Using JavaBeans with tags

In this chapter

- Reflection and the Java Reflection API
- JavaBeans primer
- Printing bean attributes
- Exploring bean attributes and scripting variables

One of the most common tasks in any Java program is to get and set properties of JavaBeans. Web development in Java is no different in that respect. In an object-oriented Java web application, your servlets and JSPs likely contain a good deal of code that interacts with JavaBeans. For servlets, working with JavaBeans is the same as is in any other Java program: beans are instantiated and the proper get or set methods are called. JSPs interact with JavaBeans differently, by using three standard tags that are defined in the JSP specification (`<jsp:usebean>`, `<jsp:setProperty>`, and `<jsp:getProperty>`). These tags, discussed in chapter 2, instantiate beans, introduce them to the page as scripting variables, and set properties in the beans in addition to printing properties back to the user. Unfortunately, the functionality exposed by the JavaBeans-related tags is rather limited. Two of the more troublesome limitations are:

- The tags cannot operate on objects unless they were introduced to the page via a JavaBeans-related JSP tag.
- The tags cannot manipulate indexed properties (properties that have multiple values and are referred to by an index like `getSomeProperty(i)`).

These shortcomings render JavaBeans standard tags useless for solving many practical problems such as iterating over an indexed property of a bean, or printing an attribute value of a bean that we would rather not introduce as a JSP scripting variable. Not surprisingly, developers increasingly find themselves in need of a more versatile set of JavaBeans tags. In this chapter, we address that need.

To this end we'll create a custom tag library that supports interaction with JavaBeans, free from the limitations of today's standard tags. We start with a brief discussion of the Java reflection API, which we'll soon see is crucial to building our JavaBean tags. We then move on to discuss the JavaBean component standard and how reflection assists us in getting and setting properties of Java Beans. We finish by creating our custom tag library which will consist of two tags: one that allows us to expose any JavaBean as a JSP scripting variable, and one that supports printing of the value of any JavaBean property to the user.

8.1 Java reflection

If tags are to be useful, they must be created such that they are versatile enough to get and set the properties of any JavaBean. This requires that they be capable of calling a bean's get or set methods, simply by knowing the class name of the JavaBean and the name of the property to be manipulated. If you've heard of the Java reflection API, you know that it is a set of Java methods and classes that allows us to do just that. Using reflection, we can learn about the attributes of any Java class (including its methods) and even call any method of the class we choose. Before

discussing the integration of reflection, JavaBeans, and custom tags, a quick look at the Java reflection API is in order. This will not be an in-depth description of reflection or method invocation by using reflection, but will provide a helpful explanation of the technology, its advantages, and the API that can put it to good use.

8.1.1 What is reflection?

Javasoft describes the purpose of the reflection API as "to enable Java code to discover information about the fields, methods, and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions." Quite a mouthful, so what does all that mean? In brief, the reflection API lets us learn about a particular Java class and then use that knowledge to properly call methods, get and set fields, or call the constructor of any instance of that class. The reflection API is particularly useful for debuggers and development tools that need to browse classes and display certain information from the classes to the user. The API is also useful when parts of your Java program have to interact with any type of Java object and need to learn about it at runtime. The tags we create in this chapter will have this requirement, since they will be designed to work with any and all JavaBeans. This definition can be clarified with a few examples of programs that use Java reflection.

Reflection and development tools

Consider the case of an environment wherein a developer manipulates program components through the GUI of a Java IDE (like JBuilder, Visual Café, etc.). We recognize that the development environment knows nothing about the components in advance, yet it must be able to present the developer with the possible methods that can be used in each component. Spying on the component to discern the interfaces, methods, and properties they expose can be accomplished using the reflection API, and is better known as *Introspection*.

Reflection and scripting engines

Another case is one wherein a user employs a JavaBean-capable scripting engine to create an application. Since a script is not usually precompiled with all its components, it does not know anything in advance about the different JavaBeans components with which it will interact at runtime (not even their type); yet, during runtime it should be able to perform the following:

- Introspect the components to find the method that it should execute.
- Dynamically execute the method on the scripted object.

Both of these functions are available through the reflection API.

