

Developing JSP components

This chapter covers

- The JavaBeans API
- Developing your own JSP components
- Mixing scriptlets and Beans

This chapter will help developers create their own JavaBeans for use as JSP components, and teach web designers how they are implemented behind the scenes. Fortunately, it is not necessary to understand all of the details of JavaBeans development to work with JSP. As component architectures go, the interface between JavaServer Pages and JavaBeans is quite simple, as we will see.

8.1 **What makes a bean a bean?**

So what makes a bean so special? A bean is simply a Java class that follows a set of simple naming and design conventions outlined by the JavaBeans specification. Beans are not required to extend a specific base class or implement a particular interface. If a class follows these bean conventions, and you treat it like a bean—then it is a bean. A particularly good thing about the bean conventions is that they are rooted in sound programming practices that you may already be following to some extent.

8.1.1 **Bean conventions**

The JavaBean conventions are what enable us to develop beans because they allow a bean container to analyze a Java class file and interpret its methods as properties, designating the class as a JavaBean. The conventions dictate rules for defining a bean's constructor and the methods that will define its properties.

The JavaBeans API

Following the conventions specified by the JavaBeans API allows the JSP container to interact with beans at a programmatic level, even though the containing application has no real understanding of what the bean does or how it works. For JSP we are primarily concerned with the aspects of the API that dictate the method signatures for a bean's constructors and property access methods.

Beans are just objects

Like any other Java class, instances of bean classes are simply Java objects. As a result, you always have the option of referencing beans and their methods directly through Java code in other classes or through JSP scripting elements. Because they follow the JavaBeans conventions, we can work with them a lot easier than by writing Java code. Bean containers, such as a JSP container, can provide easy access to beans and their properties. Following the JavaBeans API coding conventions, as we will see, means creating methods that control access to each property we wish to define for our bean. Beans can also have regular methods like any other Java object. However,

JSP developers will have to use scriptlets, expressions, or custom tags to access them since a bean container can manipulate a bean only through its properties.

Class naming conventions

You might have noticed that in most of our examples bean classes often include the word *bean* in their name, such as `UserBean`, `AlarmClockBean`, `DataAccessBean`, and so forth. While this is a common approach that lets other developers immediately understand the intended role of the class, it is not a requirement for a bean to be used inside a JSP page or any other bean container. Beans follow the same class-naming rules as other Java classes: they must start with an alphabetic character, contain only alphanumeric and underscore characters, and be case sensitive. Additionally, like other Java classes it is common, but not required, to start the name of a bean class with a capital letter.

The magic of introspection

How can the JSP container interact with any bean object without the benefit of a common interface or base class to fall back on? Java manages this little miracle through a process called *introspection* that allows a class to expose its methods and capabilities on request. The introspection process happens at run time, and is controlled by the bean container. It is introspection that allows us to rely on conventions to establish properties.

Introspection occurs through a mechanism known as *reflection*, which allows the bean container to examine any class at run time to determine its method signatures. The bean container determines what properties a bean supports by analyzing its public methods for the presence of methods that meet criteria defined by the JavaBeans API. For a property to exist, its bean class must define an access method to return the value of the property, change the value of the property, or both. It is the presence of these specially named access methods alone that determine the properties of a bean class, as we will soon see.

8.1.2 The bean constructor

The first rule of JSP bean building is that you must implement a constructor that takes no arguments. It is this constructor that the JSP container will use to instantiate your bean through the `<jsp:useBean>` tag. Every Java class has a constructor method that is used to create instances of the class. If a class does not explicitly specify any constructors, then a default zero-argument constructor is assumed. Because of this default constructor rule the following Java class is perfectly valid, and technically satisfies the bean conventions:

```
public class DoNothingBean { }
```

This bean has no properties and can't do or report anything useful, but it is a bean nonetheless. We can create new instances of it, reference it from scriptlets, and control its scope. Here is a better example of a class suitable for bean usage, a bean which knows the time. This class has a zero-argument constructor that records the time of its instantiation:

```
package com.taglib.wdjsp.components;
import java.util.*;

public class CurrentTimeBean {
    private int hours;
    private int minutes;

    public CurrentTimeBean() {
        Calendar now = Calendar.getInstance();
        this.hours = now.get(Calendar.HOUR_OF_DAY);
        this.minutes = now.get(Calendar.MINUTE);
    }
}
```

We've used the constructor to initialize the bean's instance variables `hours` and `minutes` to reflect the current time at instantiation. The constructor of a bean is the appropriate place to initialize instance variables and prepare the instance of the class for use. Of course to be useful within a JSP page we will need to define some properties for the bean and create the appropriate access methods to control them.

8.1.3 Defining a bean's properties

As we've mentioned, a bean's properties are defined simply by creating appropriate access methods for them. Access methods are used either to retrieve a property's value or make changes to it. A method used to retrieve a property's value is called a *getter* method, while a method that modifies its value is called a *setter* method. Together these methods are generally referred to as *access methods*—they provide access to values stored in the bean's properties.

To define properties for a bean simply create a `public` method with the name of the property you wish to define, prefixed with the word `get` or `set` as appropriate. Getter methods should return the appropriate data type, while the corresponding setter method should be declared `void` and accept one argument of the appropriate type. It is the `get` or `set` prefix that is Java's clue that you are defining a property. The signature for property access methods, then, is:

```
public void setPropertyName(PropertyType value);
public PropertyType getPropertyName();
```

For example, to define a property called `rank`, which can be used to store text, and is both readable and writable, we would need to create methods with these signatures:

```
public void setRank(String rank);
public String getRank();
```

Likewise, to create a property called `age` that stores numbers:

```
public void setAge(int age);
public int getAge();
```

NOTE Making your property access methods `public` is more than a good idea, it's the law! Exposing your bean's access methods by declaring them `public` is the only way that JSP pages will be able to call them. The JSP container will not recognize properties without `public` access methods.

Conversely, if the actual data being reflected by the component's properties is stored in instance variables it should be purposely hidden from other classes. Such instance variables should be declared `private` or at least `protected`. This helps ensure that developers restrict their interaction with the class to its access methods and not its internal workings. Otherwise, a change to the implementation might negatively impact code dependent on the older version of the component.

Let's revisit our previous example and make it more useful. We will add a couple of properties to our `CurrentTimeBean` called `hours` and `minutes`, that will allow us to reference the current time in the page. These properties must meet the getter method signatures defined by the JavaBeans design patterns. They therefore should look like this:

```
public int getHours();
public int getMinutes();
```

In our constructor we store the current time's hours and minutes into instance variables. We can have our properties reference these variables and return their value where appropriate. The source for this bean is shown in listing 8.1.

