JavaServer Faces Technology Advanced Concepts

Previous chapters have introduced JavaServer Faces technology and Facelets, the preferred presentation layer for the Java for the Java EE platform.

This chapter and the following chapters introduce advanced concepts in this area.

- This chapter describes the JavaServer Faces lifecycle in detail.

Some of the complex applications use the well-defined and identified lifecycle phases to customize application behavior.

- Chapter 11, Configuring JavaServer Faces Applications, introduces the process of creating and deploying JavaServer Faces applications, the use of various configuration files, and the deployment structure.

- Chapter 12, Using Ajax with JavaServer Faces Technology, introduces Ajax concepts and the use of Ajax in JavaServer Faces applications.

- Chapter 13, Advanced Composite Components, introduces advanced features of composite components.

- Chapter 14, Creating Custom UI Components, describes the process of creating new components, renderers, converters, listeners, and validators from scratch.

The following topics are addressed here:

Overview of the JavaServer Faces Lifecycle

The lifecycle of an application refers to the various stages of processing that application, from its initiation to its conclusion.

All applications have lifecycles.

During a web application lifecycle, common tasks such as the following are performed:

- Handling incoming requests
- Decoding parameters
- Modifying and saving state
- Rendering web pages to the browser

The JavaServer Faces web application framework manages its lifecycle phases automatically for simple applications or allows you to manage them manually for more complex applications as required.

JavaServer Faces applications that use advanced features may require interaction with the lifecycle at certain phases.

For example, Ajax applications use partial processing features of the lifecycle.

A clearer understanding of the lifecycle phases is key to creating well-designed components.

A simplified view of the JavaServer faces lifecycle, consisting of the two main phases of a JavaServer Faces web application, is introduced in The Lifecycle of the `hello` Application.

This chapter examines the JavaServer Faces lifecycle in more detail.

The Lifecycle of a JavaServer Faces Application

The lifecycle of a JavaServer Faces application begins when the client makes an HTTP request for a page and ends when the server responds with the page, translated to HTML.

The lifecycle can be divided into two main phases, `execute` and `render`.

The execute phase is further divided into sub-phases to support the sophisticated component tree.

This structure requires that component data be converted and validated, component events be handled, and component data be propagated to beans in an orderly fashion.