By now you have seen how reflection is used to learn about a class at runtime. The tags we build in this chapter, like the standard JavaBean tags, will take as tag attributes the reference to an instance of a JavaBean and, for our printing tag, also the name of the property to print. Since the JSP author may specify any JavaBean instance at all, our tags will need to be able to take that instance at runtime and use reflection to find and call the proper methods. It should be clear that the only way to accomplish this is through the reflection API. Let us look at the syntax of that API in greater detail.

NOTE Using the reflection API in order to introspect methods and later invoke them follows strict security rules that disallow overriding the Java security model. For example, one may not introspect private and package protected classes.

8.1.2 The reflection API

The reflection APIs are contained in the Java package `java.lang.reflect`. Some of the more important classes in this package are shown in table 8.1.

Table 8.1 Classes that are important for reflection

Class	Description
`Class`	Represents a Java class. Although not part of the `java.lang.reflect` package, this class is very important for reflection.
`Method`	Represents a method of class (also allows that method to be called).
`Array`	Supports creation and accessing of Java arrays (without having to know the exact type of the array in advance).
`Constructor`	Represents a specific class constructor (and allows that constructor to be called).

The Class class

The means to obtain all the constructors, methods, and fields for a particular class (or interface) in Java is through an instance of `java.lang.Class`. We obtain an instance of `Class` that corresponds to a particular Java class in a couple of ways:

- Calling `Class.forName("com.manning.SomeClass")` where "com.manning.SomeClass" is the fully qualified name of the class we want to study.
- Referring to the `class` field of a particular Java class. Code for this looks like:

```
Class c= com.manning.SomeClass.class
```

Once we have a `Class` object for our class, we may use methods such as `getConstructors()`, `getMethods()`, and `getFields()` to retrieve information about this class.

The Method class

For the purpose of this book, the class most important to us is `Method`, mainly because, by obtaining a `Method` object from a certain class, we are able to call that method repeatedly and on any instance of this class. This is the approach we will use to call the `get` and `set` property methods within the custom JavaBean tags we are building. The methods that are part of the `Method` class are presented in table 8.2, along with a brief description of each.

Table 8.2 Methods in the Method class

Method name	Description
`getDeclaringClass()`	Returns a `Class` object that represents the class or interface to which that method belongs. Think of it as the `Class` object that, if you call one of its `getMethod()` methods, will return this as the `Method` object.
`getModifiers()`	Returns an integer representation of Java language modifiers. Later on, the `Modifier` utility class can decode this integer.
`getName()`	Returns the name of the method.
`getParameterTypes()`	Returns an array of `Class` objects that represent the parameter types, in declaration order, of the method.
`getReturnType()`	Returns a `Class` object that represents the return type of the method.
`invoke(Object obj, Object[] args)`	Invokes the underlying method on the specified object with the specified parameters.
`getExceptionTypes()`	Returns an array of `Class` objects representing the types of exceptions declared to be thrown by the method.
`equals(Object obj)`	Compares this `Method` against the specified object.
`hashCode()`	Returns a `hashcode` for this Method.
`toString()`	Returns a string describing this Method.

Some of the methods mentioned in table 8.2, such as `equals()`, `hashCode()`, and `toString()` do not require any real explanation, as anyone familiar with Java programming knows how and when to use these methods. The remainder of the methods, however, require some ground rules:

- All parameters and return codes in `invoke()` are passed wrapped within an object type. If some of the parameters or the return value are of primitive types (such as `char`) they need to be wrapped in an object (such as `java.lang.Character`). Table 8.3 presents the primitive types and their corresponding wrapping objects.
- The object on which `invoke()` will execute the method is passed as the `obj` parameter to `invoke()`.
- Since the number of parameters differs from one method to another, `invoke()` accepts an array of objects in which we place the parameters according to the order of declaration.
- The value returned from the invoked method is returned from `invoke()` (wrapped in an object if necessary).
- Exceptions thrown by the invoked methods are thrown by `invoke()` wrapped inside a `java.lang.reflect.InvocationTargetException` (from which you can then obtain the original exception).
- All methods that return type information, for example `getParameterTypes()`, return `Class` objects that represent this type. Even `void` has a `Class` object of type `java.lang.Void` to represent it.

