.

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 whether you will need the flexibility offered by separate renderers but you 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 create a custom component, you can create a custom renderer to go with it.

To associate the component with the renderer and to reference the component from the page, you will also need a custom tag.

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 summarizes what's required for a custom component, renderer, and tag.

You would use a custom renderer without a custom component if you wanted 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, and then render the JavaScript with the custom renderer.

In this situation, you need a custom tag to go with the renderer so that its tag handler can register the renderer on the standard component.

Custom components as well as custom renderers need custom tags associated with them.

However, you can have a custom tag without a custom renderer or custom component.

For example, suppose that 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 and not to a custom component or custom renderer.

In any case, you still need to associate the custom tag with a server-side object.

Table 14-1 summarizes what you must or can associate with a custom component, custom renderer, or custom tag.

Table 14-1 Requirements for Custom Components, Custom Renderers, and Custom Tags

Custom Item	Must Have	Can Have
Custom component	Custom tag	Custom renderer or standard 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

Steps for Creating a Custom Component

You can apply the following steps while developing your own custom component.

1. Create a custom component class that does the following:

a. Overrides the `getFamily` method to return the component family, which is used to look up renderers that can render the component.

b. Includes the rendering code or delegates it to a renderer (explained in step 2).

c. Enables component attributes to accept expressions.

d. Queues an event on the component if the component generates events.

e. Saves and restores the component state.

2. Delegate rendering to a renderer if your component does not handle the rendering.

To do this:

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.

3. Register the component.

4. Create an event handler if your component generates events.

5. Write a tag handler class that extends `javax.faces.webapp.UIComponentELTag`.

In this class, you need a `getRendererType` method, which returns the type of your custom renderer if you are using one (explained in step 2); a `getComponentType` method, which returns the type of the custom component; and a `setProperties` method, with which you set all the new attributes of your component.

6. Create a tag library descriptor (TLD) that defines the custom tag.

The section <u>Using a Custom Component</u> discusses how to use the custom component in a JavaServer Faces page.

Creating Custom Component Classes

As explained in When to Use a Custom Component, a component class defines the state and behavior of a UI component.

The state information includes the component's type, identifier, and local value.

The behavior defined by the component class includes the following:

- Decoding (converting the request parameter to the component's local value)
- Encoding (converting the local value into the corresponding markup)
- Saving the state of the component
- Updating the bean 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 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 must 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`.

If your custom component serves the same purpose as a standard component, you should extend that standard component rather than directly extend `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 to make the menu editable.

Whether you decide to have your component extend `UIComponentBase` or a standard component, you might also want your component to implement one or more of these behavioral interfaces:

- `ActionSource`: Indicates that the component can fire an `ActionEvent`.

- **`ActionSource2`: Extends `ActionSource` and allows component properties referencing methods that handle action events to use method expressions as defined by the unified EL.**

This class was introduced in JavaServer Faces Technology 1.2.

- `EditableValueHolder`: Extends `ValueHolder` and specifies additional features for editable components, such as validation and emitting value-change events.

- `NamingContainer`: Mandates that each component rooted at this component have a unique ID.

- `StateHolder`: **Denotes that a component has state that must be saved between requests.**

- `ValueHolder`: **Indicates that the component maintains a local value as well as the option of accessing data in the model tier.**

If your component extends `UIComponentBase`, it automatically implements only `StateHolder`.

Because all components directly or indirectly extend `UIComponentBase`, they all implement `StateHolder`.

If your component extends one of the other standard components, it might also implement other behavioral interfaces in addition to `StateHolder`.

If your component extends `UICommand`, it automatically implements `ActionSource2`.

If your component extends `UIOutput` or one of the component classes that extend `UIOutput`, it automatically implements `ValueHolder`.

If your component extends `UIInput`, it automatically implements `EditableValueHolder` and `ValueHolder`.

See the JavaServer Faces API documentation to find out what the other component classes implement.

You can also make your component explicitly implement a behavioral interface that it doesn't already by virtue of extending a particular standard component.

For example, if you have a component that extends `UIInput` and you want it to fire action events, you must make it explicitly implement `ActionSource2` because a `UIInput` component doesn't automatically implement this interface.

Specifying the Component Family

If your custom component class delegates rendering, it needs to override the `getFamily` method of `UIComponent` to return the identifier of a component family, which is used to refer to a component or set of components that can be rendered by a renderer or set of renderers.

The component family is used along with the renderer type to look up renderers that can render the component:

```
public String getFamily()
{ return ("Map"); }
```

The component family identifier, `Map`, must match that defined by the `component-family` elements included in the component and renderer configurations in the application configuration resource file.

Performing Encoding

During the Render Response phase, the JavaServer Faces implementation processes the encoding methods of all components and their associated renderers in the view.

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`, and `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`.

Alternatively, you can use the `encodeALL` method, which encompasses all the methods.

Here is an example of the `encodeBegin` and `encodeEnd` methods:

```
public void encodeBegin
(FacesContext context,
UIComponent component)
throws IOException {
if ((context == null)||
(component == null))
{throw new NullPointerException();}

MapComponent map =
```

```
(MapComponent) component;
ResponseWriter writer =
context.getResponseWriter();
writer.startElement("map", map);
writer.writeAttribute
("name", map.getId(),"id");
}
public void encodeEnd
(FacesContext context)
throws IOException {
if ((context == null) ||
```

```
(component == null))
{throw new NullPointerException();}
MapComponent map =
(MapComponent) component;
ResponseWriter writer =
context.getResponseWriter();
writer.startElement("input", map);
writer.writeAttribute
("type", "hidden", null);

writer.writeAttribute
```

```
("name", getName(context,map),
"clientId");
writer.endElement("input");
writer.endElement("map");
}
```

The encoding methods accept a `UIComponent` argument and a `FacesContext` argument.

The `FacesContext` instance contains all the information associated with the current request.

The `UIComponent` argument is the component that needs to be rendered.

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 encoding method to check whether 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's `encodeEnd` method, which does the work of finding the renderer.

