MANAGED BEANS

Topics in This Chapter

Chapter 2

A central theme of web application design is the separation of presentation and business logic. JSF uses *beans* to achieve this separation. JSF pages refer to bean properties, and the program logic is contained in the bean implementation code. Because beans are so fundamental to JSF programming, we discuss them in detail in this chapter.

The first half of the chapter discusses the essential features of beans that every JSF developer needs to know. We then present an example program that puts these essentials to work. The remaining sections cover more technical aspects about bean configuration and value expressions. You can safely skip these sections when you first read this book and return to them when the need arises.

Definition of a Bean

According to the JavaBeans specification (available at `http://java.sun.com/products/javabeans/`), a Java bean is "a reusable software component that can be manipulated in a builder tool." That is a pretty broad definition and indeed, as you will see in this chapter, beans are used for a variety of purposes.

At first glance, a bean seems to be similar to an object. However, beans serve a different purpose. Objects are created and manipulated inside a Java program when the program calls constructors and invokes methods. Yet, beans can be configured and manipulated *without programming*.

NOTE: You may wonder where the term "bean" comes from. Well, Java is a synonym for coffee (at least in the United States), and coffee is made from beans that encapsulate its flavor. You may find the analogy cute or annoying, but the term has stuck.

The "classic" application for JavaBeans is a user interface builder. A palette window in the builder tool contains component beans such as text fields, sliders, checkboxes, and so on. Instead of writing Swing code, you use a user interface designer to drag and drop component beans from the palette into a form. Then you can customize the components by selecting property values from a *property sheet* dialog (see Figure 2–1).

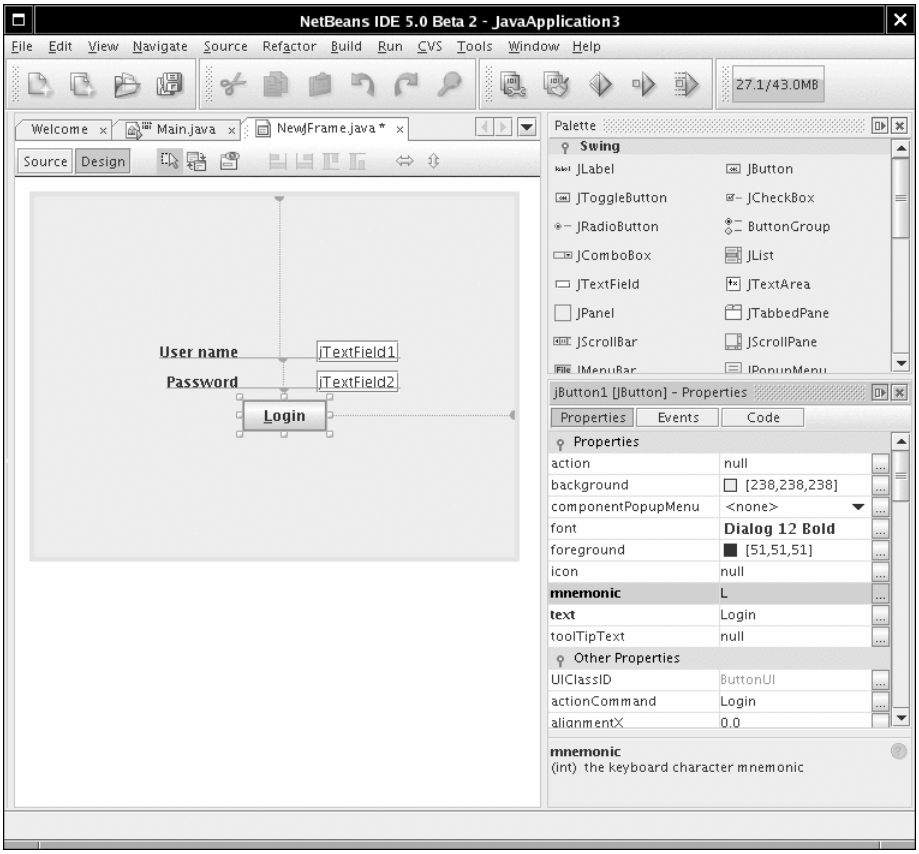

Figure 2–1 Customizing a bean in a GUI builder

In the context of JSF, beans go beyond user interface components. You use beans whenever you need to wire up Java classes with web pages or configuration files.

Consider the `login` application in Chapter 1, shown in "A Simple Example" on page 6. A `UserBean` instance is configured in the `faces-config.xml` file:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

This means: Construct an object of the class `com.corejsf.UserBean`, give it the name `user`, and keep it alive for the duration of the *session*—that is, for all requests that originate from the same client.

Once the bean has been defined, it can be accessed by JSF components. For example, this input field reads and updates the `password` property of the `user` bean:

```
<h:inputSecret value="#{user.password}"/>
```

As you can see, the JSF developer does not need to write any code to construct and manipulate the `user` bean. The JSF implementation constructs the beans according to the `managed-bean` elements in the configuration file.

In a JSF application, beans are commonly used for the following purposes:

- User interface components (traditional user interface beans)
- Tying together the behavior of a web form (called "backing beans")
- Business objects whose properties are displayed on web pages
- Services such as external data sources that need to be configured when an application is assembled

Because beans are so ubiquitous, we now turn to a review of those parts of the JavaBeans specification that are relevant to JSF programmers.

Bean Properties

Bean classes need to follow specific programming conventions to expose features that tools can use. We discuss these conventions in this section.

The most important features of a bean are the properties that it exposes. A *property* is any attribute of a bean that has

- A name
- A type
- Methods for getting and/or setting the property value

For example, the `UserBean` class of the preceding chapter has a property with name `password` and type `String`. The methods `getPassword` and `setPassword` access the property value.

Some programming languages, in particular Visual Basic and C#, have direct support for properties. However, in Java, a bean is simply a class that follows certain coding conventions.

The JavaBeans specification puts a single demand on a bean class: It must have a public default constructor—that is, a constructor without parameters. However, to define properties, a bean must either use a *naming pattern* for property getters and setters, or it must define property descriptors. The latter approach is quite tedious and not commonly used, and we will not discuss it here. See Horstmann and Cornell, 2004, 2005. *Core Java™ 2,* vol. 2, chap. 8, for more information.

Defining properties with naming patterns is straightforward. Consider the following pair of methods:

```
public T getFoo()
public void setFoo(T newValue)
```

The pair corresponds to a read-write property with type *T* and name `foo`. If you have only the first method, then the property is read-only. If you have only the second method, then the property is write-only.

The method names and signatures must match the pattern precisely. The method name must start with `get` or `set`. A `get` method must have no parameters. A `set` method must have one parameter and no return value. A bean class can have other methods, but the methods do not yield bean properties.