Table 8.3 The primitive types and their corresponding wrappers

Primitive type	Wrapper class
boolean	java.lang.Boolean
char	java.lang.Character
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

The `Method` class provides the functionality we need to call any method with whatever arguments are necessary for a given class. This class will be very useful as we build our JavaBean tags.

Array class

The `Array` class offers functionality for manipulating arrays of an unknown type. We'll forgo a deeper discussion of this class since the tags in this chapter won't need to use it.

Constructor class

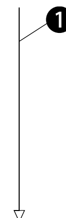
`Constructor` class represents the constructor of a JavaBean, including any and all parameters to it (much like `Method` class). This class will not be used in our tags either so, once again, we'll forgo discussing it here.

Using reflection: QueryRequestTag

To better understand reflection, let's develop a tag that uses the reflection API. The tag will call some methods (to fetch a request property) of the request (`HttpServletRequest`) object using reflection. The source for the `QueryRequestTag` is in listing 8.1.

Listing 8.1 Source code for the QueryRequestTag handler

```
package book.reflection;

import java.util.Hashtable;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;

import book.util.LocalStrings;
import book.util.ExTagSupport;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.jsp.JspException;

public class QueryRequestTag extends ExTagSupport {

    static Object []params = new Object[0];

    static Hashtable methods = new Hashtable();

    static LocalStrings ls =
        LocalStrings.getLocalStrings(QueryRequestTag.class);

    static {
        try {
            Class []p = new Class[0];
            Class reqc = HttpServletRequest.class;
            methods.put("method",
                        reqc.getMethod("getMethod", p));
            methods.put("queryString",
                        reqc.getMethod("getQueryString", p));
            methods.put("requestURI",
                        reqc.getMethod("getRequestURI", p));
```



```
            methods.put("userPrincipal",
                        reqc.getMethod("getUserPrincipal", p));
            methods.put("remoteUser",
                        reqc.getMethod("getRemoteUser", p));
        } catch(Throwable t) {
        }
    }

    protected String property = null;

    public void setProperty(String property)
    {
        this.property = property;
    }

    public int doStartTag()
               throws JspException
    {
        try {
            Method m = (Method)methods.get(property);
            if(null != m) {
                writeHtml(pageContext.getOut(),
                          m.invoke(pageContext.getRequest(),
                                   params).toString());
                return SKIP_BODY;
            } else {
                // Log and throw a JspTagException
            }

        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            // Log and throw a JspTagException
        } catch(InvocationTargetException ite) {
            // Exception in the called method
            // Log and throw a JspTagException
        } catch(IllegalAccessException iae) {
            // We are not allowed to access this method
            // Log and throw a JspTagException
        }
    }

    protected void clearProperties()
    {
        property = null;
        super.clearProperties();
    }
}
```

❶ **Obtains method objects from the `HttpServletRequest` class and stores them in a method cache for later use.** First we create an empty array of classes (see note) and procure an instance of `HttpServletRequest` from which to retrieve the methods.

Instead of the conventional approach, our tag extracts `Method` objects from the `HttpServletRequest` class and stores them in a hashtable. The key to the stored methods is the name of the property the method retrieves.

NOTE Note that we use `Class.getMethod()` to obtain the `Method` objects. `Class.getMethod()` expects two parameters: (1) The name of the method and (2) an array of class objects in which each entry in the array specifies the type of argument. In our case, using this approach was easy since all the needed methods have an empty argument list (array of size zero). This parameter is required since Java supports method overloading; meaning, a class may contain more than one method with the same name as long as the arguments to those methods are different.

---

❷ **Fetches the method object from the method cache; the key to the method cache is the property name** ❸ **Invokes the method using the current request variable; the method parameter list is an empty array of objects (no parameters)** When a request arrives, serving it is a breeze. All we need to do is fetch the method object that is stored in the cache, using the property name as the key. We use the `Method` class's `invoke()` with the current request object as the first parameter, and an empty argument list as the second. We use an empty argument list since none of the methods we are calling takes any arguments.

❹ **Handles the `InvocationTargetException`** If the invoked method throws an exception, we will need to handle it here.

The implementation of `QueryRequestTag` as seen in listing 8.1 is very different from what might have been expected had we worked without reflection. A conventional implementation of this tag would have taken the property name and performed a few `if-else` statements based on its value until it knew the method to use, and then it would call that method. Using reflection completely changes this algorithm.

