Using Converters, Listeners, and Validators

The previous chapter described components and explained how to add them to a web page.

This chapter provides information on adding more functionality to the components through converters, listeners, and validators.

- Converters are used to convert data that is received from the input components.
- Listeners are used to listen to the events happening in the page and perform actions as defined.
- Validators are used to validate the data that is received from the input components.

The following topics are addressed here:

- Using the Standard Converters
- Registering Listeners on Components
- Using the Standard Validators
- Referencing a Backing Bean Method

Using the Standard Converters

The JavaServer Faces implementation provides a set of `Converter` implementations that you can use to convert component data.

The standard `Converter` implementations, located in the `javax.faces.convert` package, are as follows:

- BigDecimalConverter
- BigIntegerConverter
- BooleanConverter
- ByteConverter
- CharacterConverter
- DateTimeConverter
- DoubleConverter
- EnumConverter
- FloatConverter
- IntegerConverter
- LongConverter

- `NumberConverter`
- `ShortConverter`

A standard error message is associated with each of these converters.

If you have registered one of these converters onto a component on your page, and the converter is not able to convert the component's value, the converter's error message will display on the page.

For example, the following error message appears if `BigIntegerConverter` fails to convert a value:

`{0} must be a number consisting of one or more digits`

In this case, the `{0}` substitution parameter will be replaced with the name of the input component on which the converter is registered.

Two of the standard converters (`DateTimeConverter` and `NumberConverter`) have their own tags, which allow you to configure the format of the component data using the tag attributes.

For more information about using `DateTimeConverter`, see Using `DateTimeConverter`.

For more information about using `NumberConverter`, see Using `NumberConverter`.

The following section explains how to convert a component's value, including how to register other standard converters with a component.

Converting a Component's Value

To use a particular converter to convert a component's value, you need to register the converter onto the component.

You can register any of the standard converters in one of the following ways:

- Nest one of the standard converter tags inside the component's tag.

These tags are `convertDateTime` and `convertNumber`, which are described in Using `DateTimeConverter` and Using `NumberConverter`, respectively.

- Bind the value of the component to a backing bean property of the same type as the converter.
- Refer to the converter from the component tag's `converter` attribute.
- Nest a `converter` tag inside of the component tag, and use either the `converter` tag's `converterId` attribute or its `binding` attribute to refer to the converter.

As an example of the second method, if you want a component's data to be converted to an Integer, you can simply bind the component's value to a backing bean property.

Here is an example:

```
Integer age = 0;
public Integer getAge()
{ return age;}
```

```
public void setAge(Integer age)
{this.age = age;}
```

If the component is not bound to a bean property, you can use the third method by using the `converter` attribute directly on the component tag:

```
<h:inputText
converter=
"javax.faces.convert.IntegerConverter"
/>
```

This example shows the `converter` attribute referring to the fully qualified class name of the converter.

The `converter` attribute can also take the ID of the component.

The data from the `inputText` tag in the this example will be converted to a `java.lang.Integer` value.

The `Integer` type is a supported type of `NumberConverter`.

If you don't need to specify any formatting instructions using the `convertNumber` tag attributes, and if one of the standard converters will suffice, you can simply reference that converter by using the component tag's `converter` attribute.

Finally, you can nest a `converter` tag within the component tag and use either the converter tag's `converterId` attribute or its `binding` attribute to reference the converter.

The `converterId` attribute must reference the converter's ID.

Here is an example:

```
<h:inputText
value="#{LoginBean.Age}"
/>
<f:converter
converterId="Integer"
/>
</h:inputText>
```

Instead of using the `converterId` attribute, the `converter` tag can use the `binding` attribute.

The `binding` attribute must resolve to a bean property that accepts and returns an appropriate `Converter` instance.

Using `DateTimeConverter`

You can convert a component's data to a `java.util.Date` by nesting the `convertDateTime` tag inside the component tag.

The `convertDateTime` tag has several attributes that allow you to specify the format and type of the data.

Table 8-1 lists the attributes.

Here is a simple example of a `convertDateTime` tag:

```
<h:outputText id= "shipDate"
value="#{cashier.shipDate}"
>
<f:convertDateTime
dateStyle="full"
/>
</h:outputText>
```

When binding the `DateTimeConverter` to a component, ensure that the backing bean property to which the component is bound is of type `java.util.Date`.

In the preceding example, `cashier.shipDate` must be of type `java.util.Date`.