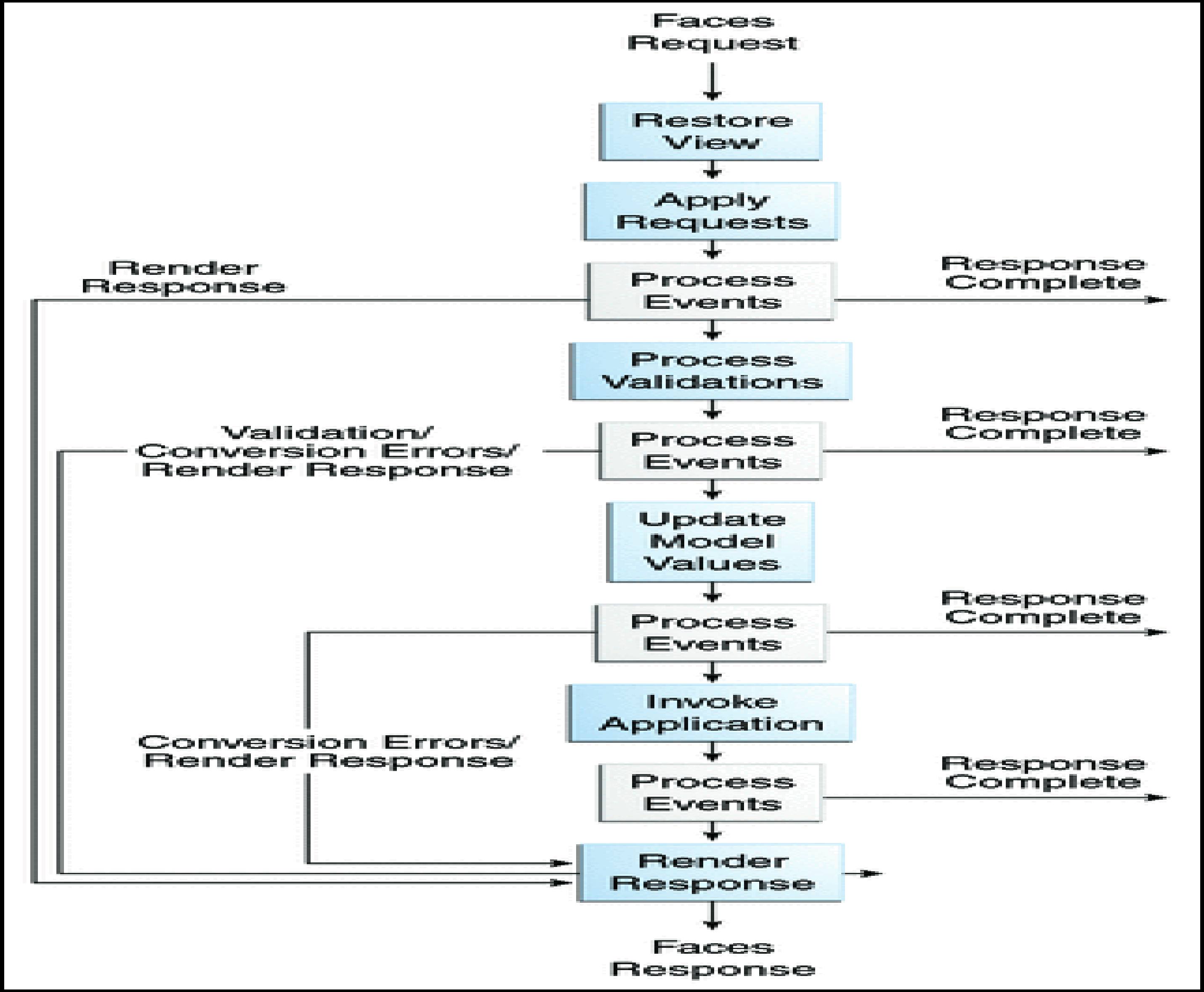
A JavaServer Faces page is represented by a tree of components, called a view.

During the lifecycle, the JavaServer Faces implementation must build the view while considering the state saved from a previous submission of the page.

When the client submits a page, the JavaServer Faces implementation performs several tasks, such as validating the data input of components in the view and converting input data to types specified on the server side.

The JavaServer Faces implementation performs all these tasks as a series of steps in the JavaServer Faces request-response lifecycle. Figure 10-1 illustrates these steps.

Figure 10-1 JavaServer Faces Standard Request-Response Lifecycle



The lifecycle handles two kinds of requests: initial requests and postbacks.

An initial request occurs when a user makes a request for a page for the first time.

A postback request occurs when a user submits the form contained on a page that was previously loaded into the browser as a result of executing an initial request.

When the lifecycle handles an initial request, it executes only the Restore View and Render Response phases, because there is no user input or actions to process.

Conversely, when the lifecycle handles a postback, it executes all of the phases.

Usually, the first request for a JavaServer Faces page comes in from a client, as a result of clicking a hyperlink on an HTML page that links to the JavaServer Faces page.

To render a response that is another JavaServer Faces page, the application creates a new view and stores it in the `FacesContext` instance, which represents all of the information associated with processing an incoming request and creating a response.

The application then acquires object references needed by the view and calls `FacesContext.renderResponse` method, which forces immediate rendering of the view by skipping to the Render Response Phase of the lifecycle, as is shown by the arrows labelled Render Response in the diagram.

Sometimes, an application might need to redirect to a different web application resource, such as a web service, or generate a response that does not contain JavaServer Faces components.

In these situations, the developer must skip the Render Response phase by calling the `FacesContext.responseComplete` method.

This situation is also shown in the diagram, this time with the arrows labelled Response Complete.

The most common situation is that a JavaServer Faces component submits a request for another JavaServer Faces page.