Listing 8.1 `CurrentTimeBean.java`

```
package com.taglib.wdjsp.components;
import java.util.*;

public class CurrentTimeBean {
    private int hours;
```

```
private int minutes;

public CurrentTimeBean() {
    Calendar now = Calendar.getInstance();
    this.hours = now.get(Calendar.HOUR_OF_DAY);
    this.minutes = now.get(Calendar.MINUTE);
}

public int getHours() {
    return hours;
}

public int getMinutes() {
    return minutes;
}
}
```

That's all there is to it. These two methods simply return the appropriate values as stored in the instance variables. Since they meet the JavaBean rules for naming access methods, we have just defined two properties that we can access through JSP Bean tags. For example:

```
<jsp:useBean id="time" class="CurrentTimeBean"/>
<html><body>
It is now <jsp:getProperty name="time" property="minutes"/>
minutes past the hour.
</body></html>
```

Properties should not be confused with instance variables, even though instance variables are often mapped directly to property names but properties of a bean are not required to correspond directly with instance variables. A bean's properties are defined by the method names themselves, not the variables or implementation behind them. This leaves the bean designer free to alter the inner workings of the bean without altering the interface and collection of properties that you expose to users of the bean.

As an example of dynamically generating property values, here is a bean that creates random numbers in its property access methods rather than simply returning a copy of an instance variable. Its code is shown in listing 8.2.

Listing 8.2 DiceBean.java

```
package com.taglib.wdjsp.components;
import java.util.*;

public class DiceBean {
    private Random rand;
    public DiceBean() {
```

```
        rand = new Random();
    }

    public int getDieRoll() {
        // return a number between 1 and 6
        return rand.nextInt(6) + 1;
    }

    public int getDiceRoll() {
        // return a number between 2 and 12
        return getDieRoll() + getDieRoll();
    }
}
```

In this example, our `dieRoll` and `diceRoll` properties are not managed by instance variables. Instead, we create a `java.util.Random` object in the constructor and call its random number generator from our access methods to dynamically generate property values. In fact, nowhere in the bean are any static values stored for these properties—their values are recomputed each time the properties are requested.

You are not required to create both getter and setter methods for each property you wish to provide for a bean. If you wish to make a property read-only then define a getter method without providing a corresponding setter method. Conversely creating only a setter method specifies a write-only property. The latter might be useful if the bean uses the property value internally to affect other properties but is not a property that you want clients manipulating directly.

Property name conventions

A common convention is that property names are mixed case, beginning with a lowercase letter and uppercasing the first letter of each word in the property name. For the properties `firstName` and `lastName` for example, the corresponding getter methods would be `getFirstName()` and `getLastName()`. Note the case difference between the property names and their access methods. Not to worry, the JSP container is smart enough to convert the first letter to uppercase when constructing the target getter method. If the first two or more letters of a property name are uppercased, for example `URL`, then the JSP container assumes that you really mean it, so its corresponding access methods would be `getURL()` and `setURL()`.

TIP **Naming Properties**—One situation that often leads to confusing property names is acronyms. For example consider a property representing an identification number. It could get be `getId` or `getID`, making the bean property `id` or `ID`. This leads to more confusion (and ugly method names) when you combine acronyms with additional words using capitalization of their own.

For example something like an accessor for an XML document, is that `getXMLDocument` or `getXmlDocument`? Is the property name `xmlDocument`, `XMLDocument`, or `XmlDocument`? To keep down confusion and improve consistency, you should only capitalize the first letter of acronyms. Without this rule teams tend to end up with several variations for the same basic property throughout their code base. It is first and foremost consistent and predictable and also clearly deliniates multiple word property names through capitalization. So a property method representing a Social Security number is immediately understood to be `getUserSsn` with a property name of `userSsn`. It may look funny, but you'll be amazed how much confusion it avoids.

8.1.4 Indexed properties

Bean properties are not limited to single values. Beans can also contain multivalued properties. For example, you might have a property named `contacts` that is used to store a list of objects of type `Contact`, containing phone and address information. Such a property would be used in conjunction with scriptlets or a custom iteration tag to step through the individual values. Each value must be of the same type; a single indexed property cannot contain both string and integer elements, for example.

To define an indexed valued property you have two options. The first style is creating an access method that returns the entire set of properties as a single array. In this case, a JSP page author or iterative custom tag can determine the size of the set and iterate through it. For example:

```
public PropertyType[] getProperty()
```

In the second option, you can access elements of the set by using an index value. This allows you additional flexibility. For example you might want to access only particular contacts from the collection.

```
public PropertyType getProperty(int index)
```

While not specifically required by JavaBean conventions, it is useful to implement both styles for a multivalued property. It's not much more work and it adds a good deal more flexibility in using the bean.

To set multivalue properties there are setter method signatures analogous to the getter method naming styles described earlier. The syntax for these methods is:

```
public void setProperty(int index, PropertyType value)
public void setProperty(PropertyType[] values)
```

Another type of method commonly implemented and recognized by bean containers is the `size()` method that can be used to determine the size of an indexed property. A typical implementation would be:

```
public int getPropertySize()
```

This is another method that is not required but increases the flexibility of the design to give page developers more options with which to work.

Example: a bean with indexed properties

In this example we will build a component that can perform statistical calculations on a series of numbers. The numbers themselves are stored in a single, indexed property. Other properties of the bean hold the value of statistical calculations like the average or the sum. This `StatBean`'s source code is shown in listing 8.3:

Listing 8.3 StatBean.java

```
package com.taglib.wdjsp.components;
import java.util.*;

public class StatBean {
    private double[] numbers;

    public StatBean() {
        numbers = new double[2];
        numbers[0] = 1;
        numbers[1] = 2;
    }

    public double getAverage() {
        double sum = 0;
        for (int i=0; i < numbers.length; i++)
            sum += numbers[i];
        return sum/numbers.length;
    }

    public double[] getNumbers() {
        return numbers;
    }

    public double getNumbers(int index) {
        return numbers[index];
    }

    public void setNumbers(double[] numbers) {
        this.numbers = numbers;
    }

    public void setNumbers(int index, double value) {
        numbers[index] = value;
    }
}
```

```
    }  
    public int getNumbersSize() {  
        return numbers.length;  
    }  
}
```

Since the JSP bean tags deal exclusively with scalar properties, the only way to interact with indexed properties such as these is through JSP scriptlets and expressions. In this JSP page we'll use a JSP scriptlet in the body of the `<jsp:useBean>` tag to pass an array of integers to the bean's `numbers` property. We'll have to use a scriptlet to display back the numbers themselves, but we can use a `<jsp:getProperty>` tag to display the average. The page is shown in listing 8.4:

Listing 8.4 stats.jsp

```
<jsp:useBean id="stat" class="com.taglib.wdjsp.StatBean">  
    <%  
        double[] mynums = {100, 250, 150, 50, 450};  
        stat.setNumbers(mynums);  
    %>  
</jsp:useBean>  
<html>  
<body>  
The average of  
<%  
double[] numbers = stat.getNumbers();  
for (int i=0; i < numbers.length; i++) {  
    if (i != numbers.length)  
        out.print(numbers[i] + ",");  
    else  
        out.println(" + numbers[i]);  
}  
%>  
is equal to  
<jsp:getProperty name="stat" property="average"/>  
</body>  
</html>
```

The use of custom tags, a technique that we will discuss in chapters 18 and 19, can greatly aid in working with indexed properties by eliminating the need for inline code by encapsulating common functionality into simple tag elements. With custom tags, we could eliminate the need for Java code in this example. We can also move this code inside the bean, which is what we'll do for now.

Accessing indexed values through JSP bean tags

We might also want to include a method that will enable us to pass in the array of numbers through a standard bean tag. Since bean tags deal exclusively with single values, we will have to perform the conversion ourselves in the property access methods. We'll create another pair of access methods that treat the array as a list of numbers stored in a comma delimited string. To differentiate between these two approaches, we will map the `String` versions of our new access methods to a new property we will call `numbersList`. Note that even though we are using a different property name, it is still modifying the same internal data, and will cause changes in the `average` and `numbers` properties. (Another example of this technique can be found in the Whois example of chapter 17.)

```
public void setNumbersList(String values) {
    Vector n = new Vector();
    StringTokenizer tok = new StringTokenizer(values, ",");
    while (tok.hasMoreTokens())
        n.addElement(tok.nextToken());
    numbers = new double[n.size()];
    for (int i=0; i < numbers.length; i++)
        numbers[i] = Double.parseDouble((String) n.elementAt(i));
}

public String getNumbersList() {
    String list = new String();
    for (int i=0; i < numbers.length; i++) {
        if (i != (numbers.length - 1))
            list += numbers[i] + ",";
        else
            list += "" + numbers[i];
    }
    return list;
}
```

Now we can access this bean through JSP tags alone, as shown in listing 8.5.

Listing 8.5 stats2.jsp

```
<jsp:useBean id="stat" class="com.taglib.wdjsp.components.StatBean">
  <jsp:setProperty name="stat" property="numbersList" value="100,250,150,50,450" />
</jsp:useBean>
<html>
<body>
The average of <jsp:getProperty name="stat" property="numbersList" />
is equal to
<jsp:getProperty name="stat" property="average" />
</body>
</html>
```

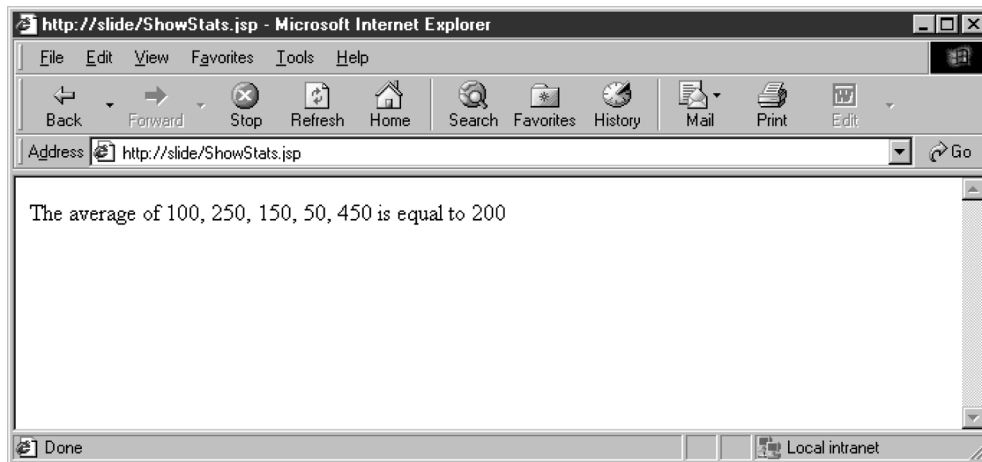


Figure 8.1 The ShowStat's page in action

The resulting display is shown in figure 8.1.

8.1.5 Implementing bean properties as cursors

Another technique for exposing the indexed properties of beans is creating a cursor. If you are familiar with JDBC's `ResultSet` class, or the `CachedRowSet` class of JDBC 2.0, then you can probably guess where we're headed. The idea here is to move the index inside the bean class as an instance variable, allowing us to access each indexed property through the `<jsp:getProperty>` tags by simply iterating the index. We provide a `next()` method which increments the index, returning `false` when the index counter has gone past the end of the list. This greatly reduces the amount of scriptlet code in the page, without introducing the complexity of custom tags. Here's an example of a page using this technique to display a table of the planets and their moons. The `PlanetBean` referenced in the page is shown in listing 8.7 and the resulting display is shown in figure 8.2.

Listing 8.6 planets.jsp

```
<html>
<body bgcolor="white">
<jsp:useBean id="planet" class="wdjsp.PlanetBean"/>
<table border="1">
<tr><th>Planet</th> <th>Number of Moons</th></tr>
<% while (planet.next()) { %>
<tr><td><jsp:getProperty name="planet" property="name"/></td>
```

```
<td align="center"><jsp:getProperty name="planet" property="moons"/></td></tr>
<% } %>
</table>
</body>
</html>
```