Note that the name of the property is the "decapitalized" form of the part of the method name that follows the `get` or `set` prefix. For example, `getFoo` gives rise to a property named `foo`, with the first letter turned into lower case. However, if the first *two* letters after the prefix are upper case, then the first letter stays unchanged. For example, the method name `getURL` defines a property `URL` and not `uRL`.

For properties of type `boolean`, you have a choice of prefixes for the method that reads the property. Both

```
public boolean isConnected()
```

and

```
public boolean getConnected()
```

are valid names for the reader of the `connected` property.

NOTE: The JavaBeans specification also defines indexed properties, specified by method sets such as the following:

```
public T[] getFoo()
public T getFoo(int index)
public void setFoo(T[] newArray)
public void setFoo(int index, T newValue)
```

However, JSF provides no support for accessing the indexed values.

The JavaBeans specification is silent on the *behavior* of the getter and setter methods. In many situations, these methods simply manipulate an instance field. But they may equally well carry out more sophisticated operations, such as database lookups, data conversion, validation, and so on.

A bean class may have other methods beyond property getters and setters. Of course, those methods do not give rise to bean properties.

Value Expressions

Many JSF user interface components have an attribute `value` that lets you specify either a value or a *binding* to a value that is obtained from a bean property. For example, you can specify a direct value:

```
<h:outputText value="Hello, World!"/>
```

Or you can specify a value expression:

```
<h:outputText value="#{user.name}"/>
```

In most situations, a value expression such as `#{user.name}` describes a property. Note that the expression can be used both for reading and writing when it is used in an input component, such as

```
<h:inputText value="#{user.name}"/>
```

The property getter is invoked when the component is rendered. The property setter is invoked when the user response is processed.

We will discuss the syntax of value expressions in detail under "The Syntax of Value Expressions" on page 64.

NOTE: JSF value expressions are related to the expression language used in JSP. Those expressions are delimited by `${...}` instead of `#{...}`. As of JSF 1.2 and JSP 2.1, the syntax of both expression languages has been unified. (See "The Syntax of Value Expressions" on page 64 for a complete description of the syntax.)

The `${...}` delimiter denotes *immediate* evaluation of expressions, at the time that the application server processes the page. The `#{...}` delimiter denotes *deferred* evaluation. With deferred evaluation, the application server retains the expression and evaluates it whenever a value is needed.

As a rule of thumb, you always use deferred expressions for JSF component properties, and you use immediate expressions in plain JSP or JSTL (JavaServer Pages Standard Template Library) constructs. (These constructs are rarely needed in JSF pages.)

Message Bundles

When you implement a web application, it is a good idea to collect all message strings in a central location. This process makes it easier to keep messages consistent and, crucially, makes it easier to localize your application for other locales. In this section, we show you how JSF makes it simple to organize messages. In the section "A Sample Application" on page 46, we put managed beans and message bundles to work.

You collect your message strings in a file in the time-honored *properties* format:

```
guessNext=Guess the next number in the sequence!
answer=Your answer:
```

NOTE: Look into the API documentation of the `load` method of the `java.util.Properties` class for a precise description of the file format.

Save the file together with your classes—for example, `insrc/java/com/corejsf/messages.properties`. You can choose any directory path and file name, but you must use the extension `.properties`.

You can declare the message bundle in two ways. The simplest way is to include the following elements in your `faces-config.xml` file:

```
<application>
   <resource-bundle>
      <base-name>com.corejsf.messages</base-name>
      <var>msgs</var>
   </resource-bundle>
</application>
```

Alternatively, you can add the `f:loadBundle` element to each JSF page that needs access to the bundle, like this:

```
<f:loadBundle basename="com.corejsf.messages" var="msgs"/>
```

In either case, the messages in the bundle are accessible through a map variable with the name `msgs`. (The base name `com.corejsf.messages` looks like a class name, and indeed the properties file is loaded by the class loader.)

You can now use value expressions to access the message strings:

```
<h:outputText value="#{msgs.guessNext}"/>
```

That is all there is to it! When you are ready to localize your application for another locale, you simply supply localized bundle files.

NOTE: The `resource-bundle` element is more efficient than the `f:loadBundle` action since the bundle can be created once for the entire application. However, it is a JSF 1.2 feature. If you want your application to be compatible with JSF 1.1, you must use `f:loadBundle`.

When you localize a bundle file, you need to add a locale suffix to the file name: an underscore followed by the lower case, two-letter ISO-639 language code. For example, German strings would be in `com/corejsf/messages_de.properties`.

NOTE: You can find a listing of all two- and three-letter ISO-639 language codes at `http://www.loc.gov/standards/iso639-2/`.

As part of the internationalization support in Java, the bundle that matches the current locale is automatically loaded. The default bundle without a locale prefix is used as a fallback when the appropriate localized bundle is not available. See Horstmann and Cornell, 2004, 2005. *Core Java™ 2,* vol. 2, chap. 10, for a detailed description of Java internationalization.

NOTE: When you prepare translations, keep one oddity in mind: Message bundle files are not encoded in UTF-8. Instead, Unicode characters beyond 127 are encoded as `\uxxxx` escape sequences. The Java SDK utility `native2ascii` can create these files.

You can have multiple bundles for a particular locale. For example, you may want to have separate bundles for commonly used error messages.

Messages with Variable Parts

Often, messages have variable parts that need to be filled. For example, suppose we want to display the sentence "You have *n* points.", where *n* is a value that is retrieved from a bean. Make a resource string with a placeholder:

```
currentScore=Your current score is {0}.
```

Placeholders are numbered {0}, {1}, {2}, and so on. In your JSF page, use the h:outputFormat tag and supply the values for the placeholders as f:param child elements, like this:

```
<h:outputFormat value="#{msgs.currentScore}">
   <f:param value="#{quiz.score}"/>
</h:outputFormat>
```

The h:outputFormat tag uses the MessageFormat class from the standard library to format the message string. That class has several features for locale-aware formatting.

You can format numbers as currency amounts by adding a suffix number,currency to the placeholder, like this:

```
currentTotal=Your current total is {0,number,currency}.
```

In the United States, a value of 1023.95 would be formatted as $1,023.95. The same value would be displayed as €1.023,95 in Germany, using the local currency symbol and decimal separator convention.

The choice format lets you format a number in different ways, such as "zero points", "one point", "2 points", "3 points", and so on. Here is the format string that achieves this effect:

```
currentScore=Your current score is {0,choice,0#zero points|1#one point|2#{0} points}.
```

There are three cases: 0, 1, and ≥ 2. Each case defines a separate message string.

Note that the 0 placeholder appears twice, once to select a choice, and again in the third choice, to produce a result such as "3 points".

Listings 2–5 and 2–6 on page 53 illustrate the choice format in our sample application. The English locale does not require a choice for the message, "Your score is . . . ". However, in German, this is expressed as "Sie haben . . . punkte" (You have . . . points). Now the choice format is required to deal with the singular form "einen punkt" (one point).