In this case, the JavaServer Faces implementation handles the request and automatically goes through the phases in the lifecycle to perform any necessary conversions, validations, and model updates, and to generate the response.

There is one exception to the lifecycle described in this section.

When a component's `immediate` attribute is set to `true`, the validation, conversion, and events associated with these components are processed during the Apply Request Values Phase rather than in a later phase.

The details of the lifecycle explained in the following sections are primarily intended for developers who need to know information such as when validations, conversions, and events are usually handled and what they can do to change how and when they are handled.

For more information on each of the lifecycle phases, download the latest JavaServer Faces Specification documentation from https://javaserverfaces.java.net/.

The JavaServer Faces application lifecycle contains the following phases:

- Restore View Phase
- Apply Request Values Phase
- Process Validations Phase
- Update Model Values Phase
- Invoke Application Phase
- Render Response Phase

Restore View Phase

When a request for a JavaServer Faces page is made, usually by an action on the page, such as when a link or a button is clicked, the JavaServer Faces implementation begins the Restore View phase.

During this phase, the JavaServer Faces implementation builds the view of the page, wires event handlers and validators to components in the view, and saves the view in the `FacesContext` instance, which contains all the information needed to process a single request.

All the application's component tags, event handlers, converters, and validators have access to the `FacesContext` instance.

If the request for the page is an initial request, the JavaServer Faces implementation creates an empty view during this phase and the lifecycle advances to the Render Response phase, during which the empty view is populated with the components referenced by the tags in the page.

If the request for the page is a postback, a view corresponding to this page already exists.

During this phase, the JavaServer Faces implementation restores the view by using the state information saved on the client or the server.

Apply Request Values Phase

After the component tree is restored during a postback request, each component in the tree extracts its new value from the request parameters by using its `decode` (`processDecodes()`) method.

The value is then stored locally on the component.

If the conversion of the value fails, an error message that is associated with the component is generated and queued on `FacesContext`.

This message will be displayed during the Render Response phase, along with any validation errors resulting from the Process Validations phase.

If any `decode` methods or event listeners have called `renderResponse` on the current `FacesContext` instance, the JavaServer Faces implementation skips to the Render Response phase.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts the events to interested listeners.

If some components on the page have their `immediate` attributes (see The `immediate` Attribute) set to `true`, then the validation, conversion, and events associated with these components will be processed during this phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete`.

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

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

Process Validations Phase

During this phase, the JavaServer Faces implementation processes all validators registered on the components in the tree, by using its `validate` (`processValidators`) method.

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` instance, 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 the Apply Request Values phase, the messages for these errors are also displayed.

If any `validate` methods or event listeners have called the `renderResponse` method on the current `FacesContext`, the JavaServer Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the `FacesContext.responseComplete` method.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the `Faces Context`, and the partial processing method is applied.

Update Model Values Phase

After the JavaServer Faces implementation determines that the data is valid, it traverses the component tree and sets the corresponding server-side object properties to the components' local values.

The JavaServer Faces implementation updates only the bean properties pointed at by an input component's value attribute.

If the local data cannot be converted to the types specified by the bean properties, the lifecycle advances directly to the Render Response phase so that the page is re-rendered with errors displayed.

This is similar to what happens with validation errors.

If any `updateModels` methods or any listeners have called the `renderResponse` method on the current `FacesContext` instance, the JavaServer Faces implementation skips to the Render Response phase.

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call `FacesContext.responseComplete` method.

If events have been queued during this phase, the JavaServer Faces implementation broadcasts them to interested listeners.

If the current request is identified as a partial request, the partial context is retrieved from the `FacesContext`, and the partial processing method is applied.

Invoke Application Phase

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

At this point, if the application needs to redirect to a different web application resource or generate a response that does not contain any JavaServer Faces components, it can call the `FacesContext.responseComplete` method.

If the view being processed was reconstructed from state information from a previous request and if a component has fired an event, these events are broadcast to interested listeners.

Finally, the JavaServer Faces implementation transfers control to the Render Response phase.

Render Response Phase

During this phase, JavaServer Faces builds the view and delegates authority to the appropriate resource for rendering the pages.

