

You are here: [Home](#) > [Developers](#) > [View Source Articles](#) > Java View Source Article

Developer Program  
[Membership](#)  
[One-to-One Support](#)  
[Newsgroups](#)

Developer Publications  
[View Source](#)  
[Developer News](#)

Documentation  
[Technical Manuals](#)  
[White Papers](#)  
[TechNotes](#)  
[Sample Code](#)  
[FAQs](#)  
[Books](#)

Technologies  
[CORBA](#)  
[Directory & LDAP](#)  
[Dynamic HTML](#)  
[Java](#)  
[JavaScript](#)  
[Linux](#)  
[Security](#)  
[SSJS](#)  
[XML](#)

Developer Downloads  
[Tools & SDKs](#)  
[Patches](#)

iPlanet Products  
[Technical Resources](#)

## Building Your Own JSP Components

By [Duane K. Fields](#) and [Mark Kolb](#)

---

[Send comments and questions](#) about this article to [View Source](#).

---

*[Editor's note: This article is adapted from a chapter of the book Web Development with JavaServer Pages, to be published in May 2000 by Manning Publications Co. It appears here with the permission of the publisher. More information about this book is available at [Manning's web site](#).]*

The JavaServer Pages (JSP) component model is centered on Java software components called *Beans*, which must adhere to specifications outlined in the JavaBeans API. The JavaBeans API, created by Sun Microsystems with industry cooperation, dictates the rules that software developers must follow to create stand-alone, reusable Java software components. By using JSP's collection of Bean tags, content developers can use the power of Java to add dynamic elements to their pages without writing a single line of code.

This article is written for developers who want to create their own Beans for use as JSP components, and for interested web designers who want to understand how these components are implemented behind the scenes. It is not necessary to understand the details of Beans development to work with JSP. As component architectures go, the interface between JSP and Beans is quite simple, as we will see.

### WHAT MAKES A BEAN A BEAN?

So, what makes a Bean so special? You might be surprised to learn that 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 the 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. In the next section, we will discuss these conventions and show you how to create your own Beans.

### 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 container 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.

### Bean Are Just Objects

As with 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. Since they follow the Bean conventions, we can work with Beans without having to write 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

expressions, or custom tags to access them, since a Bean container can manipulate a Bean only through its properties.

## Class Naming Conventions

You might notice that in most 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 and lets other developers immediately understand the intended role of the class, it is not a requirement in order for a Bean to be used inside a JSP page. 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.

## BEAN CONVENTIONS

The Bean 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 Bean. The conventions dictate rules for defining a Bean's constructor and the methods that will define its properties.

### 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, 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 cannot 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; it has a zero-argument constructor that records the time of its instantiation:

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

    public CurrentTimeBean() {
        java.util.Date now = new java.util.Date();
        this.hours = now.getHours();
        this.minutes = now.getMinutes();
    }
}
```

In this example, we have 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, in order for it 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.

### The Magic of Introspection

You may be wondering how the JSP container can interact with Bean objects 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 properties on request. The introspection process happens at run time, controlled by the Bean container.

One way that introspection can occur is through a mechanism known as *reflection*, which allows the Bean container to examine any class at run time to determine its set of properties. The Bean container determines what properties a Bean supports by analyzing its public methods for the presence of property access 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 alone of the specially named access methods that determines a Bean class's properties.

### Specifying a Bean's Properties

## Specifying a Bean's Properties

As we have mentioned, a Bean's properties are defined simply by creating appropriate access methods for them. Access methods are used to either retrieve a property's value or make changes to it. A method used to retrieve a property's value is called a *getter*, while a method that modifies its value is called a *setter*. Together these 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. A getter method 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 as follows:

```
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 writeable, 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();
```

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 access 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 intentionally 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 affect code that is dependent on the older version of the component.

Let us 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. For Java to recognize the existence of these properties, they will have to follow the getter method signatures defined by the JavaBeans design patterns. These methods should therefore look like this:

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

In our constructor, we store the current time's hours and minutes in instance variables. We can have our properties reference these variables and return their values where appropriate:

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

    public CurrentTimeBean() {
        java.util.Date now = new java.util.Date();
        this.hours = now.getHours();
        this.minutes = now.getMinutes();
    }

    public int getHours() {
        return hours;
    }
}
```

```

    }

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

```

That's all there is to it. The two methods simply return the appropriate values as stored in the instance variables. Since these methods meet the Bean conventions 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. 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 exposed 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 instead of simply returning a copy of an instance variable:

```

public class DiceBean {
    private java.util.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 actual 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*, 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 you want clients accessing 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

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 - the JSP container assumes that you really mean it, so its corresponding access methods would be `getURL()` and `setURL()`.

## 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. All values must be of the same type; however, a single indexed property cannot contain both string and integer elements, for example.

To define an indexed valued property, you have two options: create an access method that returns the entire set of properties as a single array, or access elements of the set by using an index value.

In creating an access method that returns the entire set of properties as a single array, a JSP page author or iterative custom tag can determine the size of the set and iterate through it. For example:

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

Accessing elements of the set by using an index value allows 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 the Bean conventions, we find it useful to implement both method styles for a multivalued property. It is not much more work and it gives you a good deal more flexibility in using the Bean.

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

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

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

```
public int getPropertySize()
```

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

## 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.