In some custom component classes that extend standard components, you might need to implement other methods in addition to `encodeEnd`.

For example, if you need to retrieve the component's value from the request parameters, you must also 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 uses a `Converter` class to convert the value to a type that is acceptable to the component class.

A custom component class or its renderer must implement the `decode` method only if it must retrieve the local value or if it needs to queue events.

The component queues the event by calling `queueEvent`.

Here is an example of the `decode` method:

```
public void decode
(FacesContext context,
UIComponent component) {
if ((context == null) ||
(component == null))
```

```
{throw new NullPointerException();}
MapComponent map =
 (MapComponent) component;
String key = getName(context, map);
String value = (String)
context.getExternalContext().
getRequestParameterMap().get(key);
if (value != null)
map.setCurrent(value);
}}
```

Enabling Component Properties to Accept Expressions

Nearly all the attributes of the standard JavaServer Faces tags can accept expressions, whether they are value expressions or method expressions.

It is recommended that you also enable your component attributes to accept expressions because this is what page authors expect, and it gives page authors much more flexibility when authoring their pages.

Creating the Component Tag Handler describes how a tag handler sets the component's values when processing the tag.

It does this by providing the following:

- A method for each attribute that takes either a `ValueExpression` or `MethodExpression` object depending on what kind of expression the attribute accepts.
- A `setProperties` method that stores the `ValueExpression` or `MethodExpression` object for each component property so that the component class can retrieve the expression object later.

To retrieve the expression objects that `setProperties` stored, the component class must implement a method for each property that accesses the appropriate expression object, extracts the value from it and returns the value.

If your component extends `UICommand`, the `UICommand` class already does the work of getting the `ValueExpression` and `MethodExpression` instances associated with each of the attributes that it supports.

However, if you have a custom component class that extends `UIComponentBase`, you will need to implement the methods that get the `ValueExpression` and `MethodExpression` instances associated with those attributes that are enabled to accept expressions.

For example, you could include a method that gets the `ValueExpression` instance for the `immediate` attribute:

```
public boolean isImmediate() {
if (this.immediateSet)
{ return (this.immediate); }
ValueExpression ve =
getValueExpression("immediate");
if (ve != null) {
Boolean value = (Boolean)
ve.getValue
(getFacesContext().getELContext());
return (value.booleanValue());
}else{ return (this.immediate); }}
```

The properties corresponding to the component attributes that accept method expressions must accept and return a `MethodExpression` object.

For example, if the component extended `UIComponentBase` instead of `UICommand`, it would need to provide an `action` property that returns and accepts a `MethodExpression` object:

```
public MethodExpression getAction()
{ return (this.action); }
public void
setAction(MethodExpression action)
{ this.action = action; }
```

Saving and Restoring State

Because component classes implement `StateHolder`, they must implement the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods to help the JavaServer Faces implementation save and restore the state of components across multiple requests.

To save a set of values, you must implement the `saveState(FacesContext)` method.

This method is called during the Render Response phase, during which the state of the response is saved for processing on subsequent requests.

Here is an example:

```
public Object
saveState(FacesContext context){
Object values[] = new Object[2];
values[0] =
super.saveState(context);
values[1] = current;
return (values);
}
```

This method initializes an array, which will hold the saved state.

It next saves all of the state associated with the component.

A component that implements `StateHolder` must also provide an implementation for `restoreState(FacesContext, Object)`, which restores the state of the component to that saved with the `saveState(FacesContext)` method.

The `restoreState(FacesContext, Object)` method is called during the restore view phase, during which the JavaServer Faces implementation checks whether there is any state that was saved during the last render response phase and needs to be restored in preparation for the next postback.

Here is an example of the `restoreState(FacesContext, Object)` method:

```
public void
restoreState(FacesContext context,
Object state) {
Object values[] = (Object[]) state;
super.restoreState
(context, values[0]);
current = (String) values[1];
}
```

This method takes a `FacesContext` and an `Object` instance, representing the array that is holding the state for the component.

This method sets the component's properties to the values saved in the `Object` array.

When you implement these methods in your component class, be sure to specify in the deployment descriptor where you want the state to be saved: either client or server.

If state is saved on the client, the state of the entire view is rendered to a hidden field on the page.

To specify where state is saved for a particular web application, you need to set the `javax.faces.STATE_SAVING_METHOD` context parameter to either client or server in your application's deployment descriptor.

Delegating Rendering to a Renderer

This section explains in detail the process of delegating rendering to a renderer.

To delegate rendering, you perform these tasks:

- Create the `Renderer` class.
- Identify the renderer type in the `FacesRenderer` annotation or in the component's tag handler.

Creating 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.

To perform the rendering for the component, the renderer must implement an `encodeEnd` method.

In addition to the `encodeEnd` method, the renderer contains an empty constructor.

This is used to create an instance of the renderer so that it can be added to the render kit.
Here is an example of creating a renderer class:

```
@FacesRenderer
(componentFamily=
"javax.faces.Command",
rendererType=
"com.sun.bookstore6.bookstoreRenderer")
public class ImageRenderer extends
Renderer{}
```

The `@FacesRenderer` annotation registers the renderer class with the JavaServer Faces implementation as a renderer class.

The annotation also identifies the component family as well as the renderer type.

Identifying the Renderer Type

During the render response phase, the JavaServer Faces implementation calls the `getRendererType` method of the component's tag handler to determine which renderer to invoke, if there is one.

The `getRendererType` method of the tag handler must return the type associated with the renderer.

Creating the Component Tag Handler explains more about the `getRendererType` method.

Handling Events for Custom Components

As explained in Implementing an Event Listener, events are automatically queued on standard components that fire events.

A custom component, on the other hand, must manually queue events from its `decode` method if it fires events.

In addition to the method that processes the event, you need the event class itself.

This class is very simple to write: You have it extend `ActionEvent` and provide a constructor that takes the component on which the event is queued and a method that returns the component.

Creating the Component Tag Handler

After you create your component and renderer classes, you're ready to define how a tag handler processes the tag representing the component and renderer combination.