If this is an initial request, the components that are represented on the page will be added to the component tree.

If this is not an initial request, the components are already added to the tree, so they need not be added again.

If the request is a postback and 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 `message` or `messages` tags, any queued error messages are displayed on the page.

After the content of the view is rendered, the state of the response is saved so that subsequent requests can access it.

The saved state is available to the Restore View phase.

Partial Processing and Partial Rendering

The JavaServer Faces lifecycle spans the entire execute and render processes of applications.

It is also possible to process and render only parts of an application, such as a single component.

For example, the JavaServer Faces Ajax framework can generate requests containing information on which particular component may be processed and which particular component may be rendered back to the client.

Once such a partial request enters the JavaServer Faces lifecycle, the information is identified and processed by a `javax.faces.context.PartialViewContext` object.

The JavaServer Faces lifecycle is still aware of such Ajax requests and modifies the component tree accordingly.

The execute and render attributes of the `f:ajax` tag are used to identify which components may be executed and rendered.

For more information on these attributes, see Chapter 12, Using Ajax with JavaServer Faces Technology.

The Lifecycle of a Facelets Application

The JavaServer Faces specification defines the lifecycle of a JavaServer Faces application.

For more information on this lifecycle, see The Lifecycle of a JavaServer Faces Application.

The following steps describe that process as applied to a Facelets based application.

- When a client, such as a browser, makes a new request to a page that is created using Facelets, a new component tree or `UIViewRoot` is created and placed in the `FacesContext`.

- The `UIViewRoot` is applied to the Facelets, and the view is populated with components for rendering.

- The newly built view is rendered back as a response to the client.

- On rendering, the state of this view is stored for the next request.

The state of input components and form data is stored.

- The client may interact with the view and request another view or change from the JavaServer Faces application.

At this time the saved view is restored from the stored state.

- The restored view is once again passed through the JavaServer Faces lifecycle and eventually will either generate a new view or re-render the current view if there were no validation problems or no action was triggered.

- If the same view is requested, the stored view is rendered once again.

- If a new view is requested, then the process described in the second step is continued.

- The new view is then rendered back as a response to the client.

User Interface Component Model

In addition to the lifecycle description, an overview of JavaServer Faces architecture provides better understanding of the technology.

JavaServer Faces components are the building blocks of a JavaServer Faces view.

A component can be a user interface (UI) component or a non-UI component.

JavaServer Faces UI components are configurable, reusable elements that compose the user interfaces of JavaServer Faces applications.

A component can be simple, such as a button, or can be compound, such as a table, composed of multiple components.

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

- 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 various ways

- An event and listener model that defines how to handle component events
- A conversion model that defines how to register 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.

User Interface Component Classes

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

The component classes are completely extensible, allowing component writers to create their own custom components.

See Chapter 14, Creating Custom UI Components for more information.

The abstract base class for all components is `javax.faces.component.UIComponent`.

JavaServer Faces UI component classes extend `UIComponentBase` class, (a subclass of `UIComponent` class) which defines the default state and behavior of a component.

The following set of component classes is included with JavaServer Faces technology:

- `UIColumn`: Represents a single column of data in a `UIData` component.

- `UICommand`: Represents a control that fires actions when activated.
- `UIData`: Represents a data binding to a collection of data represented by a `DataModel` instance.
- `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`.

- `UIMessage`: Displays a localized error message.

- `UIMessages`: Displays a set of localized error messages.

- `UIOutcomeTarget`: Displays a hyperlink in the form of a link or a button.

- `UIOutput`: Displays data output on a page.
- `UIPanel`: Manages the layout of its child components.
- `UIParameter`: Represents substitution parameters.

- `UISelectBoolean`: Allows a user to set a `boolean` value on a control by selecting or deselecting it.

This class is a subclass of `UIInput` class.

- `UISelectItem`: Represents a single item in a set of items.

- `UISelectItems`: Represents an entire set of items.

- `UISelectMany`: Allows a user to select multiple items from a group of items.

This class is a subclass of `UIInput` class.