The example tag can display the following output:

```
Saturday, September 25, 2010
```

You can also display the same date and time by using the following tag where the date format is specified:

```
<h:outputText
value="#{cashier.shipDate}">
```

```
<f:convertDateTime
pattern="EEEEEEEE, MMM dd, yyyy"
/>
</h:outputText>
```

If you want to display the example date in Spanish, you can use the `locale` attribute:

```
<h:inputText
value="#{cashier.shipDate}">
```

```
<f:convertDateTime
dateStyle="full"
locale="Locale.SPAIN"
timeStyle="long" type="both"
/>
</h:inputText>
```

This tag would display the following output:

```
sabado 25 de septiembre de 2010
```

Refer to the "Customizing Formats" lesson of the *Java Tutorial* at http://download.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html for more information on how to format the output using the `pattern` attribute of the `convertDateTime` tag.

Table 8-1 Attributes for the `convertDateTime` Tag

Attribute	Type	Description
`binding`	`DateTimeConverter`	Used to bind a converter to a backing bean property.
`dateStyle`	`String`	Defines the format, as specified by `java.text.DateFormat`, of a date or the date part of a `date` string. Applied only if `type` is `date` or `both` and if `pattern` is not defined. Valid values: `default`, `short`, `medium`, `long`, and `full`. If no value is specified, `default` is used.

`for`	`String`	Used with composite components. Refers to one of the objects within the composite component inside which this tag is nested.
`locale`	`String` or `Locale`	`Locale` whose predefined styles for dates and times are used during formatting or parsing. If not specified, the `Locale` returned by `FacesContext.getLocale` will be used.

`pattern`	`String`	Custom formatting pattern that determines how the date/time string should be formatted and parsed. If this attribute is specified, `dateStyle`, `timeStyle`, and `type` attributes are ignored.

`timeStyle`	`String`	Defines the format, as specified by `java.text.DateFormat`, of a `time` or the time part of a `date` string. Applied only if `type` is time and `pattern` is not defined. Valid values: `default`, `short`, `medium`, `long`, and `full`. If no value is specified, `default` is used.

timeZone	String or TimeZone	Time zone in which to interpret any time information in the date string.
type	String	Specifies whether the string value will contain a date, a time, or both. Valid values are date, time, or both. If no value is specified, date is used.

Using `NumberConverter`

You can convert a component's data to a `java.lang.Number` by nesting the `convertNumber` tag inside the component tag.

The `convertNumber` tag has several attributes that allow you to specify the format and type of the data.

Table 8-2 lists the attributes.

The following example uses a `convertNumber` tag to display the total prices of the contents of a shopping cart:

```
<h:outputText
value="#{cart.total}" >
 <f:convertNumber type="currency"/>
</h:outputText>
```

When binding the `NumberConverter` to a component, ensure that the backing bean property to which the component is bound is of a primitive type or has a type of `java.lang.Number`.

In the preceding example, `cart.total` is of type `java.lang.Number`.

Here is an example of a number that this tag can display:

$934

This result can also be displayed by using the following tag, where the currency pattern is specified:

```
<h:outputText id="cartTotal"
value="#{cart.Total}"
>
```

```
<f:convertNumber pattern="$####" />
</h:outputText>
```

See the “Customizing Formats” lesson of the *Java Tutorial* at http://download.oracle.com/javase/tutorial/i18n/format/decimalFormat.html for more information on how to format the output by using the `pattern` attribute of the `convertNumber` tag.

Table 8-2 Attributes for the `convertNumber` Tag

Attribute	Type	Description
`binding`	`NumberConverter`	Used to bind a converter to a backing bean property.
`currencyCode`	`String`	ISO 4217 currency code, used only when formatting currencies.
`currencySymbol`	`String`	Currency symbol, applied only when formatting currencies.

`for`	`String`	Used with composite components. Refers to one of the objects within the composite component inside which this tag is nested.
`groupingUsed`	`Boolean`	Specifies whether formatted output contains grouping separators.
`integerOnly`	`Boolean`	Specifies whether only the integer part of the value will be parsed.

`locale`	`String` or `Locale`	`Locale` whose number styles are used to format or parse data.
`maxFractionDigits`	`int`	Maximum number of digits formatted in the fractional part of the output.
`maxIntegerDigits`	`int`	Maximum number of digits formatted in the integer part of the output.
`minFractionDigits`	`int`	Minimum number of digits formatted in the fractional part of the output.

`minIntegerDigits`	`int`	Minimum number of digits formatted in the integer part of the output.
`pattern`	`String`	Custom formatting pattern that determines how the number string is formatted and parsed.

`type`	`String`	Specifies whether the string value is parsed and formatted as a `number`, `currency`, or `percentage`. If not specified, `number` is used.

Registering Listeners on Components

An application developer can implement listeners as classes or as backing bean methods.

If a listener is a backing bean method, the page author references the method from either the component's `valueChangeListener` attribute or its `actionListener` attribute.