What did we gain? We could implement this with a simple `if-else` statement. We gained extensibility! Suppose that we want to add new property for the tag to handle—simply add the code to fetch and store the `Method` in the method cache, and we're finished. In the case of this tag, the work to support a new method with reflection isn't much more (if any) than the work it takes to add another condition to an `if-else` statement. To further appreciate reflection, imagine if our tag did not store methods in a hashtable and, instead, simply looked for the methods by name at runtime. This approach would allow our tag to call any get method on the request. If the Servlet API were updated to add new properties to the request

object, our tag would still be usable (without changes or even recompilation) since it would always look for a method with the name that was passed as the property attribute. This type of flexibility can only be achieved by using the reflection API. We now see how this flexibility can be applied to JavaBeans.

8.2 JavaBeans and reflection

The topic of JavaBeans has books devoted to the subject, so we needn't delve into its finer points. We will only mention issues that directly affect the tags we develop in this chapter; namely, JavaBean properties, introspection, and what all this has to do with reflection.

What then are JavaBeans? In a nutshell, a JavaBean is merely a Java class. JavaBeans conventions are the de facto development conventions (as introduced by JavaSoft) for Java components. These conventions define how a JavaBean is to expose its properties and events. JavaBeans publish properties and any events they provide through a strict method signature pattern, making these method names predictable so that Java development and debugging tools may easily use the reflection API to learn about the bean and offer a visual interaction with it. When a tool uses reflection to analyze a bean in this way, we call it introspection. The benefit of building Java components that adhere to JavaBeans conventions is that they are guaranteed to work well with any of the multitude of JavaBean supporting software tools available.

8.2.1 Tags and JavaBeans

Most interactions between our tags and beans will revolve around fetching data from the bean and presenting it. Typically, bean interaction will involve a JSP getting the value of some property of a bean and displaying that value to the user. In light of this, we forgo discussing the second role of the JavaBean standard we mentioned, which is defining how events are specified. Primarily, our tags are concerned with two bean-related issues:

- Introspecting the beans to find the properties and get the methods that these tags should call for property retrieval.
- Calling these methods with the correct set of parameters.

The next two sections deal with the properties of JavaBeans and introspecting them.

8.2.2 JavaBeans properties

What makes a JavaBean unique is that it conforms to specific criteria regarding how it exposes its properties and events. What exactly are properties? *Properties* are attributes of the JavaBean, something in its state or functionality that the bean

wishes to expose. The code that uses the bean takes advantage of the properties (reads them or modifies them) by calling some of the bean's methods.

Imagine that you want to represent the data you have about a computer user in a bean, so that an administration program will be able to manipulate these users. Table 8.4 lists attributes each user might have.

Table 8.4 Sample User attributes and related properties

User Attribute	Description	JavaBean property
Name	The user's name	name
Family name	The user's family name	family
Password	The password that the user needs to enter when it logs into the computer	pass
Username	The user's login name	username
Groups	An array of user groups in which the user is a member	groups
User state	Specifies whether or not a user is logged in	state

To follow the JavaBean standard, each of the attributes described in table 8.4 will map into a bean property, and the methods that expose these properties in our UserBean would resemble the following code fragment.

```
public class UserBean implements java.io.Serializable {

    public String getName() { ... }

    public void setName(String name) { ... }

    public String getFamily() { ... }

    public void setFamily(String family) { ... }

    public String getPassword() { ... }

    public void setPassword(String pass) { ... }

    public String getUsername() { ... }

    public void setUsername(String name) { ... }

    public String getGroups(int index) { ... }
    public String []getGroups() { ... }

    public void setGroups(int index,
                          String group) { ... }

    public void setGroups(String []groups) { ... }

    public int getState() { ... }
};
```

When we look into these properties we may see differences between them. Some properties such as the user's password are read-write, meaning we can read their value as well as modify them. The user's state, however, is a read-only property; there is no meaning to set the value of this property since it represents something beyond our control (user logged in or logged out). The `groups` property is an array, since a user may belong to several groups, and we can access the different groups via an index. Our other properties are single values that do not need an index. The JavaBeans specification differentiates between read-only, write-only, and read-write properties, as well as indexed and nonindexed properties.

