CONVERSION AND VALIDATION

Topics in This Chapter

- “Overview of the Conversion and Validation Process” on page 219
- “Using Standard Converters” on page 221
- “Using Standard Validators” on page 233
- “Programming with Custom Converters and Validators” on page 240
In this chapter, we discuss how form data is converted to Java objects and how the conversion results are checked for correctness. The JSF container carries out these steps before updating the model, so you can rest assured that invalid inputs will never end up in the business logic.

We first look at the concepts behind the conversion and validation process. Then we discuss the standard tags that JSF provides for conversion and validation. These tags suffice for the most common needs. Next, you see how to supply your own conversion and validation code for more complex scenarios.

It is also possible to implement custom tags—reusable converters and validators that can be configured by page authors. However, implementing custom tags requires significantly more programming. The details are covered in “Implementing Custom Converters and Validators” on page 432 of Chapter 9.

**Overview of the Conversion and Validation Process**

Now we look at user input in slow motion as it travels from the browser form to the beans that make up the business logic.

First, the user fills in a field of a web form. When the user clicks the submit button, the browser sends the form data to the server. We call this value the *request value*.

In the Apply Request Values phase, the request values are stored in component objects. (Recall that each input tag of the JSF page has a corresponding component object.) The value stored in the component object is called the *submitted value*. 
Of course, all request values are *strings*—after all, the client browser sends the strings that the user supplies. On the other hand, the web application deals with arbitrary types, such as `int`, `date`, or even more sophisticated types. A *conversion* process transforms the incoming strings to those types. In the next section, we discuss conversion in detail.

The converted values are not immediately transmitted to the beans that make up the business logic. Instead, they are first stored inside the component objects as *local values*. After conversion, the local values are *validated*. Page designers can specify validation conditions—for example, that certain fields should have a minimum or maximum length. We begin our discussion of validation under “Using Standard Validators” on page 233. After all local values have been validated, the Update Model Values phase starts, and the local values are stored in beans, as specified by their value references.

You may wonder why JSF bothers with local values at all. Could not one simply store the request values directly in the model?

JSF uses a two-step approach to make it easier to preserve model integrity. As all programmers know only too well, users enter wrong information with distressing regularity. Suppose some of the model values had been updated before the first user error was detected. The model might then be in an inconsistent state, and it would be tedious to bring it back to its old state.

For that reason, JSF first converts and validates all user input. If errors are found, the page is redisplayed with the values that the user entered so that the user can try again. The Update Model Values phase starts only if all validations are successful.

Figure 6–1 shows the journey of a field value from the browser to the server-side component object and finally to the model bean.
Using Standard Converters

In the following sections, we cover the converters and validators that are part of the JSF library. Later in this chapter, you learn how to supply your own validation code if your needs go beyond the basics.

Conversion of Numbers and Dates

A web application stores data of many types, but the web user interface deals exclusively with strings. For example, suppose the user needs to edit a `Date` object that is stored in the business logic. First, the `Date` object is converted to a string that is sent to the client browser to be displayed inside a text field. The user then edits the text field. The resulting string is returned to the server and must be converted back to a `Date` object.

The same situation holds, of course, for primitive types such as `int`, `double`, or `boolean`. The user of the web application edits strings, and the JSF container needs to convert the string to the type required by the application.

To see a typical use of a built-in converter, imagine a web application that is used to process payments (see Figure 6–2). The payment data includes

- The amount to be charged
- The credit card number
- The credit card expiration date

![An Application to Test Data Conversion - Mozilla Firefox](http://localhost:8090/convert/index.faces)

Please enter the payment information

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount</td>
<td>0.00</td>
</tr>
<tr>
<td>Credit Card</td>
<td></td>
</tr>
<tr>
<td>Expiration date (Month/Year)</td>
<td>02/2005</td>
</tr>
</tbody>
</table>

**Figure 6–2 Processing payments**
We attach a converter to the text field and tell it to format the current value with at least two digits after the decimal point:

```xml
<h:inputText value="#{payment.amount}"
  <f:convertNumber minFractionDigits="2"/>
</h:inputText>
```

The `f:convertNumber` converter is one of the standard converters supplied by the JSF implementation.

The second field in this screen does not use a converter. (Later in this chapter, we attach a custom converter.) The third field uses an `f:convertDateTime` converter whose `pattern` attribute is set to the string `MM/yyyy`. (The pattern string format is documented in the API documentation for the `java.text.SimpleDateFormat` class.)

```xml
<h:inputText value="#{payment.date}"
  <f:convertDateTime pattern="MM/yyyy"/>
</h:inputText>
```

In the `result.jsp` page, we show the inputs that the user provided, using a different converter for the payment amount:

```xml
<h:outputText value="#{payment.amount}"
  <f:convertNumber type="currency"/>
</h:outputText>
```

This converter automatically supplies a currency symbol and decimal separators (see Figure 6–3).

![An Application to Test Data Conversion - Mozilla Firefox](image)

**Payment Information**

- **Amount**: $10,000.00
- **Credit Card**: 4111111111111111
- **Expiration date (Month/Year)**: 04/2006

*Figure 6–3 Displaying the payment information*
**Converters and Attributes**

Tables 6–1 and 6–2 show the standard converters and their attributes.

NOTE: If you use a value expression whose type is either a primitive type, or, starting with JSF 1.2, an enumerated type or `BigInteger/BigDecimal`, then you do not need to specify any converter. The JSF implementation automatically picks a standard converter. However, you need to specify an explicit converter for `Date` values.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>String</td>
<td>number (default), currency, or percent</td>
</tr>
<tr>
<td>pattern</td>
<td>String</td>
<td>Formatting pattern, as defined in <code>java.text.DecimalFormat</code></td>
</tr>
<tr>
<td>maxFractionDigits</td>
<td>int</td>
<td>Maximum number of digits in the fractional part</td>
</tr>
<tr>
<td>minFractionDigits</td>
<td>int</td>
<td>Minimum number of digits in the fractional part</td>
</tr>
<tr>
<td>maxIntegerDigits</td>
<td>int</td>
<td>Maximum number of digits in the integer part</td>
</tr>
<tr>
<td>minIntegerDigits</td>
<td>int</td>
<td>Minimum number of digits in the integer part</td>
</tr>
<tr>
<td>integerOnly</td>
<td>boolean</td>
<td>True if only the integer part is parsed (default: false)</td>
</tr>
<tr>
<td>groupingUsed</td>
<td>boolean</td>
<td>True if grouping separators are used (default: true)</td>
</tr>
<tr>
<td>locale</td>
<td>java.util.Locale</td>
<td>Locale whose preferences are to be used for parsing and formatting</td>
</tr>
<tr>
<td>currencyCode</td>
<td>String</td>
<td>ISO 4217 currency code to use when converting currency values</td>
</tr>
<tr>
<td>currencySymbol</td>
<td>String</td>
<td>Currency symbol to use when converting currency values</td>
</tr>
</tbody>
</table>
The **converter** Attribute

An alternate syntax for attaching a converter to a component is to add the converter attribute to the component tag. You specify the ID of the converter like this:

```html
<h:outputText value="#{payment.date}" converter="javax.faces.DateTime"/>
```

This is equivalent to using `f:convertDateTime` with no attributes:

```html
<h:outputText value="#{payment.date}">
    <f:convertDateTime/>
</h:outputText>
```

A third way of specifying the converter would be as follows:

```html
<h:outputText value="#{payment.date}">
    <f:converter converterId="javax.faces.DateTime"/>
</h:outputText>
```

All JSF implementations must define a set of converters with predefined IDs:

- `javax.faces.DateTime` (used by `f:convertDateTime`)
- `javax.faces.Number` (used by `f:convertNumber`)
- `javax.faces.BigDecimal`, `javax.faces.BigInteger` (automatically used for `BigDecimal`/`BigInteger`)

Additional converter IDs can be configured in an application configuration file (see “Specifying Converters” on page 243 for details).
CAUTION: When the value of the converter attribute is a string, then the value indicates the ID of a converter. However, if it is a value expression, then its value must be a converter object—an object of a class that implements the Converter interface. That interface is introduced under “Programming with Custom Converters and Validators” on page 240.