If you've created your own JSP custom tags before, creating a component 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 Application.

The first thing that the tag handler does is to retrieve the type of the component associated with the tag.

Next, it sets the component's attributes to the values given in the page.

It then 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.

Finally, it releases resources used during the processing of the tag.

The class extends `UIComponentELTag`, which supports `javax.servlet.jsp.tagext.Tag` functionality as well as JavaServer Faces-specific functionality.

`UIComponentELTag` is the base class for all JavaServer Faces tags that correspond to a component.

Tags that need to process their tag bodies should instead subclass `UIComponentBodyELTag`.

Retrieving the Component Type

As explained earlier, the first thing the tag handler class does is to retrieve the type of the component.

It does this by using the `getComponentType` method.

Setting Component Property Values

After retrieving the type of the component, the tag handler sets the component's property values to those supplied as tag attributes values in the page.

This section assumes that your component properties are enabled to accept expressions, as explained in Enabling Component Properties to Accept Expressions.

Getting the Attribute Values

Before setting the values in the component class, the tag handler first gets the attribute values from the page by means of JavaBeans component properties that correspond to the attributes.

The following code shows the property used to access the value of the `immediate` attribute.

```
private javax.el.ValueExpression
immediate = null;
public void setImmediate
(javax.el.ValueExpression immediate)
{ this.immediate = immediate; }
```

As this code shows, the `setImmediate` method takes a `ValueExpression` object.

This means that the `immediate` attribute of the tag accepts value expressions.

Similarly, the `setActionListener` and `setAction` methods take `MethodExpression` objects, which means that these attributes accept method expressions.

The following code shows the properties used to access the values of the actionListener and the action attributes

```
private javax.el.MethodExpression
actionListener = null;
```

```
public void setActionListener
(javax.el.MethodExpression
actionListener){
 this.actionListener =
actionListener;
}
private javax.el.MethodExpression
action = null;
public void setAction
(javax.el.MethodExpression action)
{ this.action = action; }
```

Setting the Component Property Values

To pass the value of the tag attributes to the component, the tag handler implements the `setProperties` method.

The way `setProperties` passes the attribute values to the component class depends on whether the values are value expressions or method expressions.

Setting Value Expressions on Component Properties

When the attribute value is a value expression, `setProperties` first checks if it is not a literal expression.

If the expression is not a literal, `setProperties` stores the expression into a collection, from which the component class can retrieve it and resolve it at the appropriate time.

If the expression is a literal, `setProperties` performs any required type conversion and then does one of the following:

- **If the attribute is renderer-independent, meaning that it is defined by the component class, then `setProperties` calls the corresponding setter method of the component class.**

- **If the attribute is renderer-dependent, `setProperties` stores the converted value into the component's map of generic renderer attributes.**

Setting Method Expressions on Component Properties

The process of setting the properties that accept method expressions is done differently depending on the purpose of the method.

The `actionListener` attribute uses a method expression to reference a method that handles action events.

The `action` attribute uses a method expression to either specify a logical outcome or to reference a method that returns a logical outcome, which is used for navigation purposes.

To handle the method expression referenced by `actionListener`, the `setProperties` method must wrap the expression in a special action listener object called `MethodExpressionActionListener`.

This listener executes the method referenced by the expression when it receives the action event.

The `setProperties` method then adds this `MethodExpressionActionListener` object to the list of listeners to be notified when an event occurs.

The following piece of `setProperties` does all of this:

```
if (actionListener != null) {
map.addActionListener(
new MethodExpressionActionListener
(actionListener));
}
```

If your component fires value change events, your tag handler's `setProperties` method does a similar thing, except it wraps the expression in a `MethodExpressionValueChangeListener` object and adds the listener using the `addValueChangeListener` method.

In the case of the method expression referenced by the `action` attribute, the `setProperties` method uses the `setActionExpression` method of `ActionSource2` to set the corresponding property on the component.

Providing the Renderer Type

After setting the component properties, the tag handler provides a renderer type (if there is a renderer associated with the component) to the JavaServer Faces implementation.

It does this using the `getRendererType` method.

The renderer type that is returned is the name under which the renderer is registered with the application.

See Delegating Rendering to a Renderer for more information.

If your component does not have a renderer associated with it, `getRendererType` should return `null`.

In this case, the `renderer-type` element in the application configuration file should also be set to `null`.

Releasing Resources

It's recommended practice that all tag handlers implement a `release` method, which releases resources allocated during the execution of the tag handler by first calling the `UIComponentTag.release` method, then setting the resource values to null.

Defining the Custom Component Tag in a Tag Library Descriptor

To use a custom tag, you declare it in a Tag Library Descriptor (TLD).

The TLD file defines how the custom tag is used in a JavaServer Faces page.

The web container uses the TLD to validate the tag.

The set of tags that are part of the HTML render kit are defined in the HTML_BASIC TLD, available at http://download.oracle.com/javaee/6/javaserverfaces/2.1/docs/renderkitdocs/.

At a minimum, each tag must have a name and a namespace attached to it in the TLD.

The TLD file name must end with `taglib.xml`.

Here is an example TLD named `mytaglib.xml`, with name and namespace entries:

```
<facelet-taglib>
<namespace>
<tag>
<tag-name> </tag-name>
```

```
<component> </component>
</tag>
</namespace>
</facelet-taglib>
```

You can also add additional attributes and attribute types within a `tag` element for each custom component.

Each attribute element defines one of the tag attributes.

As described in Defining a Tag Attribute Type, the attribute element defines what kind of value the attribute accepts, which for JavaServer Faces tags is either a deferred value expression or a method expression.

Creating a Custom Converter

A JavaServer Faces converter class converts strings to objects and objects to strings as required.

Several standard converters are provided by JavaServer Faces for this purpose.

See for more information on these included converters.

If the standard converters included with JavaServer Faces cannot perform the data conversion that you need, you can create a custom converter to perform this specialized conversion.

All custom converters must implement the `Converter` interface.

This section explains how to implement this interface to perform a custom data conversion.