The differences between read-only, write-only, and read-write properties are manifested in our Java code through the types of methods we use for each. Each property can have a property `setter` method, `getter` method, or both. A property with only a `getter` method is said to be *read-only*, a property with only a `setter` method is said to be *write-only*, and a property with both methods is *read-write*.

NOTE The `state` property has only a `getter` method. This means that the `state` is read-only.

Indexed properties are handled by providing `getter` and `setter` methods that accept an index parameter. In this way the bean user accesses discrete values of the property by using their index within the property array. The bean may also provide array-based `setters` and `getters` to allow the bean user to set the whole property array.

NOTE The `group` property implements the indexed access with an integer as an index. One can consider using types other than an integer to index a property (for example, a string) but the JavaBeans specification is vague in the issue of property indexing using noninteger values. We also provide a means for the bean user to get the groups array in a single method call. Both method patterns (array `getter/setter` and indexed `getter/setter`) are permitted by the JavaBeans specification.

This clarifies properties and how the user of the JavaBean manipulates the bean's properties using `setters` and `getters`. However, how can the bean user find out about the different properties or their methods? As helpful as the beans might be, we cannot use them without knowing what methods to use to access the different properties. We answer this in our next section.

8.2.3 JavaBeans introspection

Recall that introspection is the process by which a JavaBean is analyzed, typically by a development tool, through reflection, for the purpose of determining its properties and events. The available properties as well as their `setter` and `getter` methods are discoverable by using introspection as defined in the JavaBeans specification. Introspection requires cooperation between the bean developer, who provides properties information, and the JavaBeans introspector, that reads this information and presents it to the user in a programmatic manner.

8.2.4 Properties and introspection

The simplest way for a developer to specify properties and their associated methods is to use the special JavaBean properties method naming conventions and parameter signatures in his or her JavaBeans.

Simple properties (nonindexed)

According to the JavaBean specification, either of the following methods can identify a simple property named `age` of type `int`:

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

Note that to conform to the JavaBeans standard, we've defined methods whose names are `getProperty()` and `setProperty()` wherein `property` is the name of the property to manipulate; the first character constructing the property name is capitalized (in this case, `age` becomes `Age`). The presence of `getAge()` means that the property called `age` is readable, while the presence of `setAge()` means that `age` is writable. This naming convention applies when the property is of any data type whatsoever, except `boolean`. In such a case, the `setter/getter` method looks like the following

```
public boolean isFoo();
public void setFoo(boolean foo);
```

The `getter` method name was changed to reflect the fact that we are making a query on a value that can have only true or false values (by changing `get` to `is`).

Indexed properties

Indexed properties are specifiable in one of two ways. One way is by having an array type as the input and output to the `setter` and `getter`, respectively. This approach is presented in the following code fragment.

```
public Bar[] getFoo();
public void setFoo(Bar[] foo);
```

Another way is by having setter and getter methods that take an index parameter (e.g., the next code fragment shows a setter/getter pair with an integer index).

```
public Bar getFoo(int index);
public void setFoo(int index, Bar foo);
```

Either of these index property coding conventions will suffice to inform an introspecting program of the existence of an indexed property of type Bar[] and with the name foo.

BeanInfo interface

This coding convention approach provides an easy way to inform the system of the existence of properties and their methods. But what happens if we want to provide more explicit property information? In this case, the bean author provides an implementation of the BeanInfo interface. A BeanInfo object allows the JavaBean author to provide additional information about a bean, ranging from the icon that represents it to the properties and events it exposes. If a JavaBean author opts not to create a BeanInfo object for a bean and uses the coding convention approach instead, a BeanInfo is automatically created for the class during the introspection and holds on the information that is accessible from the coding conventions. In fact, as we will soon see in code, the BeanInfo interface is a crucial component of the introspection process and, therefore, will be used by our custom tags to learn about the Beans with which they are interacting.

How introspection works

The introspection process is provided through a class called java.beans.Introspector whose methods provide control over the introspection process as well as methods to obtain BeanInfo objects for any JavaBean. The tags in this chapter will be constructed to use an Introspector to get a BeanInfo object for a particular JavaBean, in order to learn about and manipulate its properties. To reach the crux of this long story, let's look at getSetPropertyMethod() (listing 8.2), whose job it is to find the setter method of a property in a certain JavaBean (for simplicity, the method does not work on indexed properties).