For more information on the MessageFormat class, see the API documentation or Horstmann and Cornell, 2004, 2005. *Core Java™ 2,* vol. 2, chap. 10.

Setting the Application Locale

Once you have prepared your message bundles, you need to decide how to set the locale of your application. You have three choices:

1. You can let the browser choose the locale. Set the default and supported locales in `WEB-INF/faces-config.xml` (or another application configuration resource):

   ```
   <faces-config>
      <application>
         <locale-config>
            <default-locale>en</default-locale>
            <supported-locale>de</supported-locale>
         </locale-config>
     </application>
   </faces-config>
   ```

   When a browser connects to your application, it usually includes an `Accept-Language` value in the HTTP header (see `http://www.w3.org/International/questions/qa-accept-lang-locales.html`). The JSF implementation reads the header and finds the best match among the supported locales. You can test this feature by setting the preferred language in your browser (see Figure 2–2).

2. You can add a `locale` attribute to the `f:view` element—for example,

   ```
   <f:view locale="de">
   ```

   The locale can be dynamically set:

   ```
   <f:view locale="#{user.locale}"/>>
   ```

   Now the locale is set to the string that the `getLocale` method returns. This is useful in applications that let the user pick a preferred locale.

3. You can set the locale programatically. Call the `setLocale` method of the `UIViewRoot` object:

   ```
   UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
   viewRoot.setLocale(new Locale("de"));
   ```

   See "Using Command Links" on page 125 of Chapter 4 for an example.

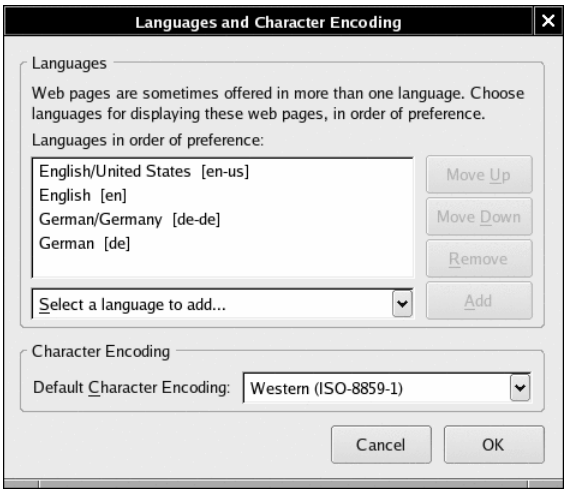

Figure 2–2 Selecting the preferred language

A Sample Application

After all these rather abstract rules and regulations, it is time for a concrete example. The application presents a series of quiz questions. Each question displays a sequence of numbers and asks the participant to guess the next number of the sequence.

For example, Figure 2–3 asks for the next number in the sequence

```
3 1 4 1 5
```

You often find puzzles of this kind in tests that purport to measure intelligence. To solve the puzzle, you need to find the pattern. In this case, we have the first digits of π.

Type in the next number in the sequence (9), and the score goes up by one.

NOTE: There is a Java-compatible mnemonic for the digits of π: "Can I have a small container of coffee?" Count the letters in each word, and you get 3 1 4 1 5 9 2 6. See http://dir.yahoo.com/Science/Mathematics/Numerical_Analysis/Numbers/Specific_Numbers/Pi/Mnemonics/ for more elaborate memorization aids.

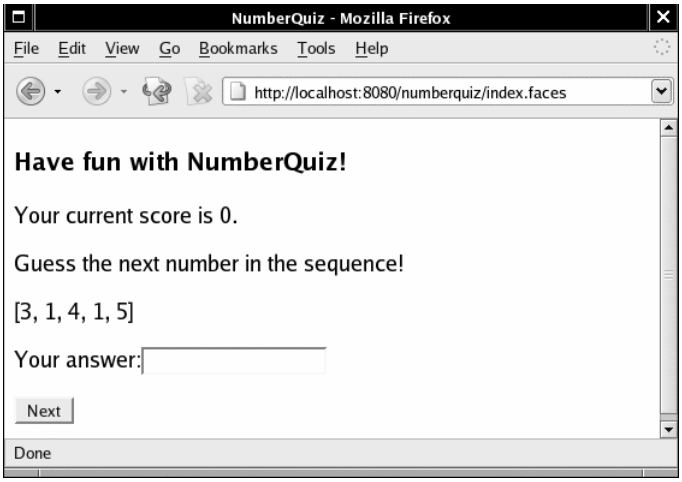

Figure 2–3 The number quiz

In this example, we place the quiz questions in the `QuizBean` class. Of course, in a real application, you would be more likely to store this information in a database. But the purpose of the example is to demonstrate how to use beans that have complex structure.

We start out with a `ProblemBean` class. A `ProblemBean` has two properties: `solution`, of type `int`, and `sequence`, of type `ArrayList` (see Listing 2–1).

Listing 2–1 numberquiz/src/java/com/corejsf/ProblemBean.java

```
1. package com.corejsf;
2. import java.util.ArrayList;
3.
4. public class ProblemBean {
5.    private ArrayList<Integer> sequence;
6.    private int solution;
7.
8.    public ProblemBean() {}
9.
10.   public ProblemBean(int[] values, int solution) {
11.      sequence = new ArrayList<Integer>();
12.      for (int i = 0; i < values.length; i++)
13.         sequence.add(values[i]);
14.      this.solution = solution;
15.   }
16.
```

Listing 2–1 numberquiz/src/java/com/corejsf/ProblemBean.java (cont.)

```
17.    // PROPERTY: sequence
18.    public ArrayList<Integer> getSequence() { return sequence; }
19.    public void setSequence(ArrayList<Integer> newValue) { sequence = newValue; }
20.
21.    // PROPERTY: solution
22.    public int getSolution() { return solution; }
23.    public void setSolution(int newValue) { solution = newValue; }
24. }
```

Next, we define a bean for the quiz with the following properties:

- `problems`: a write-only property to set the quiz problems
- `score`: a read-only property to get the current score
- `current`: a read-only property to get the current quiz problem
- `answer`: a property to get and set the answer that the user provides

The `problems` property is unused in this sample program—we initialize the problem set in the `QuizBean` constructor. However, under "Chaining Bean Definitions" on page 61, you will see how to set up the problem set inside `faces-config.xml`, without having to write any code.

The `current` property is used to display the current problem. However, the value of the `current` property is a `ProblemBean` object, and we cannot directly display that object in a text field. We make a second property access to get the number sequence:

```
<h:outputText value="#{quiz.current.sequence}"/>
```

The value of the `sequence` property is an `ArrayList`. When it is displayed, it is converted to a string by a call to the `toString` method. The result is a string of the form

```
[3, 1, 4, 1, 5]
```

Finally, we do a bit of dirty work with the `answer` property. We tie the `answer` property to the input field:

```
<h:inputText value="#{quiz.answer}"/>
```

When the input field is displayed, the getter is called, and we define the `getAnswer` method to return an empty string.