- `UISelectOne`: Allows a user to select one item from a group of items.

This class is a subclass of `UIInput` class.

- `UIViewParameter`: Represents the query parameters in a request.

This class is a subclass of `UIInput` class.

- `UIViewRoot`: Represents the root of the component tree.

In addition to extending `UIComponentBase`, the component classes also implement one or more behavioral interfaces, each of which defines certain behavior for a set of components whose classes implement the interface.

These behavioral interfaces are as follows:

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

This interface is intended for use with components based on JavaServer Faces technology 1.1_01 and earlier versions.

This interface is deprecated in JavaServer Faces 2.x.

- `ActionSource2`: Extends `ActionSource`, and therefore provides the same functionality.

However, it allows components to use the unified EL when they are referencing methods that handle action events.

- `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.

- `SystemEventListenerHolder`: Maintains a list of `SystemEventListener` instances for each type of `SystemEvent` defined by that class.

- `ClientBehaviorHolder`: Adds the ability to attach `ClientBehavior` instances such as a reusable script.

`UICommand` implements `ActionSource2` and `StateHolder`.

`UIOutput` and component classes that extend `UIOutput` implement `StateHolder` and `ValueHolder`.

`UIInput` and component classes that extend `UIInput` implement `EditableValueHolder`, `StateHolder`, and `ValueHolder`.

`UIComponentBase` implements `StateHolder`.

Only component writers will need to use the component classes and behavioral interfaces directly.

Page authors and application developers will use a standard component by including a tag that represents it on a page.

Most of the components can be rendered in different ways on a page.

For example, a `UICommand` component can be rendered as a button or a hyperlink.

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

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 class.

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 combination of component and renderer.

A render kit defines how component classes map to component tags that are appropriate for a particular client.

The JavaServer Faces implementation includes a standard HTML render kit for rendering to an HTML client.

The render kit defines a set of `Renderer` classes for each component that it supports.

Each `Renderer` class defines a different way to render the particular component to the output defined by the render kit.

For example, a `UISelectOne` component has three different renderers.

One of them renders the component as a set of radio buttons.

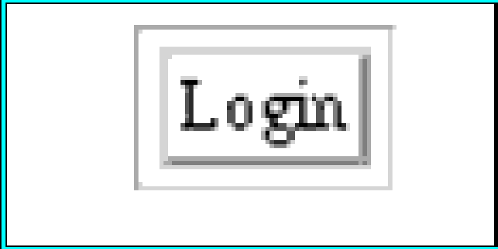
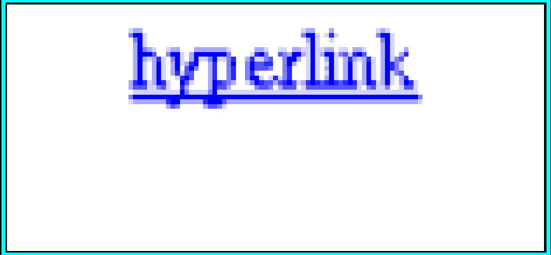
Another renders the component as a combo box.

The third one renders the component as a list box.

Each custom tag defined in the standard HTML render kit is composed of the component functionality (defined in the `UIComponent` class) and the rendering attributes (defined by the `Renderer` class).

For example, the two tags in Table 10-1 represent a `UICommand` component rendered in two different ways.

Table 10-1 `UICommand` Tags

Tag	Rendered As
`commandButton`	
`commandLink`	

The command part of the tags shown in Table 10-1 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` class, which defines how the component appears on the page.

The JavaServer Faces implementation provides a custom tag library for rendering components in HTML. The section Adding Components to a Page Using HTML Tags lists all supported component tags and describes how to use the tags in an example.

Conversion Model

A JavaServer Faces application can optionally associate a component with server-side object data.

This object is a JavaBeans component, such as a backing bean.

An application gets and sets the object data for a component by calling the appropriate object properties for that component.

When a component is bound to an object, the application has two views of the component's data:

- The model view, in which data is represented as data types, such as `int` or `long`.

- 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 when the bean property associated with the component is of one of the types supported by the component's data.