NOTE: As of JSF 1.2, the f:convertNumber, f:convertDateTime, and f:converter tags have an optional binding attribute. This allows you to tie a converter instance to a backing bean property of type javax.faces.convert.Converter.

Conversion Errors
When a conversion error occurs, the following actions are the result:

- The component whose conversion failed posts a message and declares itself invalid. (You will see in the following sections how to display the message.)
- The JSF implementation redisplays the current page immediately after the Process Validations phase has completed. The redisplayed page contains all values that the user provided—no user input is lost.

This behavior is generally desirable. If a user provides an illegal input for, say, a field that requires an integer, then the web application should not try to use that illegal input. The JSF implementation automatically redisplays the current page, giving the user another chance to enter the value correctly.

However, you should avoid overly restrictive conversion options for input fields. For example, consider the “Amount” field in our example. Had we used a currency format, then the current value would have been nicely formatted. But suppose a user enters 100 (without a leading $ sign). The currency formatter will complain that the input is not a legal currency value. That is too strict for human use.

To overcome this problem, you can program a custom converter. A custom converter can format a value prettily, yet be lenient when interpreting human input. Custom converters are described later in this chapter under “Programming with Custom Converters and Validators” on page 240.
TIP: When gathering input from the user, you should either use a lenient converter or redesign your form to be more user friendly. For example, rather than forcing users to format the expiration date as \text{MM/yyyy}, you can supply two input fields, one for the month and another for the year.

**Displaying Error Messages**

Of course, it is important that the user be able to see the messages that are caused by conversion and validation errors. You should add `h:message` tags whenever you use converters and validators.

Normally, you want to show the error messages next to the components that reported them (see Figure 6–4). Give an ID to the component and reference that ID in the `h:message` tag. As of JSF 1.2, you also need to supply a component label that is displayed in the error message.

```xml
<h:inputText id="amount" label="#{msgs.amount}" value="#{payment.amount}"/>
<h:message for="amount"/>
```

For JSF 1.1, omit the `label` attribute.

![Figure 6–4  Displaying a conversion error message](image)

The `h:message` tag takes a number of attributes to describe the appearance of the message (see “Messages” on page 157 of Chapter 4 for details). Here, we discuss only the attributes that are of particular interest for error reporting.

A message has two versions: `summary` and `detail`. 
For the number converter, the detail error message shows the label of the component, the offending value, and a sample of a correct value, like this:

Amount: ‘too much’ is not a number. Example: 99

The summary message omits the example.

NOTE: In JSF 1.1, the converters displayed a generic message “Conversion error occurred.”

By default, the \texttt{h:message} tag shows the detail and hides the summary. If you want to show the summary message instead, use these attributes:

\begin{verbatim}
<h:message for="amount" showSummary="true" showDetail="false"/>
\end{verbatim}

CAUTION: If you use a standard converter, display either the summary message or the detail message, but not both—the messages are nearly identical. You do not want your users to ponder an error message that reads “...is not a number ... is not a number. Example: 99”.

TIP: If you do not use an explicit \texttt{f:convertNumber} converter but instead rely on the standard converters for numeric types, use the summary message and not the detail message. The detail messages give \textit{far too much} detail. For example, the standard converter for double values has this detail message: “... must be a number between 4.9E-324 and 1.7976931348623157E308 Example: 1999999”.

Usually, you will want to show error messages in a different color. You use the \texttt{styleClass} or \texttt{style} attribute to change the appearance of the error message:

\begin{verbatim}
<h:messages styleClass="errorMessage"/>
\end{verbatim}

or

\begin{verbatim}
<h:message for="amount" style="color:red"/>
\end{verbatim}

We recommend that you use \texttt{styleClass} and a style sheet instead of a hardcoded style.

\textbf{Displaying All Error Messages}

It is uncommon to have multiple messages for one component, but it can happen. The \texttt{h:message} tag produces only the \textit{first} message. Unfortunately, you do not know whether the first message is the most useful one for the user. While
Chapter 6 ■ Conversion and Validation

no tag shows all messages for a particular component, you can show a listing of all messages from all components with the h:messages tag.

By default, the h:messages tag shows the message summary instead of the message detail. This behavior is opposite from that of the h:message tag.

For h:messages, you usually want to set the layout attribute to "table" so that the messages are lined up vertically. Otherwise they are concatenated.

```xml
<h:messages layout="table"/>
```

TIP: Whenever you create a message, make sure it ends with a period and a space, to ensure a neat appearance when messages are concatenated.

TIP: The h:messages tag is useful for debugging. Whenever your JSF application stalls at a particular page and is unwilling to move on, add a <h:messages/> tag to see if a failed conversion or validation is the culprit.

CAUTION: In JSF 1.1, the error messages did not include the message label. That made the h:messages tag far less useful because users were left wondering which of their inputs caused an error.

Using a Custom Error Message

Starting with JSF 1.2, you can provide a custom converter error message for a component. Set the converterMessage attribute of the component whose value is being converted. For example,

```xml
<h:inputText ... converterMessage="Not a valid number."/>
```

CAUTION: Unlike the message strings of the next section, these message attributes are taken literally. Placeholders such as {0} are not replaced.

Changing the Text of Standard Error Messages

Sometimes, you may want to change the standard conversion messages for your entire web application. Table 6–3 shows the most useful standard messages. Note that all detail message keys end in _detail. To save space, the table does not list separate summary and detail strings when the summary string is a substring of the detail string. Instead, the additional detail phrase is set in italics. In most
messages, \{0\} is the invalid value, \{1\} is a sample valid value, and \{2\} is the component label; however, for the Boolean converter, \{1\} is the component label.

To replace a standard message, set up a message bundle, as explained in Chapter 2. Add the replacement message, using the appropriate key from Table 6–3.

Suppose you do not want to fuss with input labels or example values when the \texttt{f:convertNumber} converter reports an error. Add the following definition to a message bundle:

\begin{verbatim}
javax.faces.converter.NumberConverter.NUMBER_detail='''{0}''' is not a number.
\end{verbatim}

Then set the base name of the bundle in a configuration file (such as \texttt{faces-config.xml}):

\begin{verbatim}
<faces-config>
  <application>
    <message-bundle>com.corejsf.messages</message-bundle>
  </application>
...</faces-config>
\end{verbatim}

You need only specify the messages that you want to override.

\begin{table}[h]
\centering
\caption{Standard Conversion Error Messages}
\begin{tabular}{|c|l|}
\hline
Resource ID & Default Text	\\
\hline
javax.faces.converter.IntegerConverter.INTEGER & \(2\): \curl{0}\curl{2} must be a number consisting of one or more digits.\\
\hline
javax.faces.converter.IntegerConverter.INTEGER_detail & \(2\): \curl{0}\curl{2} must be a number between -2147483648 and 2147483647. Example: \{1\}\\
\hline
javax.faces.converter.DoubleConverter.DOUBLE & \(2\): \curl{0}\curl{2} must be a number consisting of one or more digits.\\
\hline
javax.faces.converter.DoubleConverter.DOUBLE_detail & \(2\): \curl{0}\curl{2} must be a number between 4.9E-324 and 1.7976931348623157E308. Example: \{1\}\\
\hline
javax.faces.converter.BooleanConverter.BOOLEAN_detail & \(1\): \curl{0}\curl{1} must be 'true' or 'false'. Any value other than 'true' will evaluate to 'false'.\\
\hline
javax.faces.converter.NumberConverter.NUMBER_detail & \(2\): \curl{0}\curl{2} is not a number. Example: \{1\}\\
\hline
\end{tabular}
\end{table}
Table 6–3 Standard Conversion Error Messages (cont.)