When the form is submitted, the setter is called with the value that the user typed into the input field. We define `setAnswer` to check the answer, update the score for a correct answer, and advance to the next problem.

```
public void setAnswer(String newValue) {
   try {
      int answer = Integer.parseInt(newValue.trim());
      if (getCurrent().getSolution() == answer) score++;
      currentIndex = (currentIndex + 1) % problems.size();
   }
   catch (NumberFormatException ex) {
   }
}
```

Strictly speaking, it is a bad idea to put code into a property setter that is unrelated to the task of setting the property. Updating the score and advancing to the next problem should really be contained in a handler for the button action. However, we have not yet discussed how to react to button actions, so we use the flexibility of the setter method to our advantage.

Another weakness of our sample application is that we have not yet covered how to stop at the end of the quiz. Instead, we just wrap around to the beginning, letting the user rack up a higher score. You will learn in the next chapter how to do a better job. Remember—the point of this application is to show you how to configure and use beans.

Finally, note that we use message bundles for internationalization. Try switching your browser language to German, and the program will appear as in Figure 2–4.

This finishes our sample application. Figure 2–5 shows the directory structure. The remaining code is in Listings 2–2 through 2–6.

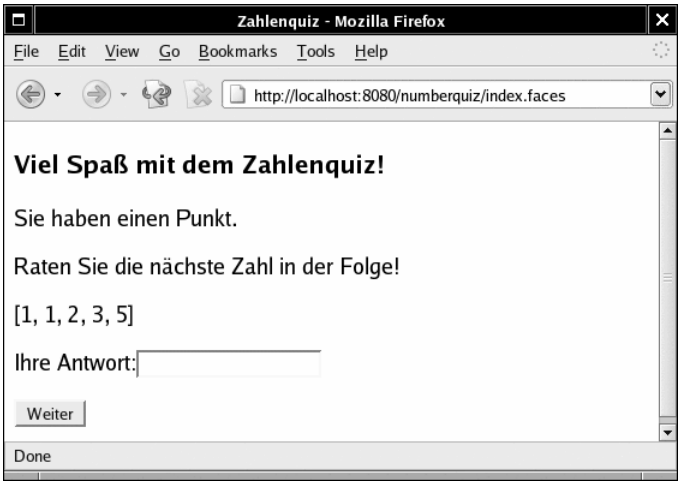

Figure 2–4 Viel Spaß mit dem Zahlenquiz!

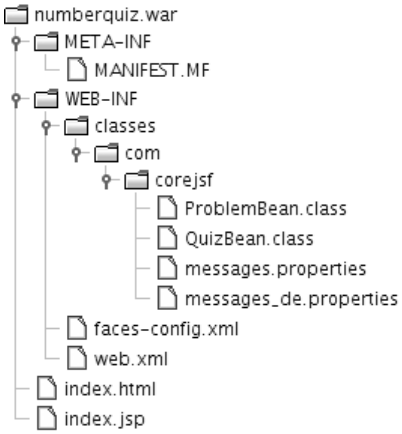

Figure 2–5 The directory structure of the number quiz example

Listing 2–2 numberquiz/web/index.jsp

```
1. <html>
2.    <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
3.    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
4.
5.    <f:view>
6.       <head>
7.          <title><h:outputText value="#{msgs.title}"/></title>
8.       </head>
9.       <body>
10.         <h:form>
11.            <h3>
12.               <h:outputText value="#{msgs.heading}"/>
13.            </h3>
14.            <p>
15.               <h:outputFormat value="#{msgs.currentScore}">
16.                  <f:param value="#{quiz.score}"/>
17.               </h:outputFormat>
18.            </p>
19.            <p>
20.               <h:outputText value="#{msgs.guessNext}"/>
21.            </p>
22.            <p>
23.               <h:outputText value="#{quiz.current.sequence}"/>
24.            </p>
```

Listing 2–2 numberquiz/web/index.jsp (cont.)

```
            <p>
               <h:outputText value="#{msgs.answer}"/>
               <h:inputText value="#{quiz.answer}"/></p>
            <p>
               <h:commandButton value="#{msgs.next}" action="next"/>
            </p>
         </h:form>
      </body>
   </f:view>
</html>
```

Listing 2–3 numberquiz/src/java/com/corejsf/QuizBean.java

```
package com.corejsf;
import java.util.ArrayList;

public class QuizBean {
   private ArrayList<ProblemBean> problems = new ArrayList<ProblemBean>();
   private int currentIndex;
   private int score;

   public QuizBean() {
      problems.add(
         new ProblemBean(new int[] { 3, 1, 4, 1, 5 }, 9)); // pi
      problems.add(
         new ProblemBean(new int[] { 1, 1, 2, 3, 5 }, 8)); // fibonacci
      problems.add(
         new ProblemBean(new int[] { 1, 4, 9, 16, 25 }, 36)); // squares
      problems.add(
         new ProblemBean(new int[] { 2, 3, 5, 7, 11 }, 13)); // primes
      problems.add(
         new ProblemBean(new int[] { 1, 2, 4, 8, 16 }, 32)); // powers of 2
   }

   // PROPERTY: problems
   public void setProblems(ArrayList<ProblemBean> newValue) {
      problems = newValue;
      currentIndex = 0;
      score = 0;
   }

   // PROPERTY: score
   public int getScore() { return score; }
```

Listing 2–3 numberquiz/src/java/com/corejsf/QuizBean.java (cont.)

```
31.
32.    // PROPERTY: current
33.    public ProblemBean getCurrent() {
34.       return problems.get(currentIndex);
35.    }
36.
37.    // PROPERTY: answer
38.    public String getAnswer() { return ""; }
39.    public void setAnswer(String newValue) {
40.       try {
41.          int answer = Integer.parseInt(newValue.trim());
42.          if (getCurrent().getSolution() == answer) score++;
43.          currentIndex = (currentIndex + 1) % problems.size();
44.       }
45.       catch (NumberFormatException ex) {
46.       }
47.    }
48. }
```

Listing 2–4 numberquiz/web/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2. <faces-config xmlns="http://java.sun.com/xml/ns/javaee"
3.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5.          http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
6.    version="1.2">
7.    <application>
8.       <locale-config>
9.          <default-locale>en</default-locale>
10.         <supported-locale>de</supported-locale>
11.      </locale-config>
12.   </application>
13.
14.   <navigation-rule>
15.      <from-view-id>/index.jsp</from-view-id>
16.      <navigation-case>
17.         <from-outcome>next</from-outcome>
18.         <to-view-id>/index.jsp</to-view-id>
19.      </navigation-case>
20.   </navigation-rule>
21.
```

Listing 2–4 numberquiz/web/WEB-INF/faces-config.xml (cont.)