For example, if a `UISelectBoolean` component is associated with a bean property of type `java.lang.Boolean`, the JavaServer Faces implementation will automatically convert the component's data from `String` to `Boolean`.

In addition, some component data must be bound to properties of a particular type.

For example, a `UISelectBoolean` component must be bound to a property of type `boolean` or `java.lang.Boolean`.

Sometimes you might want to convert a component’s data to a type other than a standard type, 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 `UIOutput` components and components whose classes subclass `UIOutput`.

If you register the `Converter` implementation on a component, the `Converter` implementation converts the component's data between the two views.

You can either use the standard converters supplied with the JavaServer Faces implementation or create your own custom converter.

Custom converter creation is covered in Chapter 14, Creating Custom UI Components.

Event and Listener Model

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 that an application can use to handle events generated by components.

The JavaServer Faces specification defines three types of events: application events, system events and data-model events.

Application events are tied to a particular application and are generated by a `UIComponent`.

They represent the standard events available in previous versions of JavaServer Faces technology.

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 must register it on the component that generates the event.

When the user activates a component, such as by 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 application events: action events and value-change events.

An action event (class `ActionEvent`) occurs when the user activates a component that implements `ActionSource`.

These components include buttons and hyperlinks.

A value-change event

(class `ValueChangeEvent`) occurs when the user changes the value of a component represented by `UIInput` or one of its subclasses.

An example is selecting a check box, an action that results in the component's value changing to `true`.

The component types that can generate these types of events are the `UIInput`, `UISelectOne`, `UISelectMany`, and `UISelectBoolean` components.

Value-change events are fired only if no validation errors were detected.

Depending on the value of the `immediate` property (see The `immediate` Attribute) of the component emitting the event, action events can be processed during the invoke application phase or the apply request values phase, and value-change events can be processed during the process validations phase or the apply request values phase.

System events are generated by an `Object` rather than a `UIComponent`.

They are generated during the execution of an application at predefined times.

They are applicable to the entire application rather than to a specific component.

A data-model event occurs when a new row of a `UIData` component is selected.

There are two ways to cause your application to react to action events or value-change events that are emitted by a standard component:

- Implement an event listener class to handle the event and register the listener on the component by nesting either a `valueChangeListener` tag or an `actionListener` tag inside the component tag.

- Implement a method of a backing bean to handle the event and refer to the method with a method expression from the appropriate attribute of the component's tag.

See Implementing an Event Listener for information on how to implement an event listener.

See Registering Listeners on Components for information on how to register the listener on a component.

See Writing a Method to Handle an Action Event and Writing a Method to Handle a Value-Change Event for information on how to implement backing bean methods that handle these events.

See Referencing a Backing Bean Method for information on how to refer to the backing bean method from the component tag.

When emitting events from custom components, you must implement the appropriate `Event` class and manually queue the event on the component in addition to implementing an event listener class or a backing bean method that handles the event.

Handling Events for Custom Components explains how to do this.

Validation Model

JavaServer Faces technology supports a mechanism for validating the local data of editable components (such as text fields).

This validation occurs before the corresponding model data is updated to match the local value.

Like the conversion model, the validation model defines a set of standard classes for performing common data validation checks.

The JavaServer Faces core tag library also defines a set of tags that correspond to the standard `Validator` implementations.

See Using the Standard Validators for a list of all the standard validation classes and corresponding tags.

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.

In addition to validators that are registered on the component, you can declare a default validator which is registered on all `UIInput` components in the application.

For more information on default validators, see Using Default Validators.

The validation model also allows you to create your own custom validator and corresponding tag to perform custom validation.

The validation model provides two ways to implement custom validation:

- Implement a `Validator` interface that performs the validation.

- Implement a backing bean method that performs the validation.

If you are implementing a `Validator` interface, you must also:

- Register the `Validator` implementation with the application.

- Create a custom tag or use a `validator` tag to register the validator on the component.

In the previously described standard validation model, the validator is defined for each input component on a page.

The Bean Validation model allows the validator to be applied to all fields in a page.