The custom converter class is created as follows:

```
@FacesConverter
("com.bookstore6.Card")
public class CredirCardConverter
implements Converter
{..............}
```

The `@FacesConverter` annotation registers the custom converter class as a converter with the name of `com.bookstore6.Card` with JavaServer Faces implementation.

Alternatively you can use the deprecated method of registering the converter with entries in the application configuration resource file as shown in the following example:

```
<converter>
<converter-id>
com.bookstore6.Card
</converter-id>
```

```
<converter-class>
com.bookstore6.CreditCardConverter
</converter-class>
</converter>
```

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

`(FacesContext, UIComponent, String)`

method from the `Converter` interface.

Here is an implementation of this method:

```
public Object
getAsObject(FacesContext context,
UIComponent component,
String newValue)
throws ConverterException {
String convertedValue = null;
if ( newValue == null )
{ return newValue; }
```

```
// Since this is only
// a String to String conversion,
// this conversion does not
// throw ConverterException.
convertedValue = newValue.trim();
if ( (convertedValue.contains("-"))
||(convertedValue.contains(" "))) {
char[] input =
convertedValue.toCharArray();
StringBuffer buffer =
new StringBuffer(input.length);
```

```
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` instance, 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 hyphens 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 an implementation of this method:

```
public String
getAsString(FacesContext context,
UIComponent component,
Object value)
throws ConverterException {
String inputVal = null;
if ( value == null )
{ return null; }
// value must be of the type
// that can be cast to a String.
try {inputVal = (String)value;}
```

```
catch (ClassCastException ce) {
FacesMessage errMsg =
MessageFactory.getMessage(
CONVERSION_ERROR_MESSAGE_ID,
 (new Object[]
{ value, inputVal }));
throw new ConverterException
 (errMsg.getSummary());
}
// insert spaces after
// every four characters for better
```

```
// readability if
// it doesn't already exist.
char[] input =
inputVal.toCharArray();
StringBuffer buffer =
new StringBuffer(input.length + 3);
for(int i = 0; i< input.length; ++i){
if ( (i % 4) == 0 && i != 0) {
if (input[i] != ' ' ||
input[i] != '-'){
buffer.append(" ");
```

```
// if there are any "-"'s
// convert them to blanks.
} 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 bean value to be converted.

Because this converter does a `String`-to-`String` conversion, this method can cast the bean value to a `String`.

If the value cannot be converted to a `String`, the method throws an exception, passing an error message from the resource bundle that is registered with the application.

Registering Custom Error Messages explains how to register custom error messages with the application.

If the value can be converted to a `String`, the method reads the `String` to a character array and loops through the array, adding a space after every four characters.

Implementing an Event Listener

The JavaServer Faces technology supports action events and value-change events for components.

Action events occur when the user activates a component that implements `ActionSource`.

These events are represented by the class `javax.faces.event.ActionEvent`.

Value-change events occur when the user changes the value of a component that implements `EditableValueHolder`.

These events are represented by the class `javax.faces.event.ValueChangeEvent`.

One way to handle events is to implement the appropriate listener classes.

Listener classes that handle the action events in an application must implement the interface `javax.faces.event.ActionListener`.

Similarly, listeners that handle the value-change events must implement the interface `javax.faces.event.ValueChangeListener`.

This section explains how to implement the two listener classes in backing beans.

To handle events generated by custom components, you must implement an event listener and an event handler and manually queue the event on the component.

See Handling Events for Custom Components for more information.

Note - You do not need to create an `ActionListener` implementation to handle an event that results solely in navigating to a page and does not perform any other application-specific processing.

See Writing a Method to Handle Navigation for information on how to manage page navigation.

Implementing Value-Change Listeners

A `ValueChangeListener` implementation must include a `processValueChange (ValueChangeEvent)` method.

This method processes the specified value-change event and is invoked by the JavaServer Faces implementation when the value-change event occurs.

The `ValueChangeEvent` instance stores the old and the new values of the component that fired the event.

Here is part of a listener implementation:

```
...
public class NameChanged extends
Object implements
ValueChangeListener{
```

```
public void processValueChange
(ValueChangeEvent event)
throws AbortProcessingException {
if (null != event.getNewValue()){
FacesContext.getCurrentInstance().
getExternalContext().getSessionMap
().put("name",event.getNewValue());}}
}
```

The `NameChanged` listener implementation is registered on a `UIInput` component of a web page.

This listener stores into session scope the name that the user entered in the text field corresponding to the `name` component.

When the user enters the name in the text field, a value-change event is generated, and the `processValueChange(ValueChangeEvent)` method of the `NameChanged` listener implementation is invoked.

This method first gets the ID of the component that fired the event from the `ValueChangeEvent` object, and it puts the value, along with an attribute name, into the session map of the `FacesContext` instance.

Registering a Value-Change Listener on a Component explains how to register this listener onto a component.

The custom converter is used by the Facelets page as follows:

```
<mystore:store>
<f:valueChangeListener
type="com.bookstore6.NameChanged"
>
</mystore:store>
```

Implementing Action Listeners

An `ActionListener` implementation must include a `processAction(ActionEvent)` method.

The `processAction(ActionEvent)` method processes the specified action event.

The JavaServer Faces implementation invokes the `processAction(ActionEvent)` method when the `ActionEvent` occurs.

For example, suppose you have a Facelets page that allows the user to select a locale for the application by clicking one of a set of hyperlinks.

When the user clicks one of the hyperlinks, an action event is generated.

The listener implementation would look like this:

```
...
public class LocaleChangeListener
extends Object implements
ActionListener {
private HashMap<String, Locale>
locales = null;
public LocaleChangeListener() {
locales =
new HashMap<String, Locale>(4);
```

```
locales.put("NAmerica",
new Locale("en", "US"));
locales.put("SAmerica",
new Locale("es", "MX"));
locales.put("Germany",
new Locale("de", "DE"));
locales.put("France",
new Locale("fr", "FR"));
}
```

```
public void
processAction(ActionEvent event)
throws AbortProcessingException {
String current =
event.getComponent().getId();
FacesContext context =
FacesContext.getCurrentInstance();
context.getViewRoot().setLocale
((Locale)
locales.get(current));
} }
```

Registering an Action Listener on a Component

explains how to register this listener onto a component.