<table>
<thead>
<tr>
<th>Resource ID</th>
<th>Default Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.faces.converter.NumberConverter.</td>
<td></td>
</tr>
<tr>
<td>CURRENCY_detail</td>
<td>[2]: &quot;[0]&quot; could not be understood as a currency value. Example: {1}</td>
</tr>
<tr>
<td>javax.faces.converter.NumberConverter.</td>
<td></td>
</tr>
<tr>
<td>PERCENT_detail</td>
<td>[2]: &quot;[0]&quot; could not be understood as a percentage. Example: {1}</td>
</tr>
<tr>
<td>javax.faces.converter.DateTimeConverter.</td>
<td></td>
</tr>
<tr>
<td>DATE_detail</td>
<td>[2]: &quot;[0]&quot; could not be understood as a date. Example: {1}</td>
</tr>
<tr>
<td>javax.faces.converter.DateTimeConverter.</td>
<td></td>
</tr>
<tr>
<td>TIME_detail</td>
<td>[2]: &quot;[0]&quot; could not be understood as a time. Example: {1}</td>
</tr>
<tr>
<td>javax.faces.converter.</td>
<td></td>
</tr>
<tr>
<td>DateTimeConverter.PATTERN_TYPE</td>
<td>[1]: A 'pattern' or 'type' attribute must be specified to convert the value &quot;{0}&quot;.</td>
</tr>
<tr>
<td>javax.faces.converter.EnumConverter.ENUM</td>
<td>[2]: &quot;[0]&quot; must be convertible to an enum.</td>
</tr>
<tr>
<td>ENUM_detail</td>
<td></td>
</tr>
<tr>
<td>javax.faces.converter.EnumConverter.ENUM</td>
<td>[2]: &quot;[0]&quot; must be convertible to an enum from the enum that contains the constant &quot;[1]&quot;.</td>
</tr>
<tr>
<td>ENUM_NO_CLASS</td>
<td>[1]: &quot;[0]&quot; must be convertible to an enum from the enum, but no enum class provided.</td>
</tr>
<tr>
<td>ENUM_NO_CLASS_detail</td>
<td>[1]: &quot;[0]&quot; must be convertible to an enum from the enum, but no enum class provided.</td>
</tr>
</tbody>
</table>

NOTE: In JSF 1.1, the generic message "Conversion error occurred" has key javax.faces.component.UIInput.Conversion.

A Complete Converter Example

We are now ready for our first complete example. Figure 6–5 shows the directory structure of the application. This web application asks the user to supply
payment information (Listing 6–1) and then displays the formatted information on a confirmation screen (Listing 6–2). The messages are in Listing 6–3 and the bean class is in Listing 6–4.

```
<html>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
<head>
<link href="styles.css" rel="stylesheet" type="text/css"/>
<title><h:outputText value="#{msgs.title}"/></title>
</head>
<body>
<h:form>
<h1><h:outputText value="#{msgs.enterPayment}"/></h1>
<h:panelGrid columns="3">
<h:outputText value="#{msgs.amount}"/>
<h:inputText id="amount" label="#{msgs.amount}" value="#{payment.amount}"
 f:convertNumber minFractionDigits="2"/>
<h:message for="amount" styleClass="errorMessage"/>
<h:outputText value="#{msgs.creditCard}"/>
<h:inputText id="card" label="#{msgs.creditCard}" value="#{payment.card}"/>
</h:panelGroup/>
</h:form>
```

**Figure 6–5  Directory structure of the converter sample**
Chapter 6 ■ Conversion and Validation

Listing 6–1  converter/web/index.jsp (cont.)

25. <h:outputText value="#{msgs.expirationDate}"/>
26. <h:inputText id="date" label="#{msgs.expirationDate}" value="#{payment.date}"
27.   <f:convertDateTime pattern="MM/yyyy"/>
28. </h:inputText>
29. <h:message for="date" styleClass="errorMessage"/>
30. </h:panelGrid>
31. <h:commandButton value="#{msgs.process}" action="process"/>
32. </h:form>
33. </f:view>
34. </html>

Listing 6–2  converter/web/result.jsp

1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4. <f:view>
5. <head>
6.   <link href="styles.css" rel="stylesheet" type="text/css"/>
7.   <title><h:outputText value="#{msgs.title}" /></title>
8. </head>
9. <body>
10. <h:form>
11.   <h1><h:outputText value="#{msgs.paymentInformation}" /></h1>
12.   <h:panelGrid columns="2">
13.     <h:outputText value="#{msgs.amount}" />
14.     <h:outputText value="#{payment.amount}" 
15.       <f:convertNumber type="currency"/>
16.     </h:outputText>
17.     <h:outputText value="#{msgs.creditCard}" />
18.     <h:outputText value="#{payment.card}" />
19.   </h:panelGrid>
20.   <h:outputText value="#{msgs.expirationDate}" />
21.   <h:outputText value="#{payment.date}" 
22.     <f:convertDateTime pattern="MM/yyyy"/>
23.   </h:outputText>
24. </h:panelGrid>
25.   <h:commandButton value="Back" action="back" />
26. </h:form>
27. </body>
28. </f:view>
29. </html>
Using Standard Validators

It is difficult to imagine a web application that should not perform a healthy dose of data validation. Since validation is so pervasive, you want it to be easy to use and extend. JSF fits the bill in both respects by providing a handful of standard validators and affording you a simple mechanism for implementing your own validators.

A key role of validation is to protect the model. Because JSF uses separate phases for processing validations and updating model values, you can be assured that the model is not put into an inconsistent state if some of the inputs cannot be validated.
Validating String Lengths and Numeric Ranges

It is easy to use JSF validators within JSF pages—add validator tags to the body of a component tag, like this:

```xml
<h:inputText id="card" value="#{payment.card}"
<f:validateLength minimum="13"/>
</h:inputText>
```

The preceding code fragment adds a validator to a text field; when the text field’s form is submitted, the validator makes sure that the string contains at least 13 characters. When validation fails (in this case, when the string has 12 or fewer characters), validators generate error messages associated with the guilty component. These messages can later be displayed in a JSF page by the `h:message` or `h:messages` tag.

NOTE: JavaServer Faces 1.x does not explicitly support client-side validation. All validation occurs on the server after the user has submitted the form data. If you want validation to occur inside the browser, you need to supply custom tags that contain the appropriate JavaScript commands. See Chapter 13 for details.

JavaServer Faces has built-in mechanisms that let you carry out the following validations:

- Checking the length of a string
- Checking limits for a numerical value (for example, > 0 or \( \leq 100 \))
- Checking that a value has been supplied

Table 6–4 lists the standard validators that are provided with JSF. You saw the string length validator in the preceding section. To validate numerical input, you use a range validator. For example,

```xml
<h:inputText id="amount" value="#{payment.amount}"
<f:validateLongRange minimum="10" maximum="10000"/>
</h:inputText>
```

The validator checks that the supplied value is \( \geq 10 \) and \( \leq 10000 \).

All the standard validator tags have `minimum` and `maximum` attributes. You need to supply one or both of these attributes.
Using Standard Validators

Checking for Required Values

To check that a value is supplied, you do not nest a validator inside the input component tag. Instead, you supply the attribute `required="true"`:

```html
<h:inputText id="date" value="#{payment.date}" required="true"/>
```

All JSF input tags support the `required` attribute. You can combine the `required` attribute with a nested validator:

```html
<h:inputText id="card" value="#{payment.card}" required="true">
    <f:validateLength minimum="13"/>
</h:inputText>
```

CAUTION: If the `required` attribute is not set and a user supplies a blank input, then no validation occurs at all! Instead, the blank input is interpreted as a request to leave the existing value unchanged.