```
package com.manning.jsp;
import java.util.*;

public class StatBean {
    private double[] numbers;

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

    public double getAverage() {
        double sum = this.getSum();
        if (sum == 0)
            return 0;
        else
            return sum/numbers.length;
    }
}
```

```

    }

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

    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 will 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 will have to use a scriptlet to display back the numbers themselves, but we can use a `<jsp:getProperty>` tag to display the average.

```

<jsp:useBean id="stat" class="com.manning.jsp.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 the book, can greatly aid in working with indexed properties, eliminating the need for inline code by encapsulating common functionality into simple tag elements.

### 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 Bean tag. Since Bean tags deal exclusively with single values, we will have to perform the conversion ourselves. We will create a 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.

```
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));
}

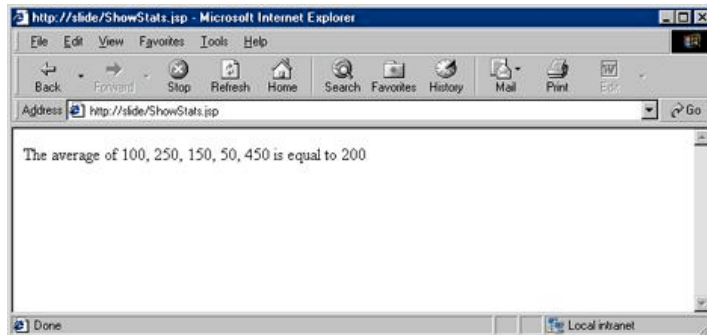
public String getNumbersList() {
    String list = new String();
    for (int i=0; i < numbers.length; i++) {
        if (i != numbers.length)
            list += numbers[i] + ",";
        else
            list += "" + numbers[i];
    }
    return list;
}
```

Now we can access this Bean through JSP tags alone:

```
<jsp:useBean id="stat" class="com.manning.jsp.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>
```



## Boolean Properties

For boolean properties, which 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 `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. You can set the value of a boolean property with the same style setter methods you would use for other properties, as shown below.

```

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

```

### 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 to strings. A getter method need not return a `String` explicitly, however, as the JSP container will automatically convert the return value to a `String` as needed. For the Java primitive types, conversion is handled by the methods shown in Table 1.

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>

Table 1.

Type conversions for `<jsp:getProperty>`

Similarly, 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 2.

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>
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>

Table 2. Type conversions for `<jsp:setProperty>`

Properties are not restricted to primitive types, either. 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 representing objects too complex to represent with a `String` or native Java type, you have several strategies. You can create getters and setters that accept a `String` or native type and then perform the necessary conversions for creating an appropriately typed object from the `String` or native data. You can also overload your getter and setter methods to accept the appropriate object type, although custom tags or JSP scripting elements will be required to access the

overloaded methods, since the `<jsp:setProperty>` and `<jsp:getProperty>` tags work exclusively with `String` values. You can also 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`.

## Configuring Beans

Many times a Bean will require runtime configuration by the page that is initializing it before it can properly perform its tasks. Since we cannot 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 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. The only way to instantiate an object requiring arguments in its constructor within a JSP page is through a scriptlet. You will be able to access the Bean's properties through the normal

`<jsp:setProperty>` and `<jsp:getProperty>` tags using the ID assigned to the object by the scriptlet. For example:

```
<%
    Thermostat t = new Thermostat(78);
%>
The thermostat was set at a temperature
of <jsp:getProperty name="t" property="temp"\>
degrees.
```

One technique we have found useful is to provide a single method that handles all of your 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 will 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 that 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.DILITHEUM;
    else
        fuelType = Fuels.NATURALGAS;
    }

```

### Example: A TimerBean

In this example, we will create a `TimerBean` to track the amount of time a user has been active in a current browsing session. In the Bean's constructor, we simply need to record the current time (which we will use as our starting time) in an instance variable:

```

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

```

The `elapsedMillis` property should return the number of milliseconds that have 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 an 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 (shown below) are simply conversions applied to the elapsed milliseconds. We have chosen to have our `minutes` and `seconds` properties return whole numbers rather than floating-point numbers 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 itself. You are often better off reducing component complexity by limiting the properties (and corresponding methods) that 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 starting time to the current time. We will 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;
}

```

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 above:

```

<jsp:useBean id="timer" class="TimerBean" scope="session">
<jsp:setProperty name="timer" property="startTime" value="-1"/>
</jsp:useBean>

```

```
<HTML>
<BODY>
Your online timer has been restarted...
</BODY>
</HTML>
```

### 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

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

This Bean will require the following information:

- the sum of money to be invested (the principal)
- the interest rate
- the number of years the money is invested
- the frequency with which 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. We will start by defining the Bean's properties (see Table 3).

---

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

**Table 3. Interest bean properties**

---

Since the user will probably want to display the input values in addition to configuring them, we have given the user both read and write access. The

`futureValue` property is designated read-only because it will reflect the results of the calculation. To determine the value of the `futureValue` property, the JSP container plugs the values of the other properties into our interest formula. (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 will store our initialization properties in instance variables:

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

It is good practice to make our instance variables `private`, since we plan to define access methods for them. This ensures 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 does not have any 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 having our initialization properties set to appropriate, legal values.