See Using Bean Validation and Chapter 47, Advanced Bean Validation Concepts and Examples for more information on Bean Validation.

Navigation Model

The JavaServer Faces navigation model makes it easy to define page navigation and to handle any additional processing that is needed to choose the sequence in which pages are loaded.

In JavaServer Faces technology, navigation is a set of rules for choosing the next page or view to be displayed after an application action, such as when a button or hyperlink is clicked.

Navigation can be implicit or user-defined.

Implicit navigation comes into play when user-defined navigation rules are not available.

For more information on implicit navigation, see Implicit Navigation Rules.

User-defined navigation rules are declared in zero or more application configuration resource files such as `faces-config.xml`, by using a set of XML elements.

The default structure of a navigation rule is as follows:

```
<navigation-rule>
<description></description
<from-view-id></from-view-id>
<navigation-case>
<from-action></from-action>
<from-outcome></from-outcome>
<if></if>
<to-view-id></to-view-id>
</navigation-case>
</navigation-rule>
```

User-defined navigation is handled as follows::

- Define the rules in the application configuration resource file.

- Refer to an outcome `String` from the button or hyperlink component's `action` attribute.

This outcome `String` is used by the JavaServer Faces implementation to select the navigation rule.

Here is an example navigation rule:

```
<navigation-rule>
<from-view-id>
/greeting.xhtml
</from-view-id>
```

```
<navigation-case>
<from-outcome>
success
</from-outcome>
<to-view-id>
/response.xhtml
</to-view-id>
</navigation-case>
</navigation-rule>
```

This rule states that when a command component (such as a `commandButton` or a `commandLink`) on `greeting.xhtml` is activated, the application will navigate from the `greeting.xhtml` page to the `response.xhtml` page if the outcome referenced by the button component's tag is `success`.

Here is the `commandButton` tag from `greeting.xhtml` that specifies a logical outcome of success:

```
<h:commandButton id="submit"
action="success"
value="Submit"
/>
```

As the example demonstrates, each `navigation-rule` element defines how to get from one page (specified in the `from-view-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-view-id`) based on a logical outcome (defined by `from-outcome`) .

In more complicated applications, the logical outcome can also come from the return value of an action method in a backing bean.

This method performs some processing to determine the outcome.

For example, the method can check whether the password the user entered on the page matches the one on file.

If it does, the method might return `success`; otherwise, it might return `failure`.

An outcome of `failure` might result in the logon page being reloaded.

An outcome of `success` might cause the page displaying the user's credit card activity to open.

If you want the outcome to be returned by a method on a bean, you must refer to the method using a method expression, with the `action` attribute, as shown by this example:

```
<h:commandButton id="submit"
action=
"#{userNumberBean.getOrderStatus}"
value="Submit"
/>
```

When the user clicks the button represented by this tag, the corresponding component generates an action event.

This event is handled by the default `ActionListener` instance, which calls the action method referenced by the component that triggered the event.

The action method returns a logical outcome to the action listener.

The listener passes the logical outcome and a reference to the action method that produced the outcome to the default `NavigationHandler`.

The `NavigationHandler` selects the page to display next by matching the outcome or the action method reference against the navigation rules in the application configuration resource file by the following process:

1. The `NavigationHandler` selects the navigation rule that matches the page currently displayed.

2. It matches the outcome or the action method reference that it received from the default `ActionListener` with those defined by the navigation cases.

3. It tries to match both the method reference and the outcome against the same navigation case.

4. If the previous step fails, the navigation handler attempts to match the outcome.

5. Finally, the navigation handler attempts to match the action method reference if the previous two attempts failed.

6. If no navigation case is matched, it displays the same view again.

When the `NavigationHandler` achieves a match, the render response phase begins.

During this phase, the page selected by the `NavigationHandler` will be rendered.

For more information on how to define navigation rules, see Configuring Navigation Rules.

For more information on how to implement action methods to handle navigation, see Writing a Method to Handle an Action Event.

For more information on how to reference outcomes or action methods from component tags, see Referencing a Method That Performs Navigation.