If the listener is a class, the page author can reference the listener from either a `valueChangeListener` tag or an `actionListener` tag and nest the tag inside the component tag to register the listener on the component.

Referencing a Method That Handles an Action Event and Referencing a Method That Handles a Value-Change Event explain how a page author uses the `valueChangeListener` and `actionListener` attributes to reference backing bean methods that handle events.

This section explains how to register the `NameChanged` value-change listener and a hypothetical `LocaleChange` action listener implementation on components.

Registering a Value-Change Listener on a Component

A `ValueChangeListener` implementation can be registered on a component that implements `EditableValueHolder` by nesting a `valueChangeListener` tag within the component's tag on the page.

The `valueChangeListener` tag supports the attributes shown in Table 8-3, one of which must be used.

Table 8-3 Attributes for the `valueChangeListener` Tag

Attribute	Description
`type`	References the fully qualified class name of a `ValueChangeListener` implementation. Can accept a literal or a value expression.
`binding`	References an object that implements `ValueChangeListener`. Can accept only a value expression, which must point to a backing bean property that accepts and returns a `ValueChangeListener` implementation.

The following example shows a value-change listener registered on a component:

```
<h:inputText id="name" size="50"
value="#{cashier.name}"
required="true"
>
<f:valueChangeListener
type="listeners.NameChanged"
/>
</h:inputText>
```

In the example, the core tag `type` attribute specifies the custom `NameChanged` listener as the `ValueChangeListener` implementation registered on the `name` component.

After this component tag is processed and local values have been validated, its corresponding component instance will queue the `ValueChangeEvent` associated with the specified `ValueChangeListener` to the component.

The `binding` attribute is used to bind a `ValueChangeListener` implementation to a backing bean property.

This attribute works in a similar way to the `binding` attribute supported by the standard converter tags.

Registering an Action Listener on a Component

A page author can register an `ActionListener` implementation on a command component by nesting an `actionListener` tag within the component's tag on the page.

Similarly to the `valueChangeListener` tag, the `actionListener` tag supports both the `type` and `binding` attributes.

One of these attributes must be used to reference the action listener.

Here is an example of a `commandLink` tag that references an `ActionListener` implementation rather than a backing bean method:

```
<h:commandLink id="NAmerica"
action="bookstore">
<f:actionListener
type="listeners.LocaleChange" />
</h:commandLink>
```

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

Similarly to the `valueChangeListener` tag, the `actionListener` tag also supports the `binding` attribute.

Using the Standard Validators

JavaServer Faces technology provides a set of standard classes and associated tags that page authors and application developers can use to validate a component's data.

Table 8-4 lists all the standard validator classes and the tags that allow you to use the validators from the page.

Table 8-4 The Validator Classes

Validator Class	Tag	Function
`BeanValidator`	`validateBean`	Registers a bean validator for the component.
`DoubleRange Validator`	`validateDouble Range`	Checks whether the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point.

`LengthValidator`	`validateLength`	Checks whether the length of a component's local value is within a certain range. The value must be a `java.lang.String`.
`LongRange Validator`	`validateLong Range`	Checks whether the local value of a component is within a certain range. The value must be any numeric type or `String` that can be converted to a `long`.

`RegexValidator`	`validateRegEx`	Checks whether the local value of a component is a match against a regular expression from the `java.util.regex` package.
`RequiredValidator`	`validateRequired`	Ensures that the local value is not empty on an `EditableValueHolder` component.

Similar to the standard converters, each of these validators has one or more standard error messages associated with it.

If you have registered one of these validators onto a component on your page, and the validator is unable to validate the component’s value, the validator’s error message will display on the page.

For example, the error message that displays when the component's value exceeds the maximum value allowed by `LongRangeValidator` is as follows:

```
{1}: Validation Error: Value is
greater than allowable maximum of
"{0}"
```

In this case, the `{1}` substitution parameter is replaced by the component's label or `id`, and the `{0}` substitution parameter is replaced with the maximum value allowed by the validator.

Instead of using the standard validators, you can use Bean Validation to validate data.

See Using Bean Validation for more information.

Validating a Component's Value

To validate a component's value using a particular validator, you need to register that validator on the component.

You can do this in one of the following ways:

. Nest the validator's corresponding tag (shown in Table 8-4) inside the component's tag.

Using `LongRangeValidator` explains how to use the `validateLongRange` tag.

You can use the other standard tags in the same way.

- Refer to a method that performs the validation from the component tag's `validator` attribute.

- Nest a `validator` tag inside the component tag, and use either the validator tag's `validatorId` attribute or its `binding` attribute to refer to the validator.

See Referencing a Method That Performs Validation for more information on using the `validator` attribute.