An alternate syntax for attaching a validator to a component is to use the `f:validator` tag. You specify the ID of the validator and the validator parameters like this:

```html
<h:inputText id="card" value="#{payment.card}"
    f:validator validatorId="javax.faces.validator.LengthValidator">
    <f:attribute name="minimum" value="13"/>
</h:inputText>
```

### Table 6–4 Standard Validators

<table>
<thead>
<tr>
<th>JSP Tag</th>
<th>Validator Class</th>
<th>Attributes</th>
<th>Validates</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f:validateDoubleRange</code></td>
<td>DoubleRangeValidator</td>
<td><code>minimum, maximum</code></td>
<td>A double value within an optional range</td>
</tr>
<tr>
<td><code>f:validateLongRange</code></td>
<td>LongRangeValidator</td>
<td><code>minimum, maximum</code></td>
<td>A long value within an optional range</td>
</tr>
<tr>
<td><code>f:validateLength</code></td>
<td>LengthValidator</td>
<td><code>minimum, maximum</code></td>
<td>A String with a minimum and maximum number of characters</td>
</tr>
</tbody>
</table>
Yet another way of specifying the validator is with a validator attribute to the component tag (see “Validating with Bean Methods” on page 259).

NOTE: As of JSF 1.2, the f:validateLength, f:validateLongRange, f:validateDoubleRange, and f:validator tags have an optional binding attribute. This allows you to tie a validator instance to a backing bean property of type javax.faces.validator.Validator.

**Displaying Validation Errors**

Validation errors are handled in the same way as conversion errors. A message is added to the component that failed validation, the component is invalidated, and the current page is redisplayed immediately after the Process Validations phase has completed.

You use the h:message or h:messages tag to display the validation errors. For details, see “Displaying Error Messages” on page 226.

As of JSF 1.2, you can supply a custom message for a component by setting the requiredMessage or validatorMessage attribute, like this:

```xml
<h:inputText id="card" value="#{payment.card}" required="true">
  requiredMessage="#{msgs.cardRequired}"
  validatorMessage="#{msgs.cardInvalid}"
  f:validateLength minimum="13"/>
</h:inputText>
```

You can also globally override the default validator messages shown in Table 6–5. Define a message bundle for your application and supply messages with the appropriate keys, as shown under “Changing the Text of Standard Error Messages” on page 228.

**Table 6–5 Standard Validation Error Messages**

<table>
<thead>
<tr>
<th>Resource ID</th>
<th>Default Text</th>
<th>Reported By</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.faces.component.UIInput.REQUIRED</td>
<td>[0]: Validation Error: Value is required.</td>
<td>UIInput with required attribute when value is missing</td>
</tr>
<tr>
<td>javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE</td>
<td>[2]: Validation Error: Specified attribute is not between the expected values of {0} and {1}.</td>
<td>DoubleRangeValidator and LongRangeValidator when value is out of range and both minimum and maximum are specified</td>
</tr>
<tr>
<td>javax.faces.validator.LongRangeValidator.NOT_IN_RANGE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Using Standard Validators

NOTE: In JSF 1.1, the input label was not included in the validation messages. The key for the “not in range” messages was `javax.faces.validator.NOT_IN_RANGE`.

## Bypassing Validation

As you saw in the preceding examples, validation errors (as well as conversion errors) force a redisplay of the current page. This behavior can be problematic with certain navigation actions. Suppose, for example, you add a “Cancel” button to a page that contains required fields. If the user clicks “Cancel”, leaving a required field blank, then the validation mechanism kicks in and forces the current page to be redisplayed.
Chapter 6 ■ Conversion and Validation

It would be unreasonable to expect your users to fill in required fields before they are allowed to cancel their input. Fortunately, a bypass mechanism is available. If a command has the immediate attribute set, then the command is executed during the Apply Request Values phase.

Thus, you would implement a “Cancel” button like this:

```html
<h:commandButton value="Cancel" action="cancel" immediate="true"/>
```

**A Complete Validation Example**

The following sample application shows a form that employs all the standard JSF validation checks: required fields, string length, and numeric limits. The application makes sure that values are entered in all fields, the amount is between $10 and $10,000, the credit card number has at least 13 characters, and the PIN is a number between 1000 and 9999. Figure 6–6 shows typical validation error messages. A “Cancel” button is also provided to demonstrate the validation bypass.

**Figure 6–6 Typical validation error messages**

Figure 6–7 shows the directory structure of the application. Listing 6–5 contains the JSF page with the validators.
Using Standard Validators

Figure 6–7  Directory structure of the validation example

Listing 6–5  validator/web/index.jsp

```html
1. <html>
2.  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.  <f:view>
5.   <head>
6.     <link href="styles.css" rel="stylesheet" type="text/css"/>
7.     <title><h:outputText value="#{msgs.title}"/></title>
8.   </head>
9.   <body>
10.  <f:form>
11.   <h1><h:outputText value="#{msgs.enterPayment}"/></h1>
12.   <h:panelGrid columns="3">
13.     <h:inputText id="amount" label="#{msgs.amount}" value="#{payment.amount}" required="true">
14.       <f:convertNumber minFractionDigits="2"/>
15.       <f:validateDoubleRange minimum="10" maximum="10000"/>
16.     </h:inputText>
17.     <h:message for="amount" styleClass="errorMessage"/>
18.   </h:panelGrid>
19.   <h:inputText id="card" label="#{msgs.creditCard}" value="#{payment.card}" required="true">
20.     <f:validateLength minimum="13"/>
21.     <f:attribute name="requiredMessage" value="#{msgs.cardRequired}"/>
22.   </h:inputText>
23. </f:form>
24. </body>
25. </f:view>
26. </html>
```
Chapter 6 ■ Conversion and Validation

Programming with Custom Converters and Validators

JSF standard converters and validators cover a lot of bases, but many web applications must go further. For example, you may need to convert to types other than numbers and dates or perform application-specific validation, such as checking a credit card.

In the following sections, we show you how to implement application-specific converters and validators. These implementations require a moderate amount of programming.

Implementing Custom Converter Classes

A converter is a class that converts between strings and objects. A converter must implement the Converter interface, which has the following two methods:

- `Object getAsObject(FacesContext context, UIComponent component, String newValue)`
- `String getAsString(FacesContext context, UIComponent component, Object value)`

The first method converts a string into an object of the desired type, throwing a ConverterException if the conversion cannot be carried out. This method is called when a string is submitted from the client, typically in a text field. The second method converts an object into a string representation to be displayed in the client interface.

To illustrate these methods, we develop a custom converter for credit card numbers. Our converter allows users to enter a credit card number with or without spaces. That is, we accept inputs of the following forms:
Listing 6–6 shows the code for the custom converter. The `getAsObject` method of the converter strips out all characters that are not digits. It then creates an object of type `CreditCard`. If an error is found, then we generate a `FacesMessage` object and throw a `ConverterException`. We will discuss these steps in the next section, “Reporting Conversion Errors,” on page 245.

The `getAsString` method of our converter makes an effort to format the credit card number in a way that is pleasing to the eye of the user. The digits are separated into the familiar patterns, depending on the credit card type. Table 6–6 shows the most common credit card formats.

<table>
<thead>
<tr>
<th>Card Type</th>
<th>Digits</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>MasterCard</td>
<td>16</td>
<td>5xxx xxxx xxxx xxxx</td>
</tr>
<tr>
<td>Visa</td>
<td>16</td>
<td>4xxx xxxx xxxx xxxx</td>
</tr>
<tr>
<td>Visa</td>
<td>13</td>
<td>4xxx xxx xxx xxx</td>
</tr>
<tr>
<td>Discover</td>
<td>16</td>
<td>6xxx xxxx xxxx xxxx</td>
</tr>
<tr>
<td>American Express</td>
<td>15</td>
<td>37xx xxxxxx xxxxx</td>
</tr>
<tr>
<td>American Express</td>
<td>22</td>
<td>3xxxxx xxxxxxxxxx xxxxxxxxx</td>
</tr>
<tr>
<td>Diners Club, Carte Blanche</td>
<td>14</td>
<td>3xxxxx xxxx xxxx</td>
</tr>
</tbody>
</table>