```
22.    <managed-bean>
23.       <managed-bean-name>quiz</managed-bean-name>
24.       <managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
25.       <managed-bean-scope>session</managed-bean-scope>
26.    </managed-bean>
27.
28.    <application>
29.       <resource-bundle>
30.          <base-name>com.corejsf.messages</base-name>
31.          <var>msgs</var>
32.       </resource-bundle>
33.    </application>
34. </faces-config>
```

Listing 2–5 numberquiz/src/java/com/corejsf/messages.properties

```
1. title=NumberQuiz
2. heading=Have fun with NumberQuiz!
3. currentScore=Your current score is {0}.
4. guessNext=Guess the next number in the sequence!
5. answer=Your answer:
6. next=Next
```

Listing 2–6 numberquiz/src/java/com/corejsf/messsages_de.properties

```
1. title=Zahlenquiz
2. heading=Viel Spa\u00df mit dem Zahlenquiz!
3. currentScore=Sie haben {0,choice,0#0 Punkte|1#einen Punkt|2#{0} Punkte}.
4. guessNext=Raten Sie die n\u00e4chste Zahl in der Folge!
5. answer=Ihre Antwort:
6. next=Weiter
```

Backing Beans

Sometimes it is convenient to design a bean that contains some or all component objects of a web form. Such a bean is called a *backing bean* for the web form.

For example, we can define a backing bean for the quiz form by adding properties for the form component:

```
public class QuizFormBean {
   private UIOutput scoreComponent;
   private UIInput answerComponent;
```

```
    // PROPERTY: scoreComponent
    public UIOutput getScoreComponent() { return scoreComponent; }
    public void setScoreComponent(UIOutput newValue) { scoreComponent = newValue; }

    // PROPERTY: answerComponent
    public UIInput getAnswerComponent() { return answerComponent; }
    public void setAnswerComponent(UIInput newValue) { answerComponent = newValue; }
    ...
}
```

Output components belong to the `UIOutput` class and input components belong to the `UIInput` class. We discuss these classes in greater detail in "The Custom Component Developer's Toolbox" on page 360 of Chapter 9.

Why would you want such a bean? As we show in "Validating Relationships Between Multiple Components" on page 260 of Chapter 6, it is sometimes necessary for validators and event handlers to have access to the actual components on a form. Moreover, visual JSF development environments generally use backing beans (called *page beans* in Java Studio Creator because a bean is added for every page). These environments automatically generate the property getters and setters for all components that are dragged onto a form.

When you use a backing bean, you need to wire up the components on the form to those on the bean. You use the `binding` attribute for this purpose:

```
<h:outputText binding="#{quizForm.scoreComponent}"/>
```

When the component tree for the form is built, the `getScoreComponent` method of the backing bean is called, but it returns `null`. As a result, an output component is constructed and installed into the backing bean with a call to `setScoreComponent`.

Backing beans have their uses, but they can also be abused. You should not mix form components and business data in the same bean. If you use backing beans for your presentation data, use a different set of beans for business objects.

Bean Scopes

For the convenience of the web application programmer, a servlet container provides separate scopes, each of which manages a table of name/value bindings.

These scopes typically hold beans and other objects that need to be available in different components of a web application.

Session Scope

Recall that the HTTP protocol is *stateless*. The browser sends a request to the server, the server returns a response, and then neither the browser nor the

server has any obligation to keep any memory of the transaction. This simple arrangement works well for retrieving basic information, but it is unsatisfactory for server-side applications. For example, in a shopping application, you want the server to remember the contents of the shopping cart.

For that reason, servlet containers augment the HTTP protocol to keep track of a *session*—that is, repeated connections by the same client. There are various methods for session tracking. The simplest method uses *cookies*: name/value pairs that a server sends to a client, hoping to have them returned in subsequent requests (see Figure 2–6).

Figure 2–6 The cookie sent by a JSF application

As long as the client does not deactivate cookies, the server receives a session identifier with each subsequent request.

Application servers use fallback strategies, such as *URL rewriting*, for dealing with those clients that do not return cookies. URL rewriting adds a session identifier to a URL, which looks somewhat like this:

```
http://corejsf.com/login/index.jsp;jsessionid=b55cd6...d8e
```

NOTE: To see this behavior, tell your browser to reject cookies from the `localhost`, then restart the web application and submit a page. The next page will have a `jsessionid` attribute.

Session tracking with cookies is completely transparent to the web developer, and the standard JSF tags automatically perform URL rewriting if a client does not use cookies.

The *session scope* persists from the time that a session is established until session termination. A session terminates if the web application invokes the `invalidate` method on the `HttpSession` object, or if it times out.

Web applications typically place most of their beans into session scope.

For example, a `UserBean` can contain information about users that is accessible throughout the entire session. A `ShoppingCartBean` can be filled up gradually during the requests that make up a session.

Application Scope

The *application scope* persists for the entire duration of the web application. That scope is shared among all requests and all sessions.

You place managed beans into the application scope if a single bean should be shared among all instances of a web application. The bean is constructed when it is first requested by any instance of the application, and it stays alive until the web application is removed from the application server.

Request Scope

The *request scope* is short-lived. It starts when an HTTP request is submitted and ends when the response is sent back to the client.

If you place a managed bean into request scope, a new instance is created with each request. Not only is this is potentially expensive, it is also not appropriate if you want your data to persist beyond a request. You would place an object into request scope only if you wanted to forward it to another processing phase inside the current request.

For example, the `f:loadBundle` tag places the bundle variable in request scope. The variable is needed only during the *Render Response* phase in the same request.

CAUTION: Only request scope beans are single-threaded and, therefore, inherently threadsafe. Perhaps surprisingly, session beans are *not* single-threaded. For example, a user can simultaneously submit responses from multiple browser windows. Each response is processed by a separate request thread. If you need thread safety in your session beans, you should provide locking mechanisms.

Life Cycle Annotations

Starting with JSF 1.2, you can specify managed bean methods that are automatically called just after the bean has been constructed and just before the bean goes out of scope. This is particularly convenient for beans that establish connections to external resources such as databases.

Annotate the methods with `@PostConstruct` or `@PreDestroy`, like this:

```
public class MyBean {
   @PostConstruct
   public void initialize() {
      // initialization code
   }
   @PreDestroy
   public void shutdown() {
      // shutdown code
   }

   // other bean methods
}
```

These methods will be automatically called, provided the web application is deployed in a container that supports the annotations of JSR (Java Specification Request) 250 (see `http://www.jcp.org/en/jsr/detail?id=250`). In particular, Java EE 5-compliant application servers such as GlassFish support these annotations. It is expected that standalone containers such as Tomcat will also provide support in the future.

Configuring Beans

This section describes how you can configure a bean in a configuration file. The details are rather technical. You may want to have a glance at this section and return to it when you need to configure beans with complex properties.

The most commonly used configuration file is `WEB-INF/faces-config.xml`. However, you can also place configuration information inside the following locations:

- Files named `META-INF/faces-config.xml` inside any JAR files loaded by the external context's class loader. (You use this mechanism if you deliver reusable components in a JAR file.)
- Files listed in the `javax.faces.CONFIG_FILES` initialization parameter inside `WEB-INF/web.xml`. For example,

```
<web-app>
   <context-param>
      <param-name>javax.faces.CONFIG_FILES</param-name>
```

```
        <param-value>WEB-INF/navigation.xml,WEB-INF/beans.xml</param-value>
    </context-param>
    ...
</web-app>
```

(This mechanism is attractive for builder tools because it separates navigation, beans, etc.)

For simplicity, we use `WEB-INF/faces-config.xml` in this chapter.

A bean is defined with a `managed-bean` element inside the top-level `faces-config` element. Minimally, you must specify the name, class, and scope of the bean.