Listing 8.7 PlanetBean.java

```
package wdjsp;

public class PlanetBean {
    private static final int numPlanets = 9;
    private static final String[] names = {
        "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus",
        "Neptune", "Pluto" };
    private static final int[] moons =
        { 0, 0, 1, 2, 16, 18, 20, 8, 1 };

    private int index;

    public PlanetBean() {
        index = -1;
    }

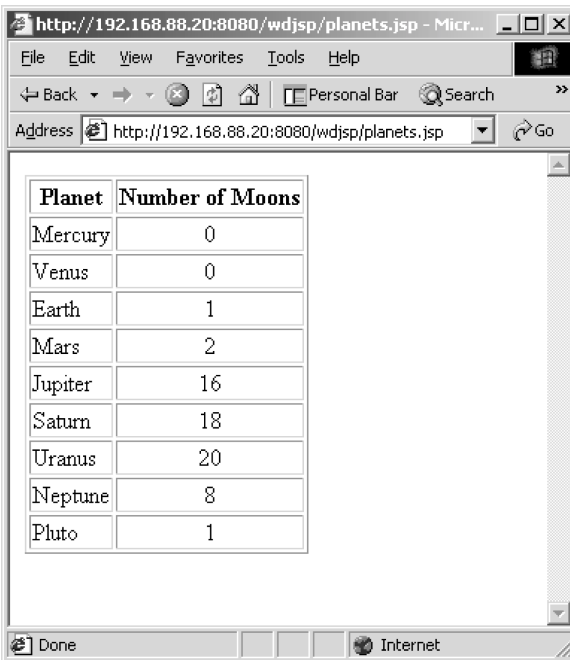
    public void first() {
        index = -1;
    }

    public boolean next() {
        index++;
        if (index >= numPlanets) {
            index--;
            return false;
        }
        else {
            return true;
        }
    }

    public String getName() {
        return names[index];
    }

    public int getMoons() {
        return moons[index];
    }
}
```

The while loop continues calling `next()`, incrementing the index, until it reaches the end of the list. Each time through, the index is pointing at a different planet's



Planet	Number of Moons
Mercury	0
Venus	0
Earth	1
Mars	2
Jupiter	16
Saturn	18
Uranus	20
Neptune	8
Pluto	1

Figure 8.2 Output of planet.jsp

data. We can then use our JSP bean tags to retrieve the corresponding properties. Although we didn't use it in this example, we provided a `first()` method to roll-back the index to just prior to first element. This lets us rewind if we need to display the list again. As this is a simple example, we've not implemented bounds checking on the properties.

8.1.6 Boolean properties

For boolean properties, that hold only true or false values, you can elect to use another bean convention for getter methods. This convention is to prefix the property name with the word `is` and return a boolean result. For example, consider these method signatures:

```
public boolean isProperty();  
public boolean isEnabled();  
public boolean isAuthorized();
```

The container will automatically look for this form of method if it cannot find a property access method matching the getter syntax discussed earlier. Setting the

value of a boolean property is no different then the setter methods for other properties.

```
public void setProperty(boolean b);
public void setEnabled(boolean b);
public void setAuthorized(boolean b);
```

8.1.7 JSP type conversion

A JSP component's properties are not limited to `String` values, but it is important to understand that all property values accessed through the `<jsp:getProperty>` tag will be converted into a `String`. A getter method need not return a `String` explicitly, however, as the JSP container will automatically convert the return value into a `String`. For the Java primitive types, conversion is handled by the methods shown in table 8.1

Table 8.1 Type conversions for `<jsp:getProperty>`

Property Type	Conversion to String
boolean	<code>java.lang.Boolean.toString(boolean)</code>
byte	<code>java.lang.Byte.toString(byte)</code>
char	<code>java.lang.Character.toString(char)</code>
double	<code>java.lang.Double.toString(double)</code>
int	<code>java.lang.Integer.toString(int)</code>
float	<code>java.lang.Float.toString(float)</code>
long	<code>java.lang.Long.toString(long)</code>
object	calls the Object's <code>toString()</code> method

Likewise, all property setter methods accessed with a `<jsp:setProperty>` tag will be automatically converted from a `String` to the appropriate native type by the JSP container. This is accomplished via methods of Java's wrapper classes as shown in table 8.2.

Table 8.2 Type conversions for `<jsp:setProperty>`

Property Type	Conversion from String
boolean OR Boolean	<code>java.lang.Boolean.valueOf(String)</code>
byte OR Byte	<code>java.lang.Byte.valueOf(String)</code>
char OR Character	<code>java.lang.Character.valueOf(String)</code>
double OR Double	<code>java.lang.Double.valueOf(String)</code>

Table 8.2 Type conversions for `<jsp:setProperty>` (continued)

Property Type	Conversion from String
int or Integer	<code>java.lang.Integer.valueOf(String)</code>
float or Float	<code>java.lang.Float.valueOf(String)</code>
long or Long	<code>java.lang.Long.valueOf(String)</code>
object	<code>as if new String(String)</code>

Properties are not restricted to primitive types. For objects, the JSP container will invoke the object's `toString()` method, which, unless you have overloaded it, will probably not be very representative of the data stored in the object. For properties holding objects rather than a `String` or native Java type you can set the property indirectly, for example allowing the user to set the hours and minutes separately through a pair of write-only properties and having a single read-only property called `time`.

Handling properties with null values

Property getter methods for Java's primitive types like `int` and `double` cannot return a `null` value, which is only valid for methods that return objects. Sometimes however, a property really is undefined. For example, if a property represents a user's age, and a call to the database reveals that we don't know their age, what do we return? While not that critical in many applications, it may be important to some. In this case, we can simply establish a convention for this property, which says if the age is a negative number then we don't have any idea what the age is—it is undefined. It is up to the JSP developer in this case to understand the convention and react to such a situation accordingly.

Unfortunately, it's not always that easy. How would we handle a temperature reading, where negative numbers are perfectly valid? We could still pick an unreasonable number, like `-999`, as an indicator that this particular value is unknown. However, such an approach is not only messy—requiring too much in-depth understanding by the JSP designer—it is also dangerous. Who knows what will be a reasonable value for this application (or its decedents) ten years from now? A better approach to this problem is to add a boolean property which can verify the legitimacy of the property in question. In that case, it doesn't matter what the property is actually set to. For example we would define both a `getTempReading()` and `isValidTempReading()` methods.

8.1.8 Configuring beans

Many times a bean will require run-time configuration by the page initializing it before it can properly perform its tasks. Since we can't pass information into the bean's constructor we have to use the bean's properties to hold configuration information. We do this by setting the appropriate property values immediately after the container instantiates the bean in the body of the `<jsp:useBean>` tag or anywhere in the page before the bean's properties are accessed. It can be useful to set a flag in your class to indicate whether or not an instance is in a useful state, toggling the flag when all of the necessary properties have been set.