In this example, the `CreditCard` class is minor; it contains just the credit card number (see Listing 6–7). We could have left the credit card number as a `String` object, reducing the converter to a formatter. However, most converters have a target type other than `String`. To make it easier for you to reuse this example, we use a distinct target type.

```
Listing 6–6  converter2/src/java/com/coresjf/CreditCardConverter.java

1. package com.corejsf;
2.
3. import javax.faces.application.FacesMessage;
4. import javax.faces.component.UIComponent;
5. import javax.faces.context.FacesContext;
6. import javax.faces.convert.Converter;
7. import javax.faces.convert.ConverterException;
8.
```
Chapter 6 ■ Conversion and Validation

Listing 6–6  converter2/src/java/com/coresjf/CreditCardConverter.java (cont.)

```java
9. public class CreditCardConverter implements Converter {
10.     public Object getAsObject(FacesContext context, UIComponent component,
11.         String newValue) throws ConverterException {
12.         String builder = new StringBuilder(newValue);
13.         boolean foundInvalidCharacter = false;
14.         char invalidCharacter = '\0';
15.         int i = 0;
16.         while (i < builder.length() && !foundInvalidCharacter) {
17.             char ch = builder.charAt(i);
18.             if (Character.isDigit(ch))
19.                 i++;
20.             else if (Character.isWhitespace(ch))
21.                 builder.deleteCharAt(i);
22.             else {
23.                 foundInvalidCharacter = true;
24.                 invalidCharacter = ch;
25.             }
26.         }
27.         if (foundInvalidCharacter) {
29.                 "com.corejsf.messages", "badCreditCardCharacter",
30.                 new Object[]{ new Character(invalidCharacter) });
31.             message.setSeverity(FacesMessage.SEVERITY_ERROR);
32.             throw new ConverterException(message);
33.         }
34.         return new CreditCard(builder.toString());
35.     }
36. }
37. }
38. }
39. }
40. public String getAsString(FacesContext context, UIComponent component,
41.     Object value) throws ConverterException {
42.     // length 13: xxxx xxx xxx xxx
43.     // length 14: xxxxx xxxx xxxxx
44.     // length 15: xxxxxxx xxxxxx
45.     // length 16: xxxxxx xxxx xxxxxx
46.     // length 22: xxxxxxx xxxxxxxx xxxxxxxx
47.     String v = value.toString();
48.     int[] boundaries = null;
49.     int length = v.length();
50.     if (length == 13)
51.         boundaries = new int[]{4, 7, 10};
52.     else if (length == 14)
53.         boundaries = new int[]{5, 9};
```

Conversion and Validation
Specifying Converters

One mechanism for specifying converters involves a symbolic ID that you register with the JSF application. We will use the ID com.corejsf.CreditCard for our credit card converter. The following entry to faces-config.xml associates the converter ID with the class that implements the converter:

```
<converter>
  <converter-id>com.corejsf.CreditCard</converter-id>
  <converter-class>com.corejsf.CreditCardConverter</converter-class>
</converter>
```
In the following examples, we will assume that the card property of the PaymentBean has type CreditCard, as shown in Listing 6–13 on page 254. Now we can use the f:converter tag and specify the converter ID:

```xml
<h:inputText value="#{payment.card}"
    f:converter converterId="com.corejsf.CreditCard"/>
</h:inputText>
```

Or, more succinctly, we can use the converter attribute:

```xml
<h:inputText value="#{payment.card}" converter="com.corejsf.CreditCard"/>
```

You can also access a converter without defining it in a configuration file. Use the converter attribute with a value expression that yields the converter object:

```xml
<h:outputText value="#{payment.card}" converter="#{bb.convert}"/>
```

Here, the bb bean must have a convert property of type Converter.

If you like, you can implement the property getter so that it returns an inner class object:

```java
public class BackingBean {
    ...
    public Converter getConvert() {
        return new Converter() {
            public Object getAsObject(FacesContext context, UIComponent component, String newValue) throws ConverterException { ... }
            public String getAsString(FacesContext context, UIComponent component, Object value) throws ConverterException { ... }
        };
    }
}
```

This approach is convenient because the conversion methods can access the bean’s private data.

Alternatively, if you are confident that your converter is appropriate for all conversions between String and CreditCard objects, then you can register it as the default converter for the CreditCard class:

```xml
<converter>
    <converter-for-class>com.corejsf.CreditCard</converter-for-class>
    <converter-class>com.corejsf.CreditCardConverter</converter-class>
</converter>
```

Now you do not have to mention the converter any longer. It is automatically used whenever a value reference has the type CreditCard. For example, consider the tag:

```xml
<h:inputText value="#{payment.card}"/>
```
When the JSF implementation converts the request value, it notices that the target type is `CreditCard`, and it locates the converter for that class. This is the ultimate in converter convenience for the page author!

```
javax.faces.convert.Converter

- Object getAsObject(FacesContext context, UIComponent component, String value)
    Converts the given string value into an object that is appropriate for storage in the given component.

- String getAsString(FacesContext context, UIComponent component, Object value)
    Converts the given object, which is stored in the given component, into a string representation.
```

**Reporting Conversion Errors**

When a converter detects an error, it should throw a `ConverterException`. For example, the `getAsObject` method of our credit card converter checks whether the credit card contains characters other than digits or separators. If it finds an invalid character, it signals an error:

```
if (foundInvalidCharacter) {
    FacesMessage message = new FacesMessage(
        "Conversion error occurred.", "Invalid card number. ");
    message.setSeverity(FacesMessage.SEVERITY_ERROR);
    throw new ConverterException(message);
}
```

The `FacesMessage` object contains the summary and detail messages that can be displayed with message tags.

```
javax.faces.application.FacesMessage

- FacesMessage(FacesMessage.Severity severity, String summary, String detail)
    Constructs a message with the given severity, summary, and detail. The severity is one of the constants SEVERITY_ERROR, SEVERITY_FATAL, SEVERITY_INFO, or SEVERITY_WARN in the FacesMessage class.

- FacesMessage(String summary, String detail)
    Constructs a message with severity SEVERITY_INFO and the given summary and detail.

- void setSeverity(FacesMessage.Severity severity)
    Sets the severity to the given level. The severity is one of the constants SEVERITY_ERROR, SEVERITY_FATAL, SEVERITY_INFO, or SEVERITY_WARN in the FacesMessage class.
```
Chapter 6 ■ Conversion and Validation

ConverterException

ConverterException(FacesMessage message)
ConverterException(FacesMessage message, Throwable cause)

These constructors create exceptions whose getMessage method returns the summary of the given message and whose getFacesMessage method returns the given message.

ConverterException()
ConverterException(String detailMessage)
ConverterException(Throwable cause)
ConverterException(String detailMessage, Throwable cause)

These constructors create exceptions whose getMessage method returns the given detail message and whose getFacesMessage method returns null.

FacesMessage getFacesMessage()

Returns the FacesMessage with which this exception object was constructed or returns null if none was supplied.

Getting Error Messages from Resource Bundles

Of course, for proper localization, you will want to retrieve the error messages from a message bundle.

Doing that involves some busywork with locales and class loaders:

1. Get the current locale.

   FacesContext context = FacesContext.getCurrentInstance();
   UIViewRoot viewRoot = context.getViewRoot();
   Locale locale = viewRoot.getLocale();

2. Get the current class loader. You need it to locate the resource bundle.

   ClassLoader loader = Thread.currentThread().getContextClassLoader();

3. Get the resource bundle with the given name, locale, and class loader.

   ResourceBundle bundle = ResourceBundle.getBundle(bundleName, locale, loader);

4. Get the resource string with the given ID from the bundle.

   String resource = bundle.getString(resourceId);

However, there are several wrinkles in the process. We actually need two message strings: one for the summary and one for the detail messages. By convention, the resource ID of a detail message is obtained by addition of the string _detail to the summary key. For example,
badCreditCardCharacter=Invalid card number.
badCreditCardCharacter_detail=The card number contains invalid characters.

Moreover, converters are usually part of a reusable library. It is a good idea to allow a specific application to override messages. (You saw in “Changing the Text of Standard Error Messages” on page 228 how to override the standard converter messages.) Therefore, you should first attempt to locate the messages in the application-specific message bundle before retrieving the default messages.