```
public CompoundInterestBean() {
    this.compounds = 12;
```

```

        this.interestRate = 8.0;
        this.years = 1;
        this.principal = 1000.0;
    }
}

```

Since interest on investment is generally compounded monthly (12 times a year), it might be handy to provide a shortcut that allows the Bean user not to specify the `compounds` property and instead use the default. It would also be nice if we could provide a more robust constructor that would allow a Bean's clients to do all of 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 and JSP page authors. We must now define access methods for our initialization properties. For each one, we will verify that it has been passed valid information. For example, money cannot be invested in the past, so the `year` property's value must be a positive number. Since the access methods are similar, we will 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, like negative interest rates, we have to decide the appropriate way of handling this situation. 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 legal values to keep our Bean in a legal state. Of course, this might not be appropriate in every situation. For Beans particularly sensitive to their configuration state, you might need to design a scheme for marking a property as uninitialized, such as setting it to

`null`, and react accordingly. Another technique that works well is setting up `boolean` flags for each property and tripping them as each setter method is called. 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 flag associated with each property is set to true. We would initialize each flag to false in our constructor and then define our setter methods like this:

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

Here is the complete code:

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

    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;
    }

    public double getFutureValue() {
        return principal * Math.pow(1 + interestRate/compounds,
            years * compounds);
    }

    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 void setCompounds(int compounds) {
```

```

        if (compounds >=1)
            this.compounds = compounds;
        else
            this.compounds = 12;
    }

    public int getCompounds() {
        return this.compounds;
    }
}

```

## BEAN INTERFACES

While not specifically required, there are a number of interfaces that you may choose to implement with your Beans. These interfaces can be used to extend the functionality of your Beans for various situations. We will cover them briefly in this section.

### 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 you would like.

To create a

`BeanInfo` class, you simply use your Bean's class name with the `BeanInfo` prefix 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.

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 you are 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 Java classes, developers can construct their own JSP components for use in such editors, even if the original component class does not follow the Bean conventions. 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.

### The Serializable Interface

One of the JavaBeans requirements that JSP does not actually mandate is that Beans 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 for later use. When a Bean is serialized to disk (or anywhere else for that matter), its state is preserved such that its property values remain 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 Bean's properties as required and 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 create brand new instances from a class file. If the Bean does not already exist in the scope, the beanName attribute is passed to

```
java.beans.Bean.instantiate()
```

, which will instantiate the Bean for the classloader. It first assumes that the name corresponds to a serialized Bean file (identified by the .ser extension), in which case it will bring the Bean to life; however, if it cannot find or invoke the serialized Bean, it will fall back to instantiating a new Bean from its class.

### The HttpSessionBindingListener Interface

Implementing the Java Servlet API's HttpSessionBindingListener interface in your Bean'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 in) 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. Several situations could cause your Bean to be removed from the session. When the JSP container plans to terminate a user's session due to inactivity, it is required to first remove each item from the session, triggering the valueUnbound() notification. 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.

### CONCLUSION

JSP allows developers to add dynamic elements to web pages by interleaving their HTML pages with Java code. While this powerful new technology allows Java programmers to create web-based applications and expressive sites, it lacks elegant separation between presentation and implementation. JSP Bean tags provide an alternative, component-centric approach to dynamic page and web application design. JSP Bean tags allow content developers to interact with Java components not through Java code, but through HTML-like tags.

By building its component model around the relatively simple JavaBeans API, JSP enables Java developers to quickly package a web application's core functionality into discrete, reusable software components. These components

can then be incorporated into JSP pages as easily as HTML elements. This approach allows for a cleaner division of labor between application and content developers.

---

## FURTHER RESOURCES

- [JavaServer Pages information from Sun Microsystems](#)
- *View Source* articles:
  - [JavaBeans: An Architecture for Reusable Software Components](#)
  - [Advanced JavaBeans](#)
  - [Using Components to Develop Applications: An Introduction to Visual JavaScript](#)
  - [Introduction to JavaServer Pages: Server-Side Scripting the Java Way](#)
  - [Looking Into Enterprise JavaBeans](#)

---

*View Source* wants your feedback!  
[Write to us](#) and let us know  
what you think of this article.

---

[Duane Fields](#) is a Senior Engineer for the E-Business Enablement group of IBM's Tivoli Systems, where he creates web-based applications with Java and JSP. He lives in Austin, Texas.

[Mark Kolb](#) has a Ph.D. in aerospace engineering and was the recipient of a NASA Space Act Award. He now leads the development of server-side web applications using servlets and JSP in Tivoli Systems' Internet Business Unit. He lives in Round Rock, Texas.

(11.99)

Any sample code included above is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#)