```
<faces-config>
   <managed-bean>
      <managed-bean-name>user</managed-bean-name>
      <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
   </managed-bean>
</faces-config>
```

The scope can be `request`, `session`, `application`, or `none`. The `none` scope denotes an object that is not kept in one of the three scope maps. You use objects with scope `none` as building blocks when wiring up complex beans. You will see an example in the section "Chaining Bean Definitions" on page 61.

Setting Property Values

We start with a simple example. Here we customize a `UserBean` instance:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>name</property-name>
      <value>me</value>
   </managed-property>
   <managed-property>
      <property-name>password</property-name>
      <value>secret</value>
   </managed-property>
</managed-bean>
```

When the `user` bean is first looked up, it is constructed with the `UserBean()` default constructor. Then the `setName` and `setPassword` methods are executed.

To initialize a property with `null`, use a `null-value` element. For example,

```
<managed-property>
   <property-name>password</property-name>
   <null-value/>
</managed-property>
```

Initializing Lists and Maps

A special syntax initializes values that are of type `List` or `Map`. Here is an example of a list:

```
<list-entries>
   <value-class>java.lang.Integer</value.class>
   <value>3</value>
   <value>1</value>
   <value>4</value>
   <value>1</value>
   <value>5</value>
</list-entries>
```

Here we use the `java.lang.Integer` wrapper type since a `List` cannot hold values of primitive type.

The list can contain a mixture of `value` and `null-value` elements. The `value-class` is optional. If it is omitted, a list of `java.lang.String` objects is produced.

A map is more complex. You specify optional `key-class` and `value-class` elements (again, with a default of `java.lang.String`). Then you provide a sequence of `map-entry` elements, each of which has a `key` element, followed by a `value` or `null-value` element.

Here is an example:

```
<map-entries>
   <key-class>java.lang.Integer</key-class>
   <map-entry>
      <key>1</key>
      <value>George Washington</value>
   </map-entry>
   <map-entry>
      <key>3</key>
      <value>Thomas Jefferson</value>
   </map-entry>
   <map-entry>
      <key>16</key>
      <value>Abraham Lincoln</value>
   </map-entry>
   <map-entry>
```

```
        <key>26</key>
        <value>Theodore Roosevelt</value>
      </map-entry>
    </map-entries>
```

You can use `list-entries` and `map-entries` elements to initialize either a `managed-bean` or a `managed-property`, provided that the bean or property type is a `List` or `Map`.

Figure 2–7 shows a *syntax diagram* for the `managed-bean` element and all of its child elements. Follow the arrows to see which constructs are legal inside a `managed-bean` element. For example, the second graph tells you that a `managed-property` element starts with zero or more `description` elements, followed by zero or more `display-name` elements, zero or more `icons`, then a mandatory `property-name`, an optional `property-class`, and exactly one of the elements `value`, `null-value`, `map-entries`, or `list-entries`.

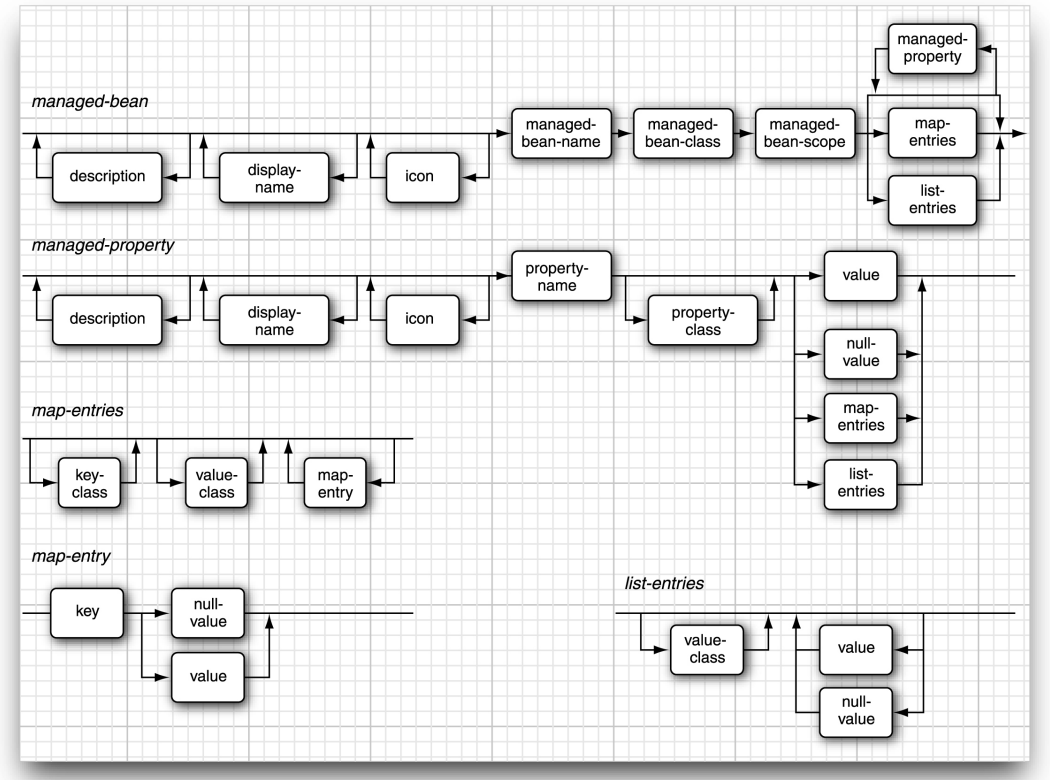

Figure 2–7 Syntax diagram for `managed-bean` elements

Chaining Bean Definitions

You can achieve more complex arrangements by using value expressions inside the `value` element to chain beans together. Consider the quiz bean in the `number-quiz` application.