Even though the bean tags do not allow you to pass any arguments into a bean's constructor, you can still define constructors that take arguments. You will not however, be able to call them through bean tags. You can only instantiate an object requiring arguments in its constructor through a JSP scriptlet. For example:

```
<% Thermostat t = new Thermostat(78); %>  
The thermostat was set at a temperature  
of <%= t.getTemp() %> degrees.
```

One technique we have found useful is to provide a single method that handles all configuration steps. This method can be called by your constructors that take arguments, for use outside of bean tags, as well as by your property access methods once all the necessary properties have been configured. In this example we'll provide two constructors for this `Thermostat` class, as well as an `init()` method which would handle any necessary internal configuration. The zero argument constructor is provided for bean compatibility, calling the constructor which takes an initial temperature argument with a default value. Our `init()` method is then called through this alternate constructor.

```
public class Thermostat {  
    private int temp;  
    private int maxTemp;  
    private int minTemp;  
    private int fuelType;  
  
    public Thermostat() {  
        // no argument constructor for Bean use  
        this(75);  
    }  
  
    public Thermostat(int temp) {  
        this.temp = temp;  
        init();  
    }  
  
    public void setTemp(int temp) {  
        this.temp = temp;  
    }  
}
```

```
// initialize settings with this temp
init();
}

public int getTemp() {
    return temp;
}

private void init() {
    maxTemp = this.temp + 10;
    minTemp = this.temp - 15;
    if (maxTemp > 150)
        fuelType = Fuels.DILITHIUM;
    else
        fuelType = Fuels.NATURALGAS;
}
}
```

8.2 Some examples

In this section we will present a number of more detailed examples of creating JavaBeans for use in JSP. These examples are more in-depth than the ones we've been looking at so far, and they will help give you the feel for developing more complex components. For additional examples, see the beans we develop in chapters 9 and 11.

8.2.1 Example: a TimerBean

In the previous chapter we used a `TimerBean` to track the amount of time a user has been active in the current browsing session. In the bean's constructor we simply need to record the current time, which we will use as our starting time, into an instance variable:

```
private long start;

public TimerBean() {
    start = System.currentTimeMillis();
}
```

The `elapsedMillis` property should return the number of milliseconds that has elapsed since the session began. The first time we place a `TimerBean` into the session with a `<jsp:useBean>` tag, the JSP container will create a new instance of the bean, starting our timer. To calculate the elapsed time we simply compute the difference between the current time and our starting time:

```
public long getElapsedMillis() {
    long now = System.currentTimeMillis();
    return now - start;
}
```

The other property access methods are simply conversions applied to the elapsed milliseconds. We have chosen to have our `minutes` and `seconds` properties return whole numbers rather than floating points to simplify the display of properties within the JSP page and eliminate the issues of formatting and precision. If the application using our bean needs a finer degree of resolution, it can access the `milliseconds` property and perform the conversions themselves. You are often better off reducing component complexity by limiting the properties (and corresponding methods) you provide with the component. We have found it helpful to focus on the core functionality we are trying to provide, rather than attempt to address every possible use of the component.

```
public long getElapsedSeconds() {
    return (long)this.getElapsedMillis() / 1000;
}

public long getElapsedMinutes() {
    return (long)this.getElapsedMillis() / 60000;
}
```

For convenience we will add a method to restart the timer by setting our `start` to the current time. We'll then make this method accessible through the JSP bean tags by defining the necessary access methods for a `startTime` property and interpreting an illegal argument to `setStartTime()` as a request to reset the timer.

```
public void reset() {
    start = System.currentTimeMillis();
}

public long getStartTime() {
    return start;
}

public void setStartTime(long time) {
    if (time <= 0)
        reset();
    else
        start = time;
}
```

The complete source for the bean is shown in listing 8.8.

Listing 8.8 TimerBean

```
package com.taglib.wdjsp.components;
public class TimerBean {
    private long start;
    public TimerBean() {
        start = System.currentTimeMillis();
    }
}
```

```
    }  
    public long getElapsedMillis() {  
        long now = System.currentTimeMillis();  
        return now - start;  
    }  
    public long getElapsedSeconds() {  
        return (long)this.getElapsedMillis() / 1000;  
    }  
    public long getElapsedMinutes() {  
        return (long)this.getElapsedMillis() / 60000;  
    }  
    public void reset() {  
        start = System.currentTimeMillis();  
    }  
    public long getStartTime() {  
        return start;  
    }  
    public void setStartTime(long time) {  
        if (time <= 0)  
            reset();  
        else  
            start = time;  
    }  
}
```

Here's an example of a JSP page that pulls a `TimerBean` from the user's session (or instantiates a new Bean, if necessary) and resets the clock, using the approach described in listing 8.8:

```
<jsp:useBean id="timer" class="TimerBean" scope="session"/>  
<jsp:setProperty name="timer" property="startTime" value="-1"/>  
<html><body>  
Your online timer has been restarted...  
</body></html>
```

8.2.2 A bean that calculates interest

As a more complex example let's create a JSP component that knows how to calculate the future value of money that is accumulating interest. Such a bean would be useful for an application allowing the user to compare investments. The formula for calculating the future value of money collecting compounding interest is:

$$FV = \text{principal}(1 + \text{rate}/\text{compounding periods})^{(\text{years} * \text{compounding periods})}$$

This bean will require:

- The sum of money to be invested (the principal)
- The interest rate
- The number of years for the investment
- How often interest is compounded

This gives us the list of properties that the user must be able to modify. Once all of these properties have been initialized, the bean should be able to calculate the future value of our principal amount. In addition, we will need to have a property to reflect the future value of the money after the calculation has been performed. Table 8.3 defines the bean's properties.

Table 8.3 Properties of a bean that calculates interest

Property Name	Mode	Type
principal	read/write	double
years	read/write	int
compounds	read/write	int
interestRate	read/write	double
futureValue	read-only	double

Since users will probably want to display the input values in addition to configuring them, they have been given both read and write access. The `futureValue` property is designated read-only because it will reflect the results of the calculation. Retrieving the value of the `futureValue` property uses the other properties to calculate our results. (If you wanted to get fancy, you could write a bean that, given any four of the properties, could calculate the remaining property value.) We'll store our initialization properties in instance variables:

```
public class CompoundInterestBean {
    private double interestRate;
    private int years;
    private double principal;
    private int compounds;
}
```

It is a good practice to make your instance variables `private` since we plan to define access methods for them. This assures that all interaction with the class is restricted to the access methods allowing us to modify the implementation without affecting code that makes use of our class. Following the bean conventions, we must define a

constructor that has no arguments. In our constructor we should set our initialization properties to some default values that will leave our bean property initialized. We cannot calculate the future value without our initialization properties being set to appropriate, legal values.

```
public CompoundInterestBean() {
    this.compounds = 12;
    this.interestRate = 8.0;
    this.years = 1;
    this.principal = 1000.0;
}
```