Listing 8.2 Source code of a method that uses introspection to find a property setter

```
package book.util;

import java.util.Hashtable;

import java.beans.BeanInfo;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.beans.IndexedPropertyDescriptor;
import java.beans.IntrospectionException;

import java.lang.reflect.Method;

public class BeanUtil {

    // Snipped some of the code...
    /*
     * We are not dealing with indexed properties.
     */
    public static Method
        getSetPropertyMethod(Class claz,
                             String propName)
            throws IntrospectionException,
                   NoSuchMethodException
    {
        Method rc                  = null;
        BeanInfo             info = Introspector.getBeanInfo(claz);         ❶
        PropertyDescriptor[] pd   = info.getPropertyDescriptors();
        if(null != pd) {
            for(int i = 0; i < pd.length;  i++) {                           ❷
                if(propName.equals(pd[i].getName()) &&
                   !(pd[i] instanceof IndexedPropertyDescriptor)) {
                    Method m = pd[i].getWriteMethod();               ❸
                    if(null == m) {
                        continue;
                    }
                    Class[]params = m.getParameterTypes();
                    if(1 == params.length) {
                       rc = m;
                        break;
                    }
                }
            }
        }
        if(null == rc) {    ❹
            throw new NoSuchMethodException();
        }

        return rc;
    }
}
```

❶ **Gets an array of the property descriptors of this class** Listing 8.2 shows an elementary example of bean property introspection that covers all the important introspection issues. The first step in `getSetPropertyMethod()`, as in any method that performs some type of bean introspection, is to get the properties descriptors (or events, depending on what you want to find). To get to properties descriptors we use the built-in bean `Introspector` in two phases; the first one fetches a `BeanInfo` for the class, and later obtains the `PropertyDescriptor` array from `BeanInfo`. The `PropertyDescriptor` interface (as its name implies) provides methods to retrieve logical information about JavaBean properties. The obtained array provides information on all the properties as identified by the `Introspector`, so we can now iterate this array and learn about the bean's properties.

❷ **Iterates over all the properties in the class and looks for a property with a nonindexed matching name** A simple `for` statement will suffice; while iterating on the array we can check each of the properties as represented by a `PropertyDescriptor` object.

NOTE There are two `PropertyDescriptor` types: the one that provides information on nonindexed properties, and `IndexedPropertyDescriptor` that extends `PropertyDescriptor` to provide information on indexed properties. The main difference between them is that `IndexedPropertyDescriptor` also provides a method informing us of the index variable's type.

Since (in this example) we are only looking for a specific named, nonindexed property, we will ignore all other properties and look only at simple properties.

❸ **Found a property, performs a sanity check over the mehod. Do we have a method (meaning, is this property writable)? Does the method accept only a single parameter (the value to set)?** When we find a property of the type we want, we need to verify that the method we are seeking exists (maybe the property is read-only?). Thus we get the `setter` method from `PropertyDescriptor` and check it out. (We did not have to check the number of method arguments.)

❹ **We could not find a matching property** The method we were looking for does not exist. We should notify the user by throwing an exception.

We have outlined some of the basics of reflection, and more specifically, JavaBean introspection. We'll see more code examples of introspection as we develop our tags in the next section. If, at this point you feel less than entirely comfortable with the topic of introspection and reflection, that's all right. Only the most rudimentary grasp is required to comprehend our custom tags. If you are, however, interested in learning more about reflection, take a look at JavaSoft's tutorial on the subject

which is available online at http://java.sun.com/docs/books/tutorial/reflect/index.html.

Now, to the main event of this chapter: writing our JavaBean tags.

8.3 The Show tag

Our first JavaBeans-related tag is going to improve upon the standard `<jsp:getproperty>` tag by providing a JavaBeans property `getter` tag with the following enhancements:

- No need for previous bean declaration through a `<jsp:useBean>` tag (or any other tag).
  This makes it much easier to use the tag.
- Accessibility to all types of properties, including indexed properties with possible index type of string and integer.
  The inability of `<jsp:getproperty>` to access indexed properties cripples the use of the property `getters`, and string indices are very important in the web arena.
- Bean object specification either through name and scope or using a runtime expression.
  We can use this tag with values created within scriptlets.