The quiz contains a collection of problems, represented as the write-only `problems` property. You can configure it with the following instructions:

```
<managed-bean>
   <managed-bean-name>quiz</managed-bean-name>
   <managed-bean-class>com.corejsf.QuizBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>problems</property-name>
      <list-entries>
         <value-class>com.corejsf.ProblemBean</value-class>
         <value>#{problem1}</value>
         <value>#{problem2}</value>
         <value>#{problem3}</value>
         <value>#{problem4}</value>
         <value>#{problem5}</value>
      </list-entries>
   </managed-property>
</managed-bean>
```

Of course, now we must define beans with names `problem1` through `problem5`, like this:

```
<managed-bean>
   <managed-bean-name>problem1</managed-bean-name>
   <managed-bean-class>
      com.corejsf.ProblemBean
   </managed-bean-class>
   <managed-bean-scope>none</managed-bean-scope>
      <managed-property>
         <property-name>sequence</property-name>
         <list-entries>
            <value-class>java.lang.Integer</value-class>
            <value>3</value>
            <value>1</value>
            <value>4</value>
            <value>1</value>
            <value>5</value>
         </list-entries>
      </managed-property>
```

```
        <managed-property>
           <property-name>solution</property-name>
           <value>9</value>
        </managed-property>
  </managed-bean>
```

When the `quiz` bean is requested, then the creation of the beans `problem1` through `problem5` is triggered automatically. You need not worry about the order in which you specify managed beans.

Note that the problem beans have scope `none` since they are never requested from a JSF page but are instantiated when the `quiz` bean is requested.

When you wire beans together, make sure that their scopes are compatible. Table 2–1 lists the permissible combinations.

Table 2–1 Compatible Bean Scopes

When defining a bean of this scope . . .	. . . you can use beans of these scopes
`none`	`none`
`application`	`none`, `application`
`session`	`none`, `application`, `session`
`request`	`none`, `application`, `session`, `request`

String Conversions

You specify property values and elements of lists or maps with a `value` element that contains a string. The enclosed string needs to be converted to the type of the property or element. For primitive types, this conversion is straightforward. For example, you can specify a `boolean` value with the string `true` or `false`.

Starting with JSF 1.2, values of enumerated types are supported as well. The conversion is performed by calling `Enum.valueOf(`*propertyClass*`,` *valueText*`)`.

For other property types, the JSF implementation attempts to locate a matching `PropertyEditor`. If a property editor exists, its `setAsText` method is invoked to convert strings to property values. Property editors are heavily used for client-side beans, to convert between property values and a textual or graphical representation that can be displayed in a property sheet (see Figure 2–8).

Defining a property editor is somewhat involved, and we refer the interested reader to Horstmann and Cornell, 2004, 2005. *Core Java™ 2*, vol. 2, chap. 8.

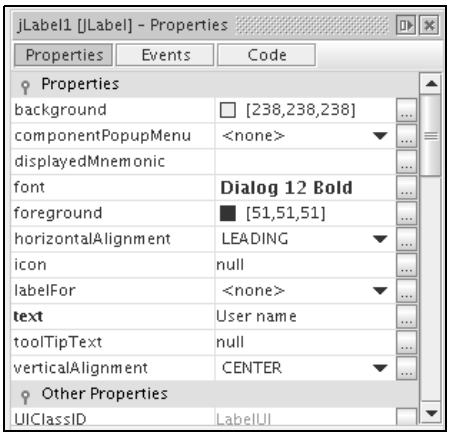

Figure 2–8 A property sheet in a GUI builder

Note that the rules are fairly restrictive. For example, if you have a property of type `URL`, you cannot simply specify the URL as a string, even though there is a constructor `URL(String)`. You would need to supply a property editor for the `URL` type or reimplement the property type as `String`.

Table 2–2 summarizes these conversion rules. They are identical to the rules for the `jsp:setProperty` action of the JSP specification.

Table 2–2 String Conversions

Target Type	Conversion
`int`, `byte`, `short`, `long`, `float`, `double`, or the corresponding wrapper type	The `valueOf` method of the wrapper type, or 0 if the string is empty.
`boolean` or `Boolean`	The result of `Boolean.valueOf`, or `false` if the string is empty.
`char` or `Character`	The first character of the string, or `(char) 0` if the string is empty.
`String` or `Object`	A copy of the string; `new String("")` if the string is empty.
bean property	A type that calls the `setAsText` method of the property editor if it exists. If the property editor does not exist or it throws an exception, the property is set to `null` if the string is empty. Otherwise, an error occurs.

The Syntax of Value Expressions

In this section, we discuss the syntax for value expressions in gruesome detail. This section is intended for reference. Feel free to skip it at first reading.

We start with an expression of the form `a.b`. For now, we will assume that we already know the object to which `a` refers. If `a` is an array, a list, or a map, then special rules apply (see "Using Brackets" below). If `a` is any other object, then `b` must be the name of a property of `a`. The exact meaning of `a.b` depends on whether the expression is used in *rvalue mode* or *lvalue mode*.

This terminology is used in the theory of programming languages to denote that an expression on the *right-hand side* of an assignment is treated differently from an expression on the *left-hand side*.

Consider the assignment

```
left = right;
```

A compiler generates different code for the `left` and `right` expressions. The `right` expression is evaluated in rvalue mode and yields a value. The `left` expression is evaluated in lvalue mode and stores a value in a location.

The same phenomenon happens when you use a value expression in a user interface component:

```
<h:inputText value="#{user.name}"/>
```

When the text field is rendered, the expression `user.name` is evaluated in rvalue mode, and the `getName` method is called. During decoding, the same expression is evaluated in lvalue mode and the `setName` method is called.

In general, the expression `a.b` in rvalue mode is evaluated by calling the property getter, whereas `a.b` in lvalue mode calls the property setter.

Using Brackets

Just as in JavaScript, you can use brackets instead of the dot notation. That is, the following three expressions all have the same meaning:

```
a.b
a["b"]
a['b']
```

For example, `user.password`, `user["password"]`, and `user['password']` are equivalent expressions.

Why would anyone write `user["password"]` when `user.password` is much easier to type? There are a number of reasons:

- When you access an array or map, the `[]` notation is more intuitive.
- You can use the `[]` notation with strings that contain periods or dashes—for example, `msgs["error.password"]`.
- The `[]` notation allows you to dynamically compute a property: `a[b.propname]`.

TIP: Use single quotes in value expressions if you delimit attributes with double quotes: `value="#{user['password']}"`. Alternatively, you can switch single and double quotes: `value='#{user["password"]}'`.

Map and List Expressions

The value expression language goes beyond bean property access. For example, let `m` be an object of any class that implements the `Map` interface. Then `m["key"]` (or the equivalent `m.key`) is a binding to the associated value. In rvalue mode, the value

```
m.get("key")
```

is fetched. In lvalue mode, the statement

```
m.put("key", right);
```

is executed. Here, `right` is the *right-hand side* value that is assigned to `m.key`.