Since investments are generally compounded monthly (that is twelve times a year) it might be handy to provide a shortcut that allows the bean user to not specify the `compounds` property and instead use the default. It would also be nice if we could provide other clients of the bean with a more robust constructor that would allow them to do all their initialization through the constructor. This can be accomplished by creating a constructor that takes a full set of arguments and calling it from the zero-argument constructor with the default values we have selected for our bean's properties:

```
public CompoundInterestBean() {
    this(12, 8.0, 1, 1000.0);
}

public CompoundInterestBean(int compounds, double interestRate,
    int years, double principal) {
    this.compounds = compounds;
    this.interestRate = interestRate;
    this.years = years;
    this.principal = principal;
}
```

This is a good compromise in the design. The bean is now useful to both traditional Java developers as well as JSP authors. We must now define access methods for our initialization properties. For each one we will verify that they have been passed valid information. For example, money cannot be invested into the past, so the `year` property's value must be a positive number. Since the access methods are all similar, we'll just look at those for the `interestRate` property.

```
public void setInterestRate(double rate) {
    if (rate > 0)
        this.interestRate = rate;
    else
        this.interestRate = 0;
}
```

```
public double getInterestRate() {
    return this.interestRate;
}
```

When we catch illegal arguments, such as negative interest rates, we have to decide the appropriate way of handling it. We can pick a reasonable default value, as we did here for example, or take a stricter approach and throw an exception.

We chose to initialize our properties with a set of legitimate, but hard-coded values to keep our bean in a legal state. Of course, this approach might not be appropriate in every situation. Another technique for handling uninitialized data is setting up boolean flags for each property which has no legal value until it is initialized, and tripping them as each setter method is called. Another method could then be used to check the status of the flags to determine if the component had been initialized yet or not. For example, we could have defined our `futureValue` access method like this:

```
public double getFutureValue() {
    if (isInitialized())
        return principal * Math.pow(1 + interestRate/compounds,
            years * compounds);
    else
        throw new RuntimeException("Bean requires configuration!");
}

private boolean isInitialized() {
    return (compoundsSet && interestRateSet && yearsSet && principalSet);
}
```

In such a case, the bean is considered initialized if and only if the flags for each property are set to `true`. We would initialize each flag to `false` in our constructor and then define our setter methods as:

```
public void setYears(int years) {
    yearsSet = true;
    if (years >=1 )
        this.years = years;
    else
        this.years = 1;
}
```

The complete code is shown in listing 8.9:

Listing 8.9 CompoundInterestBean.java

```
package com.taglib.wdjsp.components;
public class CompoundInterestBean {
```

```
private double interestRate;
private int years;
private double principal;
private int compounds;

public CompoundInterestBean() {
    this(12);
}

public CompoundInterestBean(int compounds) {
    this.compounds = compounds;
    this.interestRate = -1;
    this.years = -1;
    this.principal = -1;
}

public double getFutureValue() {
    if ((compounds != -1) &&
        (interestRate != -1) &&
        (years != -1))
        return principal * Math.pow(1+interestRate/compounds, compounds*12);
    else
        throw new RuntimeException("Bean requires configuration!");
}

public void setInterestRate(double rate) {
    if (rate > 0)
        this.interestRate = rate;
    else
        this.interestRate = 0;
}

public double getInterestRate() {
    return this.interestRate;
}

public void setYears(int years) {
    if (years >=1)
        this.years = years;
    else
        this.years = 1;
}

public int getYears() {
    return this.years;
}

public void setPrincipal(double principal) {
    this.principal = principal;
}

public double getPrincipal() {
    return this.principal;
}
```

```
    }  
  
    public static void main(String[] args) {  
        CompoundInterestBean bean = new CompoundInterestBean();  
        bean.setInterestRate(0.06);  
        bean.setYears(30);  
        bean.setPrincipal(1200.00);  
        System.out.println("FutureValue = " + bean.getFutureValue());  
    }  
}
```

8.3 Bean interfaces

While not specifically required, there are a number of interfaces that you may choose to implement with your beans to extend their functionality. We'll cover them briefly in this section.

8.3.1 The BeanInfo interface

We learned about reflection earlier, but another way that a bean class can inform the bean container about its properties is by providing an implementation of the `BeanInfo` interface. The `BeanInfo` interface allows you to create a companion class for your bean that defines its properties and their corresponding levels of access. It can be used to adapt existing Java classes for bean use without changing their published interface. It can also be used to hide what would normally be accessible properties from your client, since sometimes Java's standard reflection mechanism can reveal more information than we would like.

To create a `BeanInfo` class use your bean's class name with the suffix `BeanInfo` and implement the `java.beans.BeanInfo` interface. This naming convention is how the bean container locates the appropriate `BeanInfo` class for your bean. This interface requires you to define methods that inform the container about your bean's properties. This explicit mapping eliminates the introspection step entirely.

There is also a `java.beans.SimpleBeanInfo` class that provides default, do-nothing implementations of all of the required `BeanInfo` methods. This often provides a good starting point when designing a `BeanInfo` class for a JSP bean, because many of the bean features designed for working with visual beans are irrelevant in the context of JSP, and are ignored by the JSP container.

One area where the `BeanInfo` approach is particularly useful is in visual, or WYSIWYG, JSP editors. JSP was designed to be machine-readable in order to support visual editors and development tools. By applying the `BeanInfo` interface to existing Java classes, developers can construct their own JSP components for use in such editors, even if the original component class does not follow the JavaBean con-

ventions. Using `BeanInfo` classes you can designate which methods of an arbitrary class correspond to bean properties, for use with the `<jsp:setProperty>` and `<jsp:getProperty>` tags.

8.3.2 The *Serializable* interface

One of the JavaBean requirements that JSP does not mandate is that beans should implement the `Serializable` interface. This will allow an instance of the bean to be *serialized*, turning it into a flat stream of binary data that can be stored to disk for later reuse. When a bean is serialized to disk (or anywhere else for that matter), its state is preserved such that its property values remained untouched. There are several reasons why you might want to “freeze-dry” a bean for later use.

Some servers support indefinite, long-term session persistence by writing any session data (including beans) to disk between server shutdowns. When the server comes back up, the serialized data is restored. This same reasoning applies to servers that support clustering in heavy traffic environments. Many of them use serialization to replicate session data among a group of web servers. If your beans do not implement the `Serializable` interface, the server will be unable to properly store or transfer your beans (or other classes) in these situations.

Using a similar tactic, you might choose to store serialized copies of your beans to disk, an LDAP server, or a database for later use. You could, for example, implement a user’s shopping cart as a bean, which you store in the database between visits.