Creating a Custom Validator

If the standard validators or Bean Validation don't perform the validation checking you need, you can create a custom validator to validate user input.

There are two ways to implement validation code:

- **Implement a backing bean method that performs the validation.**

- **Provide an implementation of the `Validator` interface to perform the validation.**

Writing a Method to Perform Validation explains how to implement a backing bean method to perform validation.

The rest of this section explains how to implement the `Validator` interface.

If you choose to implement the `Validator` interface and you want to allow the page author to configure the validator's attributes from the page, you also must create a custom tag for registering the validator on a component.

If you prefer to configure the attributes in the `Validator` implementation, you can forgo creating a custom tag and instead let the page author register the validator on a component using the `validator` tag, as described in Using a Custom Validator.

You can also create a backing bean property that accepts and returns the `Validator` implementation you create, as described in Writing Properties Bound to Converters, Listeners, or Validators.

You can use the `validator` tag's binding attribute to bind the `Validator` implementation to the backing bean property.

Usually, you will want to display an error message when data fails validation.

You need to store these error messages in a resource bundle.

After creating the resource bundle, you have two ways to make the messages available to the application.

You can queue the error messages onto the `FacesContext` programmatically, or you can register the error messages in the application configuration resource file, as explained in Registering Custom Error Messages.

For example, an e-commerce application might use a general-purpose custom validator called `FormatValidator.java` to validate input data against a format pattern that is specified in the custom validator tag.

This validator would be used with a Credit Card Number field on a Facelets page.

Here is the custom validator tag:

```
<mystore:formatValidator
formatPatterns="9999999999999999|
9999 9999 9999 9999|
9999-9999-9999-9999"
/>
```

According to this validator, the data entered in the field must be one of the following:

- A 16–digit number with no spaces
- A 16–digit number with a space between every four digits
- A 16–digit number with hyphens between every four digits

The rest of this section describes how this validator would be implemented and how to create a custom tag so that the page author can register the validator on a component.

Implementing the Validator Interface

A `Validator` 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 also defines accessor methods for setting the `formatPatterns` attribute, which specifies the acceptable format patterns for input into the fields.

In addition, the class overrides the `validate` method of the `Validator` interface.

This method validates the input and also accesses the custom error messages to be displayed when the `String` is invalid.

The `validate` method performs the actual validation of the data.

It takes the `FacesContext` instance, the component whose data needs to be validated, and the value that needs to be validated.

A validator can validate only data of a component that implements `EditableValueHolder`.

Here is an implementation of the `validate` method:

```java
@FacesValidator
public void validate
(FacesContext context,
UIComponent component,
Object toValidate){
boolean valid = false;
String value = null;
if ((context == null) ||
(component == null)) {
throw new NullPointerException();
}
```

```
if(!(component instanceof UIInput))
{ return; }
if ( null == formatPatternsList ||
null == toValidate)
{ return; }
value = toValidate.toString();
// validate the value against
// the list of valid patterns.
Iterator patternIt =
formatPatternsList.iterator();
while (patternIt.hasNext()) {
```

```
valid = isFormatValid
(((String)patternIt.next()),value);
if (valid) { break; }
}
if ( !valid ) {
FacesMessage errMsg =
MessageFactory.getMessage(context,
FORMAT_INVALID_MESSAGE_ID,
(new Object[] {formatPatterns}));
throw
new ValidatorException(errMsg);} }
```

The `@FacesValidator` annotation registers the above method as a converter with the JavaServer Faces implementation.

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 custom validator tag.

While iterating over the list, this method checks the pattern of the component's local value against the patterns in the list.

If the pattern of the local value does not match any pattern in the list, this method generates an error message.

It then passes the message to the constructor of `ValidatorException`.

Eventually the message is queued onto the `FacesContext` instance so that the message is displayed on the page during the Render Response phase.

The error messages are retrieved from the `Application` instance by `MessageFactory`.

An application that creates its own custom messages must provide a class, such as `MessageFactory`, that retrieves the messages from the `Application` instance.

The `getMessage(FacesContext, String, Object)` method of `MessageFactory` takes a `FacesContext`, a static `String` that represents the key into the `Properties` file, and the format pattern as an `Object`.

The key corresponds to the static message ID in the `FormatValidator` class:

```
public static final String
FORMAT_INVALID_MESSAGE_ID =
"FormatInvalid";
}
```

When the error message is displayed, the format pattern will be substituted for the `{0}` in the error message, which, in English, is as follows:

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

JavaServer Faces applications can save the state of validators and components on either the client or the server.

Specifying Where State Is Saved explains how to configure your application to save state on either the client or the server.

If your JavaServer Faces application saves state on the client (which is the default), you need to make the `Validator` implementation implement `StateHolder` as well as `Validator`.

In addition to implementing `StateHolder`, the `Validator` implementation needs to implement the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods of `StateHolder`.

With these methods, the `Validator` implementation tells the JavaServer Faces implementation which attributes of the `Validator` implementation to save and restore across multiple requests.

To save a set of values, you must implement the `saveState(FacesContext)` method.

This method is called during the Render Response phase, during which the state of the response is saved for processing on subsequent requests.

When implementing the `saveState(FacesContext)` method, you need to create an array of objects and add the values of the attributes you want to save to the array.

Here is the `saveState(FacesContext)` method from the custom validator class:

```
public Object
saveState(FacesContext context){
Object values[] = new Object[2];
values[0] = formatPatterns;
values[1] = formatPatternsList;
return (values);
}
```

To restore the state saved with the `saveState(FacesContext)` method in preparation for the next postback, the `Validator` implementation implements `restoreState(FacesContext, Object)`.

The `restoreState(FacesContext, Object)` method takes the `FacesContext` instance and an `Object` instance, which represents the array that is holding the state for the `Validator` implementation.