You can also access a value of any object of a class that implements the `List` interface (such as an `ArrayList`). You specify an integer index for the list position. For example, `a[i]` (or, if you prefer, `a.i`) binds the `i`th element of the list `a`. Here `i` can be an integer, or a string that can be converted to an integer. The same rule applies for array types. As always, index values start at zero.

Table 2–3 summarizes these evaluation rules.

Table 2–3 Evaluating the Value Expression `a.b`

Type of `a`	Type of `b`	lvalue Mode	rvalue Mode
`null`	any	error	`null`
any	`null`	error	`null`
`Map`	any	`a.put(b, right)`	`a.get(b)`

Table 2–3 Evaluating the Value Expression `a.b` (cont.)

Type of `a`	Type of `b`	lvalue Mode	rvalue Mode
`List`	convertible to `int`	`a.set(b, right)`	`a.get(b)`
array	convertible to `int`	`a[b] = right`	`a[b]`
bean	any	call setter of property with name `b.toString()`	call getter of property with name `b.toString()`

CAUTION: Unfortunately, value expressions do not work for indexed properties. If `p` is an indexed property of a bean `b`, and `i` is an integer, then `b.p[i]` does not access the `i`th value of the property. It is simply a syntax error. This deficiency is inherited from the JSP expression language.

Resolving the Initial Term

Now you know how an expression of the form `a.b` is resolved. The rules can be applied repetitively to expressions such as `a.b.c.d` (or, of course, `a['b'].c["d"]`). We still need to discuss the meaning of the initial term `a`.

In the examples you have seen so far, the initial term referred to a bean that was configured in the `faces-config.xml` file or to a message bundle map. Those are indeed the most common situations. But it is also possible to specify other names.

There are a number of predefined objects. Table 2–4 shows the complete list. For example,

```
header['User-Agent']
```

is the value of the `User-Agent` parameter of the HTTP request that identifies the user's browser.

If the initial term is not one of the predefined objects, the JSF implementation looks for it in the *request*, *session*, and *application scopes*, in that order. Those scopes are map objects that are managed by the servlet container. For example, when you define a managed bean, its name and value are added to the appropriate scope map.

Table 2–4 Predefined Objects in the Value Expression Language

Variable Name	Meaning
`header`	A `Map` of HTTP header parameters, containing only the first value for each name.
`headerValues`	A `Map` of HTTP header parameters, yielding a `String[]` array of all values for a given name.
`param`	A `Map` of HTTP request parameters, containing only the first value for each name.
`paramValues`	A `Map` of HTTP request parameters, yielding a `String[]` array of all values for a given name.
`cookie`	A `Map` of the cookie names and values of the current request.
`initParam`	A `Map` of the initialization parameters of this web application. Initialization parameters are discussed in Chapter 10.
`requestScope`	A `Map` of all request scope attributes.
`sessionScope`	A `Map` of all session scope attributes.
`applicationScope`	A `Map` of all application scope attributes.
`facesContext`	The `FacesContext` instance of this request. This class is discussed in Chapter 6.
`view`	The `UIViewRoot` instance of this request. This class is discussed in Chapter 7.

Finally, if the name is still not found, it is passed to the `VariableResolver` of the JSF application. The default variable resolver looks up `managed-bean` elements in a configuration resource, typically the `faces-config.xml` file.

Consider, for example, the expression

```
#{user.password}
```

The term `user` is not one of the predefined objects. When it is encountered for the first time, it is not an attribute name in request, session, or application scope.

Therefore, the variable resolver processes the `faces-config.xml` entry:

```
<managed-bean>
   <managed-bean-name>user</managed-bean-name>
   <managed-bean-class>com.corejsf.UserBean</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The variable resolver calls the default constructor of the class `com.corejsf.UserBean`. Next, it adds an association to the `sessionScope` map. Finally, it returns the object as the result of the lookup.

When the term `user` needs to be resolved again in the same session, it is located in the session scope.

Composite Expressions

You can use a limited set of operators inside value expressions:

- Arithmetic operators `+ - * / %`. The last two operators have alphabetic variants `div` and `mod`.
- Relational operators `< <= > >= == !=` and their alphabetic variants `lt le gt ge eq ne`. The first four variants are required for XML safety.
- Logical operators `&& || !` and their alphabetic variants `and or not`. The first variant is required for XML safety.
- The `empty` operator. The expression `empty a` is `true` if `a` is `null`, an array or `String` of length 0, or a `Collection` or `Map` of size 0.
- The ternary `?:` selection operator.

Operator precedence follows the same rules as in Java. The `empty` operator has the same precedence as the unary - and `!` operators.

Generally, you do not want to do a lot of expression computation in web pages—that would violate the separation of presentation and business logic. However, occasionally, the presentation layer can benefit from operators. For example, suppose you want to hide a component when the `hide` property of a bean is true. To hide a component, you set its `rendered` attribute to `false`. Inverting the bean value requires the `!` (or `not`) operator:

```
<h:inputText rendered="#{!bean.hide}" ... />
```

Finally, you can concatenate plain strings and value expressions by placing them next to each other. Consider, for example,

```
<h:outputText value="#{messages.greeting}, #{user.name}!"/>
```

The statement concatenates four strings: the string returned from `#{messages.greeting}`, the string consisting of a comma and a space, the string returned from `#{user.name}`, and the string consisting of an exclamation mark.

You have now seen all the rules that are applied to resolve value expressions. Of course, in practice, most expressions are of the form `#{bean.property}`. Just come back to this section when you need to tackle a more complex expression.

Method Expressions

A *method expression* denotes an object, together with a method that can be applied to it.

For example, here is a typical use of a method expression:

```
<h:commandButton action="#{user.checkPassword}"/>
```

We assume that `user` is a value of type `UserBean` and `checkPassword` is a method of that class. The method expression is a convenient way of describing a method invocation that needs to be carried out at some future time.

When the expression is evaluated, the method is applied to the object.

In our example, the command button component will call `user.checkPassword()` and pass the returned string to the navigation handler.

Syntax rules for method expressions are similar to those of value expressions. All but the last component are used to determine an object. The last component must be the name of a method that can be applied to that object.

Four component attributes can take a method expression:

- `action` (see "Dynamic Navigation" on page 73 of Chapter 3)
- `validator` (see "Validating with Bean Methods" on page 259 of Chapter 6)
- `valueChangeListener` ("Value Change Events" on page 269 of see Chapter 7)
- `actionListener` (see "Action Events" on page 275 of Chapter 7)

The parameter and return types of the method depend on the context in which the method expression is used. For example, an `action` must be bound to a method with no parameters and return type `String`, whereas an `actionListener` is bound to a method with one parameter of type `ActionEvent` and return type `void`. The code that invokes the method expression is responsible for supplying parameter values and processing the return value.