If a bean requires particularly complicated configuration or setup it may be useful to fully configure the beans’ properties as required, then serialize the configured bean to disk. This snapshot of a bean can then be used anywhere you would normally be required to create and configure the bean by hand, including the `<jsp:useBean>` tag via the `beanName` attribute.

The `beanName` attribute of the `<jsp:useBean>` tag is used to instantiate serialized beans rather than creating new instances from a class file. If the bean doesn’t exist in the scope, then the `beanName` attribute is passed on to `java.beans.Bean.instantiate()`, which will instantiate the bean for the class loader. It first assumes that the name corresponds to a serialized bean file (identified by the `.ser` extension) in which case it will bring it to life, but if it can’t find or invoke the serialized bean it will fall back to instantiating a new bean from its class.

8.3.3 The *HttpSessionBindingListener* interface

Implementing the Java Servlet API’s `HttpSessionBindingListener` interface in your JavaBean’s class will enable its instances to receive notification of session events. The interface is quite simple, defining only two methods.

```
public void valueBound(HttpSessionBindingEvent event)
public void valueUnbound(HttpSessionBindingEvent event)
```

The `valueBound()` method is called when the bean is first bound (stored into) the user's session. In the case of JSP, this will typically happen right after a bean is instantiated by a `<jsp:useBean>` tag that specifies a `session` scope, thus assigning the bean to the user's session.

The `valueUnbound()` method is called, as you would expect, when the object is being removed from the session. There are several situations that could cause your bean to be removed from the session. When the JSP container plans to expire a user's session due to inactivity, it is required to first remove each item from the session, triggering the `valueUnbound` notification. The JSP container will automatically recognize that the bean is implementing the `HttpSessionBindingListener` interface, hence there is no need to register the bean with the container as a listener. Alternatively, this event would be triggered if a servlet, scriptlet, or other Java code specifically removed the bean from the session for some reason.

Each of these events is associated with an `HttpSessionBindingEvent` object, which can be used to gain access to the session object. Implementing this interface will allow you to react to session events by, for example, closing connections that are no longer needed, logging transactions, or performing other maintenance activities. If you are implementing your own session persistence, such as saving a shopping cart, this would be where you would move your data off to disk or database.

8.3.4 Other features of the Bean API

In addition to the access methods and constructor conventions that we have examined here, the JavaBeans Specification defines several other features. When writing beans for use with JSP we do not generally need to concern ourselves with these remaining elements of the specification because they are more oriented toward visual beans, such as GUI components. While most of this extra functionality is not reflected into the bean tags, it can be useful working with beans through JSP scriptlets or as part of a larger system. For clarity and for the sake of completeness we will quickly point out these other features. For full details on these aspects of JavaBeans, see the JavaBeans Specification or Manning's *The Awesome Power of Java Beans*.

JavaBean event model

The JavaBeans API supports Java 1.1 style event handling, a feature intended primarily for visual components. Events allow visual beans to communicate with one another in a standard way, without each bean having to be too tightly coupled to

other beans. However, JSP containers do not support the JavaBeans event model directly. Any bean-to-bean communication is the responsibility of the bean designer.

Bound properties

A bean can be designed to generate events any time changes are made to its properties. This allows users of the bean to be notified of the changes and react accordingly. If, for example, a bean contained information about the status of a radio button on a user interface which was modified by one of the bean's users, any other users of the bean would be notified and could update their displays accordingly.

Constrained properties

Constrained properties are properties whose values must fall within specific limits. For example a property representing a percentage value must be greater than or equal to zero, and less than or equal to one hundred. The only difference between the design patterns for setting a constrained versus an unconstrained property is that it must declare that it throws the `java.beans.PropertyVetoException`. Objects that want to support constrained properties must also implement methods that allow other objects to register with the bean so that they can play a part in the change approval process. Constrained property functionality is not directly implemented through the bean tags, although beans can still take advantage of this functionality internally. If a bean throws an exception in response to an illegal property value, the normal JSP error handling will take place.

8.4 Mixing scriptlets and bean tags

Since JSP bean tags, scriptlets, and expressions eventually are translated into the same single Java servlet class on the server, you can combine any of the elements. This allows you to take advantage of component-centric design while not being bound by the limits of the built-in tag commands. Using the `<jsp:useBean>` tag to create objects puts them into the scope of the page, making them available to both scriptlets and `<jsp:getProperty>` and `<jsp:setProperty>` tags.

8.4.1 Accessing beans through scriptlets

Since the `<jsp:useBean>` tag creates an object reference behind the scenes, you are free to access that object through scriptlets and expressions, using the bean's name as the object identifier. For example, it is perfectly valid to do either of these snippets, both of which produce the same results:

```
<jsp:useBean id="stocks" class="StockMarketBean" scope="page"/>
The Dow is at <jsp:getProperty name="stocks" property="dow"/> points
```

OR

```
<jsp:useBean id="stocks" class="StockMarketBean" scope="page"/>
The Dow is at <%= stocks.getDow() %> points
```

Calling bean properties through an expression rather than the somewhat lengthy `<jsp:getProperty>` tag can be a handy shortcut if you aren't afraid of a little Java code in your page. A word of caution however! You can't always assume that a bean's property returns a String or maps directly to the method you expect. It may return a different type of data than you expect (which is all right if you are calling the method in an expression), or a `BeanInfo` class may be redirecting you to a completely different method—one for which you may not even know the name.

8.4.2 Accessing scriptlet created objects

The reverse of this operation is not true. Objects created through scriptlets are not guaranteed to be accessible through the bean tags, because there is no guarantee that these objects will become part of the page context. Consider the following JSP code for example, which is not valid in most JSP containers.

```
<html><body>
Auto-Shop 2000<br>
<% Car car = (Car)request.getAttribute("car"); %>
<% car.updateRecords(); %>
This car has <jsp:getProperty name="car" property="milage"/> miles on it...
</body></html>
```

In this example we have attempted to pull an object reference, `car`, out of the request and use it in the page. However, the `<jsp:getProperty>` tag will not have a reference to the object because it was not scoped into the page through a `<jsp:useBean>` tag. The corrected code is:

```
<html><body>
Auto-Shop 2000<br>
<jsp:useBean id="car" class="Car" scope="request"/>
<% car.updateRecords(); %>
This car has <jsp:getProperty name="car" property="milage"/> miles on it...
</body></html>
```

Notice that we can access the object through both scriptlets and JSP tags, allowing us to call the `updateRecords()` method directly. We can even change the object referenced by the named identifier specified by `<jsp:useBean>`—it is the identifier that's important, not the actual object reference.