This method sets the `Validator` implementation's properties to the values saved in the `Object` array.

Here is the `restoreState(FacesContext, Object)` method from `FormatValidator`:

```
public void restoreState
(FacesContext context,Object state){
Object values[] = (Object[]) state;
formatPatterns =(String) values[0];
formatPatternsList =
(ArrayList) values[1];
}
```

As part of implementing `StateHolder`, the custom `Validator` implementation must also override the `isTransient` and `setTransient(boolean)` methods of `StateHolder`.

By default, `transientValue` is false, which means that the `Validator` implementation will have its state information saved and restored.

Here are the `isTransient` and `setTransient(boolean)` methods of `FormatValidator:`

```
private boolean transientValue =
false;
public boolean isTransient()
{ return (this.transientValue); }
public void setTransient
(boolean transientValue){
```

```
this.transientValue =
transientValue;
}
```

Saving and Restoring State describes how a custom component must implement the `saveState(FacesContext)` and `restoreState(FacesContext, Object)` methods.

Creating a Custom Tag

If you implemented a `Validator` interface rather than implementing a backing bean method that performs the validation, you need to do one of the following:

- Allow the page author to specify the `Validator` implementation to use with the `validator` tag.

In this case, the `Validator` implementation must define its own properties.

Using a Custom Validator explains how to use the `validator` tag.

- Create a custom tag that provides attributes for configuring the properties of the validator from the page.

Because the `Validator` implementation from the preceding section does not define its attributes, the application developer must create a custom tag so that the page author can define the format patterns in the tag.

To create a custom tag, you need to do two things:

- Write a tag handler to create and register the `Validator` implementation on the component.
- Write a TLD to define the tag and its attributes.

Using a Custom Validator explains how to use the custom validator tag on the page.

Writing the Tag Handler

The tag handler associated with a custom validator tag must extend the `ValidatorELTag` class.

This class is the base class for all custom tag handlers that create `Validator` instances and register them on UI components.

The `FormatValidatorTag` class registers the `FormatValidator` instance onto the component.

The `FormatValidatorTag` tag handler class does the following:

- Sets the ID of the validator.
- Provides a set of accessor methods for each attribute defined on the tag.
- Implements the `createValidator` method of the `ValidatorELTag` class.

This method creates an instance of the validator and sets the range of values accepted by the validator.

The `formatPatterns` attribute of the `formatValidator` tag supports literals and value expressions.

Therefore, the accessor method for this attribute in the `FormatValidatorTag` class must accept and return an instance of `ValueExpression`:

```
protected ValueExpression
formatPatterns = null;
public void setFormatPatterns
(ValueExpression fmtPatterns)
{ formatPatterns = fmtPatterns; }
```

Finally, the `createValidator` method creates an instance of `FormatValidator`, extracts the value from the `formatPatterns` attribute's value expression and sets the `formatPatterns` property of `FormatValidator` to this value:

the `formatPatterns` property of `FormatValidator` to this value:

```
protected Validator
createValidator()
throws JspException {
FacesContext facesContext =
FacesContext.getCurrentInstance();
FormatValidator result = null;
if(validatorID != null){
result = (FormatValidator)
facesContext.getApplication()
.createValidator(validatorID);
}
```

```
String patterns = null;
if(formatPatterns != null){
if(!formatPatterns.isLiteralText()){
patterns = (String)
formatPatterns.getValue
(facesContext.getELContext());
} else {
patterns = formatPatterns.
getExpressionString();
}
```

Writing the Tag Library Descriptor

To define a tag, you 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 it.

The custom validator tag is defined in a TLD that contains a tag definition for formatValidator:

```
<tag>
<name>formatValidator</name>
...
<tag-class>
mystore.taglib.FormatValidatorTag
</tag-class>
```

```
<attribute>
<name>formatPatterns</name>
<required>true</required>
<deferred-value>
<type>String</type>
</deferred-value>
</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.

The `formatPatterns` attribute is the only attribute that the tag supports.

The `deferred-value` element indicates that the `formatPatterns` attribute accepts deferred value expressions.

The `type` element says that the expression resolves to a property of type `String`.

Using Custom Objects

As a page author, you might need to use custom converters, validators, or components packaged with the application on your Facelets pages.

A custom converter is applied to a component in one of the following ways:

- Reference the converter from the component tag's `converter` attribute.

- Nest a `converter` tag inside the component's tag and reference the custom converter from one of the `converter` tag's attributes.

A custom validator is applied to a component in one of the following ways:

- Nest a `validator` tag inside the component's tag and reference the custom validator from the `validator` tag.

- Nest the validator's custom tag (if there is one) inside the component's tag.

To use a custom component, you add the custom tag associated with the component to the page.

As explained in Setting Up a Page, you must ensure that the TLD that defines any custom tags is packaged in the application if you intend to use the tags in your pages.

TLD files are stored in the `WEB-INF/` directory or subdirectory of the WAR file or in the `META-INF/` directory or subdirectory of a tag library packaged in a JAR file.

The next three sections describe how to use custom converter, validator, and UI components.

Using a Custom Converter

As described in the previous section, to apply the data conversion performed by a custom converter to a particular component's value, you must either reference the custom converter from the component tag's `converter` attribute or from a `converter` tag nested inside the component tag.

If you are using the component tag's `converter` attribute, this attribute must reference the `Converter` implementation's identifier or the fully-qualified class name of the converter.

Creating a Custom Converter explains how a custom converter is implemented.

The identifier for the credit card converter is `CreditCardConverter`.

The `CreditCardConverter` instance is registered on the `ccno` component, as shown in the following example:

```
<h:inputText id="ccno"
size="19"
converter="CreditCardConverter"
required="true"
>
...
</h:inputText>
```

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

Instead of referencing the converter from the component tag's `converter` attribute, you can reference the converter from a `f:converter` tag nested inside the component's tag.

To reference the custom converter using the `converter` tag, you do one of the following:

- Set the `f:converter` tag's `converterId` attribute to the `Converter` implementation's identifier defined in the `@FacesConverter` annotation or in the application configuration resource file.

This method is shown in the following example:

```
<h:inputText id="ccno"
size="19"
>
<f:converter
converterId="CreditCardConverter"
/>
</h:inputText>
```

- Bind the `Converter` implementation to a backing bean property using the `converter` tag's `binding` attribute, as described in Binding Converters, Listeners, and Validators to Backing Bean Properties.

Using a Custom Validator

To register a custom validator on a component, you must do one of the following:

- Nest the validator's custom tag inside the tag of the component whose value you want to be validated.

- Nest the standard `validator` tag within the tag of the component and reference the custom `Validator` implementation from the `validator` tag.

Here is the custom `formatValidator` tag for the Credit Card Number field:

```
<h:inputText id="ccno" size="19"
...
required="true"
>
<mystore:formatValidator
formatPatterns="9999999999999999|
9999 9999 9999 9999|
9999-9999-9999-9999"
/>
</h:inputText>
```

```
<h:message
styleClass="validationMessage"
for="ccno"
/>
```

This tag validates the input of the `ccno` field against the patterns defined by the page author in the `formatPatterns` attribute.

You can use the same custom validator for any similar component by simply nesting the custom validator tag within the component tag.

If the application developer who created the custom validator prefers to configure the attributes in the `Validator` implementation rather than allow the page author to configure the attributes from the page, the developer will not create a custom tag for use with the validator.

In this case, the page author must nest the `validator` tag inside the tag of the component whose data needs to be validated.

Then the page author needs to do one of the following:

- Set the `validator` tag's `validatorId` attribute to the ID of the validator that is defined in the application configuration resource file.

- **Bind the custom `Validator` implementation to a backing bean property using the `validator` tag's `binding` attribute, as described in Binding Converters, Listeners, and Validators to Backing Bean Properties.**

The following tag registers a hypothetical validator on a component using a `validator` tag and references the ID of the validator:

```
<h:inputText id="name"
value="#{CustomerBean.name}"
size="10" ...
>
<f:validator
validatorId="customValidator"
/>
...
</h:inputText>
```

Using a Custom Component

In order to use a custom component in a page, you need to declare the tag library that defines the custom tag that renders the custom component, as explained in Using Custom Objects, and you add the component's tag to the page.

The Duke's Bookstore application includes a custom image map component on the `chooselocale.jsp` page.

This component allows you to select the locale for the application by clicking on a region of the image map:

```
...
<h:graphicImage id="mapImage"
url="/template/world.jpg"
alt="#{bundle.chooseLocale}"
usemap="#worldMap"
/>
<bookstore:map id="worldMap"
current="NAmericas"
immediate="true"
action="bookstore"
actionListener=
```

```
"#{localeBean.chooseLocaleFromMap}"
>
<bookstore:area id="NAmerica"
value="#{NA}"
onmouseover=
"/template/world_namer.jpg"
onmouseout="/template/world.jpg"
targetImage="mapImage"
/>
```

```
...
<bookstore:area id="France"
value="#{fraA}"
onmouseover=
"/template/world_france.jpg"
onmouseout="/template/world.jpg"
targetImage="mapImage"
/>
</bookstore:map>
```

The standard `graphicImage` tag associates an image (`world.jpg`) with an image map that is referenced in the `usemap` attribute value.

The custom `map` tag that represents the custom component, `MapComponent`, specifies the image map, and contains a set of `area` tags.

Each custom `area` tag represents a custom `AreaComponent` and specifies a region of the image map.

On the page, the `onmouseover` and `onmouseout` attributes specify the image that is displayed when the user performs the actions described by the attributes.

The page author defines what these images are.

The custom renderer also renders an `onclick` attribute.

In the rendered HTML page, the `onmouseover`, `onmouseout`, and `onclick` attributes define which JavaScript code is executed when these events occur.

When the user moves the mouse over a region, the `onmouseover` function associated with the region displays the map with that region highlighted.

When the user moves the mouse out of a region, the `onmouseout` function redisplays the original image.

When the user clicks a region, the `onclick` function sets the value of a hidden `input` tag to the ID of the selected area and submits the page.

When the custom renderer renders these attributes in HTML, it also renders the JavaScript code.

The custom renderer also renders the entire `onclick` attribute rather than let the page author set it.

The custom renderer that renders the `map` tag also renders a hidden `input` component that holds the current area.

The server-side objects retrieve the value of the hidden `input` field and set the locale in the `FacesContext` instance according to which region was selected.

Binding Component Values and Instances to External Data Sources

A component tag can wire its data to a back-end data object by one of the following methods:

- Binding its component's value to a bean property or other external data source
- Binding its component's instance to a bean property

A component tag's `value` attribute uses a EL value expression to bind the component's value to an external data source, such as a bean property.

A component tag's `binding` attribute uses a value expression to bind a component instance to a bean property.

When a component instance is bound to a backing bean property, the property holds the component's local value.

Conversely, when a component's value is bound to a backing bean property, the property holds the value stored in the backing bean.

This value is updated with the local value during the update model values phase of the life cycle.

There are advantages to both of these methods.

Binding a component instance to a bean property has these advantages:

- **The backing bean can programmatically modify component attributes.**
- **The backing bean can instantiate components rather than let the page author do so.**

Binding a component's value to a bean property has these advantages:

- The page author has more control over the component attributes.
- The backing bean has no dependencies on the JavaServer Faces API (such as the component classes), allowing for greater separation of the presentation layer from the model layer.

- The JavaServer Faces implementation can perform conversions on the data based on the type of the bean property without the developer needing to apply a converter.

In most situations, you will bind a component's value rather than its instance to a bean property.

You'll need to use a component binding only when you need to change one of the component's attributes dynamically.

For example, if an application renders a component only under certain conditions, it can set the component's `rendered` property accordingly by accessing the property to which the component is bound.

When referencing the property using the component tag's `value` attribute, you need to use the proper syntax.

For example, suppose a backing bean called `MyBean` has this `int` property:

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

The value attribute that references this property must have this value-binding expression:

`#{MyBean.currentOption}`

In addition to binding a component's value to a bean property, the `value` attribute can specify a literal value or can 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.

Table 14-2 lists some example value-binding expressions that you can use with the `value` attribute.

Table 14-2 Example Value-Binding Expressions

Value	Expression
A Boolean	`cart.numberOfItems > 0`
A property initialized from a context `init` parameter	`initParam.quantity`
A bean property	`CashierBean.name`
Value in an array	`books[3]`
Value in a collection	`books["fiction"]`
Property of an object in an array of objects	`books[3].price`

The next two sections explain how to use the `value` attribute to bind a component's value to a bean property or other external data sources, and how to use the `binding` attribute to bind a component instance to a bean property.

Binding a Component Value to a Property

To bind a component's value to a bean property, you specify the name of the bean and the property using the `value` attribute.

This means that the name of the bean in the EL value expression must match the `managed-bean-name` element of the managed bean declaration up to the first period (.) in the expression.

Similarly, the part of the value expression after the period must match the name specified in the corresponding `property-name` element in the application configuration resource file.

This means that the first part of the EL value expression must match the name of the backing bean up to the first period (.) and the part of value expression after the period must match the property of the backing bean.

Much of the time you will not include definitions for a managed bean's properties when configuring it.

You need to define a property and its value only when you want the property to be initialized with a value when the bean is initialized.

Binding a Component Value to an Implicit Object

One external data source that a `value` attribute can refer to is an implicit object.

The following example shows a reference to an implicit object:

```
<h:outputFormat title="thanks"
value="#{bundle.ThankYouParam}">
<f:param
value="#{sessionScope.name}"/>
</h:outputFormat>
```

This tag gets the name of the customer from the session scope and inserts it into the parameterized message at the key `ThankYouParam` from the resource bundle.

For example, if the name of the customer is Gwen Canigetit, this tag will render:

```
Thank you, Gwen Canigetit, for
purchasing your books from us.
```

Retrieving values from other implicit objects is done in a similar way to the example shown in this section.

Table 14-3 lists the implicit objects to which a value attribute can refer.

All of the implicit objects, except for the scope objects, are read-only and therefore should not be used as a value for a `UIInput` component.

Table 14-3 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
`header`	A `Map` of HTTP header values for the current request, keyed by header name
`headerValues`	A `Map` of `String` arrays containing all 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 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
`view`	The root `UIComponent` in the current component tree stored in the `FacesRequest` for this request

Binding a Component Instance to a Bean Property

A component instance can be bound to a bean property using a value expression with the `binding` attribute of the component's tag.

You usually bind a component instance rather than its value to a bean property if the bean must dynamically change the component's attributes.

Here are two tags that bind components to bean properties:

```
<h:selectBooleanCheckbox
id="fanClub"
rendered="false"
binding="#{cashier.specialOffer}"
/>
<h:outputLabel for="fanClub"
rendered="false"
binding=
"#{cashier.specialOfferText}"
>
```

```
<h:outputText id="fanClubLabel"
value="#{bundle.DukeFanClub}"
/>
</h:outputLabel>
```

The `selectBooleanCheckbox` tag renders a check box and binds the `fanClub` `UISelectBoolean` component to the `specialOffer` property of the `cashier` bean.

The `outputLabel` tag binds the component representing the check box's label to the `specialOfferText` property of the `cashier` bean.

The `rendered` attributes of both tags are set to `false`, which prevents the check box and its label from being rendered.

If the customer makes a large order, the backing bean could set both components' `rendered` properties to `true`, causing the check box and its label to be rendered.

These tags use component bindings rather than value bindings, because the backing bean must dynamically set the values of the components' `rendered` properties.

If the tags were to use value bindings instead of component bindings, the backing bean would not have direct access to the components, and would therefore require additional code to access the components from the `FacesContext` instance to change the components' `rendered` properties.

Binding Converters, Listeners, and Validators to Backing Bean Properties

As described in Adding Components to a Page Using HTML Tags, a page author can bind converter, listener, and validator implementations to backing bean properties using the `binding` attributes of the tags which are used to register the implementations on components.

This technique has similar advantages to binding component instances to backing bean properties, as described in Binding Component Values and Instances to External Data Sources.

In particular, binding a converter, listener, or validator implementation to a backing bean property yields the following benefits:

- The backing bean can instantiate the implementation instead of allowing the page author to do so.

- The backing bean can programmatically modify the attributes of the implementation.

In the case of a custom implementation, the only other way to modify the attributes outside of the implementation class would be to create a custom tag for it and require the page author to set the attribute values from the page.

Whether you are binding a converter, listener, or validator to a backing bean property, the process is the same for any of the implementations:

- Nest the converter, listener, or validator tag within an appropriate component tag.

- Make sure that the backing bean has a property that accepts and returns the converter, listener, or validator implementation class that you want to bind to the property.

- **Reference the backing bean property using a value expression from the `binding` attribute of the converter, listener, or validator tag.**

For example, say that you want to bind the standard `DateTime` converter to a backing bean property because the application developer wants the backing bean to set the formatting pattern of the user's input rather than let the page author do it.

First, the page author registers the converter onto the component by nesting the `convertDateTime` tag within the component tag.

Then, the page author references the property with the `binding` attribute of the `convertDateTime` tag:

```
<h:inputText
value="#{LoginBean.birthDate}"
>
<f:convertDateTime
binding="#{LoginBean.convertDate}"
/>
</h:inputText>
```

The `convertDate` property would look something like this:

```
private
DateTimeConverter convertDate;
public DateTimeConverter
getConvertDate()
{... return convertDate; }
public void setConvertDate
(DateTimeConverter convertDate){
convertDate.setPattern
("EEEEEEEE, MMM dd, yyyy");
this.convertDate = convertDate;
}
```

See Writing Properties Bound to Converters, Listeners, or Validators for more information on writing backing bean properties for converter, listener, and validator implementations.