The `validatorId` attribute works similarly to the `converterId` attribute of the `converter` tag, as described in Converting a Component's Value.

Keep in mind that validation can be performed only on components that implement `EditableValueHolder`, because these components accept values that can be validated.

Using `LongRangeValidator`

The following example shows how to use the `validateLongRange` validator on an input component named `quantity`:

```
<h:inputText id="quantity" size="4"
value="#{item.quantity}" >
<f:validateLongRange minimum="1"/>
```

```
</h:inputText>
<h:message for="quantity"/>
```

This tag requires the user to enter a number that is at least 1.

The `size` attribute specifies that the number can have no more than four digits.

The `validateLongRange` tag also has a `maximum` attribute, which sets a maximum value for the input.

The attributes of all the standard validator tags accept EL value expressions.

This means that the attributes can reference backing bean properties rather than specify literal values.

For example, the `validateLongRange` tag in the preceding example can reference a backing bean property called `minimum` to get the minimum value acceptable to the validator implementation, as shown here:

```
<f:validateLongRange
minimum="#{ShowCartBean.minimum}"
/>
```

Referencing a Backing Bean Method

A component tag has a set of attributes for referencing backing bean methods that can perform certain functions for the component associated with the tag.

These attributes are summarized in Table 8-5.

Table 8-5 Component Tag Attributes That Reference Backing Bean Methods

Attribute	Function
`action`	Refers to a backing bean method that performs navigation processing for the component and returns a logical outcome `String`
`actionListener`	Refers to a backing bean method that handles action events
`validator`	Refers to a backing bean method that performs validation on the component's value
`valueChange Listener`	Refers to a backing bean method that handles value-change events

Only components that implement `ActionSource` can use the `action` and `actionListener` attributes.

Only components that implement `EditableValueHolder` can use the `validator` or `valueChangeListener` attributes.

The component tag refers to a backing bean method using a method expression as a value of one of the attributes.

The method referenced by an attribute must follow a particular signature, which is defined by the tag attribute's definition in the documentation at http://download.oracle.com/javaee/6/javaserverfaces/2.1/docs/vdldocs/jsp/.

For example, the definition of the `validator` attribute of the `inputText` tag is the following:

```
void validate
(javax.faces.context.FacesContext,
javax.faces.component.UIComponent,
java.lang.Object)
```

The following sections give examples of how to use the attributes.

Referencing a Method That Performs Navigation

If your page includes a component, such as a button or a hyperlink, that causes the application to navigate to another page when the component is activated, the tag corresponding to this component must include an `action` attribute.

This attribute does one of the following:

- Specifies a logical outcome `String` that tells the application which page to access next
- References a backing bean method that performs some processing and returns a logical outcome `String`

The following example shows how to reference a navigation method:

```
<h:commandButton
value="#{bundle.Submit}"
action="#{cashier.submit}"
/>
```

Referencing a Method That Handles an Action Event

If a component on your page generates an action event, and if that event is handled by a backing bean method, you refer to the method by using the component's `actionListener` attribute.

The following example shows how the method is referenced:

```
<h:commandLink id="NAmerica"
action="bookstore"
actionListener=
"#{localeBean.chooseLocaleFromLink}"
>
```

The `actionListener` attribute of this component tag references the `chooseLocaleFromLink` method using a method expression.

The `chooseLocaleFromLink` method handles the event when the user clicks the hyperlink rendered by this component.

Referencing a Method That Performs Validation

If the input of one of the components on your page is validated by a backing bean method, refer to the method from the component's tag by using the `validator` attribute.

The following example shows how to reference a method that performs validation on `email`, an input component:

```
<h:inputText id="email"
value="#{checkoutFormBean.email}"
size="25" maxlength="125"
validator=
"#{checkoutFormBean.validateEmail}"
/>
```

Referencing a Method That Handles a Value-Change Event

If you want a component on your page to generate a value-change event and you want that event to be handled by a backing bean method, you refer to the method by using the component's `valueChangeListener` attribute.

The following example shows how a component references a `ValueChangeListener` implementation that handles the event when a user enters a name in the `name` input field:

```
<h:inputText id="name"
size="50" value="#{cashier.name}"
required="true"
>
<f:valueChangeListener
type="listeners.NameChanged"
/>
</h:inputText>
```

To refer to this backing bean method, the tag uses the `valueChangeListener` attribute:

```
<h:inputText id="name"
size="50"value="#{cashier.name}"
required="true"
valueChangeListener=
"#{cashier.processValueChange}"
/>
</h:inputText>
```

The `valueChangeListener` attribute of this component tag references the `processValueChange` method of `CashierBean` by using a method expression.

The `processValueChange` method handles the event of a user entering a name in the input field rendered by this component.