Alternatively, you can scope the bean into the `pageContext` directly using the code:

```
pageContext.setAttribute("car", car);
```

Handling indexed properties

This technique is particularly useful in handling indexed properties, which JSP doesn't provide any easier way to deal with (other than custom tags, as we'll learn in chapters 18 and 19). We apply the same principles as before, creating objects with the `<jsp:useBean>` tag and referencing them through scriptlets and expressions. For example, to loop through an indexed property we write code similar to that which follows. The exact syntax will depend on your bean's properties and associated methods. In this example, `MusicCollectionBean` contains an array of `Album` objects, nested in its `albums` property. Each `Album` object in turn has a number of bean properties. Note however, that we must declare the `Album` object reference through a bean tag as a placeholder, or it will not be available to our page context and therefore inaccessible through the bean tags.

```
<jsp:useBean id="music" class="MusicCollectionBean"/>
<jsp:useBean id="album" class="Album"/>
<%
Album[] albums = music.getAlbums();
for (int j=0; j < albums.length; j++) {
    album = albums[j];
%>
Title: <jsp:getProperty name="album" property="title"/><BR>
Artist: <jsp:getProperty name="album" property="artist"/><BR>
Year: <jsp:getProperty name="album" property="year"/><BR>
<% } %>
```

This code will loop through each of the albums in the array returned by the `getAlbums()` method of `MusicCollectionBean`, assigning each to the variable `album` in turn. We can then treat `album` as a bean, accessing it through the `<jsp:getProperty>` tags. You can use this technique to create tables, lists, and other sequences of indexed properties.

Other bean methods

Since beans are just objects, they may also have methods that are accessible through JSP scripting elements. While it is desirable to create beans that can be used entirely through the tags, sometimes it is useful to create beans with two levels of complexity. These extra methods are not bean-related, but allow you to treat the bean as any other Java object for more benefits or advanced functionality.

Not all of your methods need to follow the bean conventions, although only those methods that can be found by introspection will be made available through the bean container. It is sometimes useful to provide basic functionality accessible through the bean container, such as JSP tags, and more advanced functionality only accessible through scriptlets or direct programmer intervention.

Removing a bean when done with it

At the end of a bean's life span, which is determined by its scope, all references to the bean will be removed and it will become eligible for garbage collection. Beans in the page or request scopes are automatically reclaimed at the end of the HTTP request, but session and application beans can live on. The life of a session bean is, as discussed, dependent on the JSP container while the application scope is tied to the life of the server. There are several situations where you might want to prematurely end the life of a bean. The first involves removing it from memory for performance reasons. When you have no more use for the bean, especially one in session or application scope, it's a good idea to get rid of it. Eliminating unused bean objects will improve the performance of your server-side applications by freeing as many of the JVM's resources as soon as possible.

Another reason you want to remove a bean is to eliminate it from the user's session for security reasons. A good example of this would be removing a user's login information from the session when the user has specifically advised that they are logging off. A typical approach to user authentication with JSP is to place the user's login credentials into the session following a successful login. The presence of these credentials in the session satisfies the login requirements for future visits to protected pages until the session expires. For security reasons however it is desirable to offer the visitor the ability to eliminate their login information from the session when they have completed their visit. We can accomplish this by simply removing their credentials from the session, returning them to their unauthenticated state. The methods available to you are summarized in table 8.4.

Table 8.4 Discarding a used bean from various scopes

Scope	Scriptlet	Servlet
session	<code>session.removeAttribute(name)</code>	<code>HttpSession.removeAttribute(name)</code>
request/page	<code>pageContext.removeAttribute(name)</code>	<code>ServletRequest.removeAttribute(name)</code>
application	<code>application.removeAttribute(name)</code>	<code>ServletContext.removeAttribute(name)</code>

The request bean

As discussed in previous chapters, JSP defines a number of implicit objects that reflect information about the environment. The request object encapsulates information about the request and has several properties that are accessible through the bean tags. Like other beans, we can access the properties of the request objects through `<jsp:getProperty>`. The `id` value assigned to the implicit request object is, as you probably guessed, `request`. For example, we can display the remote user name as follows:

```
<jsp:getProperty name="request" property="remoteUser"/>
```

Table 8.5 summarizes some of the more useful methods of the request object, which can be exposed as properties to the bean tags.

Table 8.5 Properties of the request bean

Name	Access	Use
<code>authType</code>	read	Gets the authentication scheme of this request or null if unknown. Same as the CGI variable <code>AUTH_TYPE</code>
<code>method</code>	read	Gets the HTTP method (for example, <code>GET</code> , <code>POST</code> , <code>PUT</code>) with which this request was made. Same as the CGI variable <code>REQUEST_METHOD</code>
<code>pathInfo</code>	read	Gets any optional extra path information following the servlet path of this request's URI, but immediately preceding its query string. Same as the CGI variable <code>PATH_INFO</code>
<code>pathTranslated</code>	read	Gets any optional extra path information following the servlet path of this request's URI, but immediately preceding its query string, and translates it to a real path. Same as the CGI variable <code>PATH_TRANSLATED</code>
<code>queryString</code>	read	Gets any query string that is part of the HTTP request URI Same as the CGI variable <code>QUERY_STRING</code>
<code>remoteUser</code>	read	Gets the name of the user making this request. The user name is set with HTTP authentication. Whether the user name will continue to be sent with each subsequent communication is browser-dependent. Same as the CGI variable <code>REMOTE_USER</code>
<code>requestURI</code>	read	Gets the URI corresponding to the original request
<code>characterEncoding</code>	read	Gets the character set encoding for the input of this request
<code>contentType</code>	read	Gets the Internet media type of the request entity data, or null if not known. Same as the CGI variable <code>CONTENT_TYPE</code>
<code>protocol</code>	read	Gets the protocol and version of the request as a string of the form <code><protocol>/<major version>.<minor version></code> . Same as the CGI variable <code>SERVER_PROTOCOL</code>

Table 8.5 Properties of the request bean (continued)

Name	Access	Use
<code>remoteAddr</code>	read	Gets the IP address of the agent that sent the request. Same as the CGI variable <code>REMOTE_ADDR</code>
<code>serverName</code>	read	Gets the host name of the server that received the request. Same as the CGI variable <code>SERVER_NAME</code>
<code>serverPort</code>	read	Gets the port number on which this request was received. Same as the CGI variable <code>SERVER_PORT</code>
<code>scheme</code>	read	Gets the scheme of the URL used in this request, for example "http," "https," or "ftp"
<code>remoteHost</code>	read	Gets the fully qualified host name of the agent that sent the request. Same as the CGI variable <code>REMOTE_HOST</code>