Recall that an application can supply a bundle name in a configuration file, such as

```xml
<faces-config>
  <application>
    <message-bundle>com.mycompany.myapp.messages</message-bundle>
  </application>
  ...
</faces-config>
```

The following code snippet retrieves that bundle name:

```java
Application app = context.getApplication();
String appBundleName = app.getResourceBundle();
```

Look up your resources in this bundle before going to the library default.

Finally, you may want some messages to provide detailed information about the nature of the error. For example, you want to tell the user which character in the credit card number was objectionable. Message strings can contain placeholders {0}, {1}, and so on—for example:

```
The card number contains the invalid character {0}.
```

The java.text.MessageFormat class can substitute values for the placeholders:

```java
Object[] params = ...;
MessageFormat formatter = new MessageFormat(resource, locale);
String message = formatter.format(params);
```

Here, the params array contains the values that should be substituted. (For more information about the MessageFormat class, see Horstmann and Cornell, 2004, 2005, Core Java™ 2, vol. 2, chap. 10.)

Ideally, much of this busywork should have been handled by the JSF framework. Of course, you can find the relevant code in the innards of the reference implementation, but the framework designers chose not to make it available to JSF programmers.

We provide the package com.corejsf.util with convenience classes that implement these missing pieces. Feel free to use these classes in your own code.
The `com.corejsf.util.Messages` class has a static method, `getMessage`, that returns a `FacesMessage` with a given bundle name, resource ID, and parameters:

```java
FacesMessage message = com.corejsf.util.Messages.getMessage(
    "com.corejsf.messages", "badCreditcardCharacter",
    new Object[] { new Character(invalidCharacter) });
```

You can pass `null` for the parameter array if the message does not contain placeholders.

Our implementation follows the JSF convention of displaying missing resources as `???resourceId??`. See Listing 6–8 for the source code.

---

**NOTE:** If you prefer to reuse the standard JSF message for conversion errors, call

```java
FacesMessage message = com.corejsf.util.Messages.getMessage(
    "javax.faces.Messages",
    "javax.faces.component.UIInput.CONVERSION",
    null);
```

---

### javax.faces.context.FacesContext

- **static FacesContext getCurrentInstance()**
  
  Gets the context for the request that is being handled by the current thread, or `null` if the current thread does not handle a request.

- **UIViewRoot getViewRoot()**
  
  Gets the root component for the request described by this context.

### javax.faces.component.UIViewRoot

- **Locale getLocale()**
  
  Gets the locale for rendering this view.

---

### Listing 6–8 converter2/src/java/com/corejsf/util/Messages.java

```java
package com.corejsf.util;
import java.text.MessageFormat;
import java.util.Locale;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
import javax.faces.application.Application;
```
Listing 6–8  converter2/src/java/com/corejsf/util/Messages.java (cont.)

8. import javax.faces.application.FacesMessage;
9. import javax.faces.component.UIViewRoot;
10. import javax.faces.context.FacesContext;
11.
12. public class Messages {
13.   public static FacesMessage getMessage(String bundleName, String resourceId,
14.       Object[] params) {
15.       FacesContext context = FacesContext.getCurrentInstance();
16.       Application app = context.getApplication();
17.       String appBundle = app.getMessageBundle();
18.       Locale locale = getLocale(context);
19.       ClassLoader loader = getClassLoader();
20.       String summary = getString(appBundle, bundleName, resourceId,
21.           locale, loader, params);
22.       if (summary == null) summary = "???" + resourceId + "???";
23.       String detail = getString(appBundle, bundleName, resourceId + ".detail",
24.           locale, loader, params);
25.       return new FacesMessage(summary, detail);
26.   }
27.
28.   public static String getString(String bundle, String resourceId,
29.       Object[] params) {
30.       FacesContext context = FacesContext.getCurrentInstance();
31.       Application app = context.getApplication();
32.       String appBundle = app.getMessageBundle();
33.       Locale locale = getLocale(context);
34.       ClassLoader loader = getClassLoader();
35.       return getString(appBundle, bundle, resourceId, locale, loader, params);
36.   }
37.
38.   public static String getString(String bundle1, String bundle2, String resourceId, Locale locale, ClassLoader loader, 
39.       Object[] params) {
40.       String resource = null;
41.       ResourceBundle bundle;
42.       if (bundle1 != null) {
43.         bundle = ResourceBundle.getBundle(bundle1, locale, loader);
44.         if (bundle != null)
45.           try {
46.             resource = bundle.getString(resourceId);
47.           } catch (MissingResourceException ex) {
48.             
49.           }
50.       }
51.       
52.   }
The Custom Converter Sample Application

Here are the remaining pieces of our next sample application. Figure 6–8 shows the directory structure. Listings 6–9 and 6–10 show the input and result pages. Look at the inputText and outputText tags for the credit card numbers to see the two styles of specifying a custom converter. (Both converter specifications could have been omitted if the converter had been registered to be the default for the CreditCard type.)

The custom converter is defined in the faces-config.xml file (Listing 6–11). The messages.properties file (shown in Listing 6–12) contains the error message for the credit card converter. Finally, Listing 6–13 shows the payment bean with three properties of type double, Date, and CreditCard.
Programming with Custom Converters and Validators

Figure 6–8  Directory structure of the custom converter example

Listing 6–9  converter2/web/index.jsp

```html
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4. <f:view>
5. `<head>
6. <link href="styles.css" rel="stylesheet" type="text/css"/>
7. <title><h:outputText value="#{msgs.title}"/></title>
8. </head>
9. <body>
10. <h:form>
11. <h1><h:outputText value="#{msgs.enterPayment}"/></h1>
12. <h:panelGrid columns="3">
13. <h:outputText value="#{msgs.amount}"/>
14. <h:inputText id="amount" label="#{msgs.amount}" value="#{payment.amount}"/>
15. <f:convertNumber minFractionDigits="2"/>
16. </h:inputText>
17. <h:message for="amount" styleClass="errorMessage"/>
18. </h:form>
19. <h:outputText value="#{msgs.creditCard}"/>
20. <h:inputText id="card" label="#{msgs.creditCard}" value="#{payment.card}"/>
21. <f:converter converterId="com.corejsf.CreditCard"/>
22. </h:inputText>
23. <h:message for="card" styleClass="errorMessage"/>
24. 
25. 
26. 
```
Chapter 6 ■ Conversion and Validation

Listing 6-9  converter2/web/index.jsp (cont.)

27.  <h:outputText value="#{msgs.expirationDate}"/>
28.  <h:inputText id="date" label="#{msgs.expirationDate}"
29.     value="#{payment.date}"
30.     f:convertDateTime pattern="MM/yyyy"/>
31.  </h:inputText>
32.  <h:message for="date" styleClass="errorMessage"/>
33.  </h:panelGrid>
34.  <h:commandButton value="Process" action="process"/>
35.  </h:form>
36.  </body>
37.  </f:view>
38.  </html>

Listing 6-10  converter2/web/result.jsp

1.  <html>
2.  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
3.  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
4.  <f:view>
5.  <head>
6.      <link href="styles.css" rel="stylesheet" type="text/css"/>
7.      <title><h:outputText value="#{msgs.title}"/></title>
8.  </head>
9.  <body>
10.  <h:form>
11.      <h1><h:outputText value="#{msgs.paymentInformation}"/></h1>
12.       <h:panelGrid columns="2">
13.           <h:outputText value="#{msgs.amount}"/>
14.           <h:outputText value="#{payment.amount}" f:convertNumber type="currency"/>
15.       </h:outputText>
16.       <h:outputText value="#{msgs.creditCard}"/>
17.       <h:outputText value="#{payment.card}
18.           converter="com.corejsf.CreditCard"/>
19.       <h:outputText value="#{msgs.expirationDate}"/>
20.       <h:outputText value="#{payment.date}" f:convertDateTime pattern="MM/yyyy"/>
21.       </h:outputText>
22.       <h:panelGrid>
23.           <h:commandButton value="Back" action="back"/>
24.       </h:form>
25.  </body>
26.  </f:view>
27.  </html>
Listing 6–11 converter2/web/WEB-INF/faces-config.xml

```xml
<?xml version="1.0"?
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
version="1.2">
<application>
<message-bundle>com.corejsf.messages</message-bundle>
</application>

<navigation-rule>
<from-view-id>/index.jsp</from-view-id>
<navigation-case>
<from-outcome>process</from-outcome>
<to-view-id>/result.jsp</to-view-id>
</navigation-case>
</navigation-rule>

<navigation-rule>
<from-view-id>/result.jsp</from-view-id>
<navigation-case>
<from-outcome>back</from-outcome>
<to-view-id>/index.jsp</to-view-id>
</navigation-case>
</navigation-rule>

<converter>
<converter-id>com.corejsf.CreditCard</converter-id>
<converter-class>com.corejsf.CreditCardConverter</converter-class>
</converter>

<managed-bean>
<managed-bean-name>payment</managed-bean-name>
<managed-bean-class>com.corejsf.PaymentBean</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<application>
<resource-bundle>
<base-name>com.corejsf.messages</base-name>
<var>msgs</var>
</resource-bundle>
</application>
</faces-config>
```
Implementing Custom Validator Classes

Implementing custom validator classes is a two-step process, similar to the process you saw in the preceding section:

1. Implement a validator by implementing the `javax.faces.validator.Validator` interface.
2. Register your validator in a configuration file (such as `faces-config.xml`).

Listing 6–12 converter2/src/java/com/corejsf/messages.properties

1. badCreditCardCharacter=Invalid card number.
2. badCreditCardCharacter_detail=The card number contains the invalid character {0}.
3. title=An Application to Test Data Conversion
4. enterPayment=Please enter the payment information
5. amount=Amount
6. creditCard=Credit Card
7. expirationDate=Expiration date (Month/Year)
8. process=Process
9. paymentInformation=Payment information

Listing 6–13 converter2/src/java/com/corejsf/PaymentBean.java

1. package com.corejsf;
2. import java.util.Date;
3.
4. public class PaymentBean {
5.     private double amount;
6.     private CreditCard card = new CreditCard("");
7.     private Date date = new Date();
8.     
9.     // PROPERTY: amount
10.    public void setAmount(double newValue) { amount = newValue; }
11.    public double getAmount() { return amount; }
12. 
13.     // PROPERTY: card
14.    public void setCard(CreditCard newValue) { card = newValue; }
15.    public CreditCard getCard() { return card; }
16. 
17.     // PROPERTY: date
18.    public void setDate(Date newValue) { date = newValue; }
19.    public Date getDate() { return date; }
20. }

Implementing Custom Validator Classes

Implementing custom validator classes is a two-step process, similar to the process you saw in the preceding section:

1. Implement a validator by implementing the `javax.faces.validator.Validator` interface.
2. Register your validator in a configuration file (such as `faces-config.xml`).
The `Validator` interface defines only one method:

```java
void validate(FacesContext context, UIComponent component, Object value)
```

If validation fails, generate a `FacesMessage` that describes the error, construct a `ValidatorException` from the message, and throw it:

```java
if (validation fails) {
    FacesMessage message = ...;
    message.setSeverity(FacesMessage.SEVERITY_ERROR);
    throw new ValidatorException(message);
}
```

The process is analogous to the reporting of conversion errors, except that you throw a `ValidatorException` instead of a `ConverterException`.

For example, Listing 6–14 shows a validator that checks the digits of a credit card, using the Luhn formula. Figure 6–9 shows the application at work. As described under “Getting Error Messages from Resource Bundles” on page 246, we use the convenience class `com.corejsf.util.Messages` to locate the message strings in a resource bundle.

**Figure 6–9  Luhn check failed**

**NOTE:** The Luhn formula—developed by a group of mathematicians in the late 1960s—verifies and generates credit card numbers, as well as Social Insurance numbers for the Canadian government. The formula can detect whether a digit is entered wrongly or whether two digits were transposed. See the web site [http://www.merriam-park.com/anatomycc.htm](http://www.merriam-park.com/anatomycc.htm) for more information about the Luhn formula. For debugging, it is handy to know that the number 4111 1111 1111 1111 passes the Luhn check.
Chapter 6 ■ Conversion and Validation

javax.faces.validator.Validator

- void validate(FacesContext context, UIComponent component, Object value)
  Validates the component to which this validator is attached. If there is a validation error, throw a ValidatorException.

Listing 6–14 validator2/src/java/com/corejsf/CreditCardValidator.java

```java
package com.corejsf;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class CreditCardValidator implements Validator {
  public void validate(FacesContext context, UIComponent component,
                       Object value) {
    if (value == null) return;
    String cardNumber;
    if (value instanceof CreditCard)
      cardNumber = value.toString();
    else
      cardNumber = getDigitsOnly(value.toString());
    if (!luhnCheck(cardNumber)) {
      FacesMessage message = com.corejsf.util.Messages.getMessage(
        "com.corejsf.messages", "badLuhnCheck", null);
      message.setSeverity(FacesMessage.SEVERITY_ERROR);
      throw new ValidatorException(message);
    }
  }

  private static boolean luhnCheck(String cardNumber) {
    int sum = 0;
    for (int i = cardNumber.length() - 1; i >= 0; i -= 2) {
      int d = 2 * Integer.parseInt(cardNumber.substring(i, i + 1));
      if (d > 9) d -= 9;
      sum += d;
    }
    return sum % 10 == 0;
  }
}
```
Registering Custom Validators

Now that we have created a validator, we need to register it in a configuration file (such as faces-config.xml), like this:

```xml
<validator>
  <validator-id>com.corejsf.CreditCard</validator-id>
  <validator-class>com.corejsf.CreditCardValidator</validator-class>
</validator>
```

You can use custom validators with the `f:validator` tag—for example, the following code fragment uses the credit card validator discussed above:

```xml
<h:inputText id="card" value="#{payment.card}" required="true">
  <f:converter converterId="com.corejsf.CreditCard"/>
  <f:validator validatorId="com.corejsf.CreditCard"/>
</h:inputText>
```

The `validatorId` specified for `f:validator` must correspond to a validator ID specified in the configuration file. The `f:validator` tag uses the validator ID to look up the corresponding class, creates an instance of that class if necessary, and invokes its `validate` method.

NOTE: JSF uses separate namespaces for converter and validator IDs. Thus, it is okay to have both a converter and a validator with the ID `com.corejsf.CreditCard`.


### Listing 6–14 validator2/src/java/com/corejsf/CreditCardValidator.java (cont.)

```java
private static String getDigitsOnly(String s) {
    StringBuilder digitsOnly = new StringBuilder();
    char c;
    for(int i = 0; i < s.length(); i++) {
        c = s.charAt(i);
        if (Character.isDigit(c)) {
            digitsOnly.append(c);
        }
    }
    return digitsOnly.toString();
}
```
The remainder of the sample application is straightforward. Figure 6–10 shows the directory structure, and Listing 6–15 contains the JSF page.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
  <head>
    <link href="styles.css" rel="stylesheet" type="text/css"/>
    <title><h:outputText value="#{msgs.title}"/></title>
  </head>
  <body>
    <h:form>
      <h1><h:outputText value="#{msgs.enterPayment}"/></h1>
      <h:panelGrid columns="3">
        <h:outputText value="#{msgs.amount}"/>
        <h:inputText id="amount" label="#{msgs.amount}"
          value="#{payment.amount}"
          f:convertNumber minFractionDigits="2"/>
        <h:message for="amount" styleClass="errorMessage"/>
      </h:panelGrid>
    </h:form>
  </body>
</f:view>
```

Figure 6–10  The directory structure of the Luhn check example
The `f:validator` tag is useful for simple validators that do not have parameters, such as the credit validator discussed above. If you need a validator with properties that can be specified in a JSF page, you should implement a custom tag for your validator. You will see how to do that in “Implementing Custom Converters and Validators” on page 432 of Chapter 9.

### Validating with Bean Methods

In the preceding section, you saw how to implement a validation class. However, you can also add the validation method to an existing class and invoke it through a method expression, like this:

```jsp
<h:inputText id="card" label="#{msgs.creditCard}" value="#{payment.card}" required="true" validator="#{payment.luhnCheck}"/>
```

The payment bean must then have a method with the exact same signature as the `validate` method of the `Validator` interface:

```java
public class PaymentBean {
...
    public void luhnCheck(FacesContext context, UIComponent component, Object value) {
        ... // same code as in the preceding example
    }
}
```
Why would you want to do this? There is one major advantage. The validation method can access other instance fields of the class. You will see an example in the next section, “Supplying Attributes to Converters.”

On the downside, this approach makes it more difficult to move a validator to a new web application, so you would probably only use it for application-specific scenarios.

---

**CAUTION:** The value of the validator attribute is a *method expression*, whereas the seemingly similar converter attribute specifies a converter ID (if it is a string) or a *converter object* (if it is a value expression). As Emerson said, “A foolish consistency is the hobgoblin of little minds.”

---

**Supplying Attributes to Converters**

Every JSF component can store arbitrary attributes. You can set an attribute of the component to which you attach a converter; use the `f:attribute` tag. Your converter can then retrieve the attribute from its component. Here is how that technique would work to set the separator string for the credit card converter.

When attaching the converter, also nest an `f:attribute` tag inside the component:

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

In the converter, retrieve the attribute as follows:

```java
separator = (String) component.getAttributes().get("separator");
```

In Chapter 9, you will see a more elegant mechanism for passing attributes to a converter—writing your own converter tag.

---

**Validating Relationships Between Multiple Components**

The validation mechanism in JSF was designed to validate a *single component*. However, in practice, you often need to ensure that related components have reasonable values before letting the values propagate into the model. For example, as we noted earlier, it is not a good idea to ask users to enter a date...
into a single text field. Instead, you would use three different text fields, for the
day, month, and year, as in Figure 6–11.

Figure 6–11 Validating a relationship involving three components

If the user enters an illegal date, such as February 30, you would want to show
a validation error and prevent the illegal data from entering the model.
The trick is to attach the validator to the last of the components. By the time its
validator is called, the preceding components have passed validation and had
their local values set. The last component has passed conversion, and the con-
verted value is passed as the Object parameter of the validation method.

Of course, you need to have access to the other components. You can easily
achieve that access by using a backing bean that contains all components of the
current form (see Listing 6–16). Attach the validation method to the backing
bean:

```java
public class BackingBean {
    private UIInput dayInput;
    private UIInput monthInput;
    ...
    public void validateDate(FacesContext context, UIComponent component,
                              Object value) {
        int d = ((Integer) dayInput.getLocalValue()).intValue();
        int m = ((Integer) monthInput.getLocalValue()).intValue();
        int y = ((Integer) value).intValue();

        if (!isValidDate(d, m, y)) {
            FacesMessage message = ...;
```
throw new ValidatorException(message);
}

...  
}

Note that the value lookup is a bit asymmetric. The last component does not yet have the local value set because it has not passed validation.

Figure 6–12 shows the application’s directory structure. Listing 6–17 shows the JSF page. Note the converter property of the last input field. Also note the use of the binding attributes that bind the input components to the backing bean.

```
package com.corejsf;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

class BackingBean {
    private int day;
    private int month;
    private int year;
    private UIInput dayInput;
    private UIInput monthInput;
    private UIInput yearInput;

    public void validateInput(UIInput input) {
        FacesContext context = FacesContext.getCurrentInstance();
        FacesMessage message = new FacesMessage(input.getInvalidValue().toString());
        context.addMessage(input.getComponentId(), message);
        throw new ValidatorException(message);
    }
}
```

Figure 6–12  Directory structure of the date validation example
Listing 6–16  validator3/src/java/com/corejsf/BackingBean.java (cont.)

16. // PROPERTY: day
17. public int getDay() { return day; }
18. public void setDay(int newValue) { day = newValue; }
19.
20. // PROPERTY: month
21. public int getMonth() { return month; }
22. public void setMonth(int newValue) { month = newValue; }
23.
24. // PROPERTY: year
25. public int getYear() { return year; }
26. public void setYear(int newValue) { year = newValue; }
27.
28. // PROPERTY: dayInput
29. public UIInput getDayInput() { return dayInput; }
30. public void setDayInput(UIInput newValue) { dayInput = newValue; }
31.
32. // PROPERTY: monthInput
33. public UIInput getMonthInput() { return monthInput; }
34. public void setMonthInput(UIInput newValue) { monthInput = newValue; }
35.
36. // PROPERTY: yearInput
37. public UIInput getYearInput() { return yearInput; }
38. public void setYearInput(UIInput newValue) { yearInput = newValue; }
39.
40. public void validateDate(FacesContext context, UIComponent component, Object value) {
41.    int d = ((Integer) dayInput.getLocalValue()).intValue();
42.    int m = ((Integer) monthInput.getLocalValue()).intValue();
43.    int y = ((Integer) value).intValue();
44.
45.    if (!isValidDate(d, m, y)) {
46.        FacesMessage message
47.            = com.corejsf.util.Messages.getMessage(  
48.                "com.corejsf.messages", "invalidDate", null);
49.        message.setSeverity(FacesMessage.SEVERITY_ERROR);
50.        throw new ValidatorException(message);
51.    }
52. }
53. 
54. private static boolean isValidDate(int d, int m, int y) {
55.    if (d < 1 || m < 1 || m > 12) return false;
56.    if (m == 2) {
57.        if (isLeapYear(y)) return d <= 29;
58.        else return d <= 28;
59.    }
60. }
else if (m == 4 || m == 6 || m == 9 || m == 11)
    return d <= 30;
else
    return d <= 31;
}
private static boolean isLeapYear(int y) {
    return y % 4 == 0 && (y % 400 == 0 || y % 100 != 0);
}

Listing 6–17 validator3/web/index.jsp
<html>
<head>
<link href="styles.css" rel="stylesheet" type="text/css"/>
<title><h:outputText value="#{msgs.title}" /></title>
</head>
<body>
<h:form>
<h1><h:outputText value="#{msgs.enterDate}" /></h1>
<h:panelGrid columns="3">
    <h:outputText value="#{msgs.day}" />
    <h:inputText value="#{bb.day}" binding="#{bb.dayInput}" size="2" required="true" />
    <h:panelGroup />
    <h:outputText value="#{msgs.month}" />
    <h:inputText value="#{bb.month}" binding="#{bb.monthInput}" size="2" required="true" />
    <h:panelGroup />
    <h:outputText value="#{msgs.year}" />
    <h:inputText id="year" value="#{bb.year}" binding="#{bb.yearInput}" size="4" required="true" validator="#{bb.validateDate}" />
    <h:message for="year" styleClass="errorMessage" />
</h:panelGrid>
<h:commandButton value="#{msgs.submit}" action="submit" />
</h:form>
</body>
</html>
An alternative approach is to attach the validator to a hidden input field that comes after all other fields on the form:

```xml
<h:inputHidden id="datecheck" validator="#{bb.validateDate}" value="needed"/>
```

The hidden field is rendered as a hidden HTML input field. When the field value is posted back, the validator kicks in. (It is essential that you supply some field value. Otherwise, the component value is never updated.) With this approach, the validation function is more symmetrical since all other form components already have their local values set.

**NOTE:** It would actually be worthwhile to write a custom date component that renders three input fields and has a single value of type Date. That single component could then be validated easily. However, the technique of this section is useful for any form that needs validation across fields.

As you have seen, JSF provides extensive and extensible support for conversion and validation. You can use the JSF standard converter and validators with one line of code in your JSF pages, or you can supply your own logic if more complex conversions or validations are needed. Finally, as you will see in Chapter 9, you can define your own conversion and validation tags.