As we have a rather high level of expectation here, our implementation will be rather involved. For example, our tag will let JSP authors specify the JavaBean for the tag to use in two ways:

- By specifying the bean's name and scope explicitly in an attribute
- By specifying the bean as a result of a runtime expression.

Since our tag must confirm that the JSP author properly uses one of these options, we need to use a `TagExtraInfo` object to verify the tag's attributes (a technique we saw in chapter 6). We also have the choice of allowing the tag to support indexed properties, which can become particularly tricky when a method is overloaded with different indices (such as a `String` index and an `int` index into the same property). In such a case, we do not want one tag attribute to specify the `String` index and a different one to specify the `int` index, so we need to design a way to place these two index value types into a single attribute. We'll soon see how we tackle these ambitious features.

8.3.1 *Components of the tag*

The implementation of our new property `getter` tag included using four Java classes (table 8.5).

Table 8.5 The classes comprising the bean getter tag

Java class	Description
`BeanUtil`	A utility class that performs the JavaBeans's introspection.
`ReflectionTag`	A tag whose role is to collect all our object and properties attributes and fetch the value pointed by them. This way we can reuse the logic in this class (by making it the base class) in other tags that require object property access through reflection.
`ShowTag`	A tag that takes the value fetched by our basic tag and prints it to the user. This is a relatively simple tag since most of the work is done in `ReflectionTag` from which it inherits.
`ReflectionTagExtraInfo`	A `TagExtraInfo` class to verify the tag's attributes. Since we want the tag to be very flexible, most of the attributes are not mandatory and some are applicable only in the presence of other attributes (for example, a scope attribute is not applicable without an object name attribute). This `TagExtraInfo` validates this complex attribute syntax.

Let's take a close look at the code for each of these components in order to gain a greater understanding of them.

BeanUtil

The first class we use to compose our `Show` tag is `BeanUtil` which is a utility class that will do the introspection and method caching for a bean. Its source is in listing 8.3.

Listing 8.3 Source code of JavaBeans utility class

```
package book.util;

import java.util.Hashtable;
import java.beans.BeanInfo;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.beans.IndexedPropertyDescriptor;
import java.beans.IntrospectionException;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;

public class BeanUtil {
```

```
static LocalStrings ls =
            LocalStrings.getLocalStrings(BeanUtil.class);

public static final Object []sNoParams = new Object[0];
public static Hashtable sGetPropToMethod = new Hashtable(100);

public static Object
    getObjectPropertyValue(Object obj,
                           String propName,
                           Object index)
        throws InvocationTargetException,
               IllegalAccessException,
               IntrospectionException,
               NoSuchMethodException
{
    Method m = getGetPropertyMethod(obj,                 ❶
                                    propName,
                                    null == index ?
                                    null: index.getClass());
    if(null == index) {
        return m.invoke(obj, sNoParams);                 ❷
    } else {
        Object []params = new Object[1];                 ❸
        params[0] = index;
        return m.invoke(obj, params);
    }
}

public static Method
    getGetPropertyMethod(Object obj,
                         String propName,
                         Class  paramClass)
        throws IntrospectionException,
               NoSuchMethodException
{
    Class     oClass = obj.getClass();
    MethodKey key = new MethodKey(propName,                       ❹
                                  oClass,
                                  paramClass);
    Method rc = (Method)sGetPropToMethod.get(key);
    if(rc != null) {
        return rc;
    }
    BeanInfo           info = Introspector.getBeanInfo(oClass);     ❺
    PropertyDescriptor[] pd = info.getPropertyDescriptors();
    if(null != pd) {
        for(int i = 0; i < pd.length;  i++) {
            if(pd[i] instanceof IndexedPropertyDescriptor) {      ❻
                if(null == paramClass ||
                   !propName.equals(pd[i].getName())) {
                    continue;
                }
```

```
                    IndexedPropertyDescriptor ipd =                  ❼
                        (IndexedPropertyDescriptor)pd[i];
                    Method m = ipd.getIndexedReadMethod();
                    if(null == m) {
                        continue;
                    }
                    Class[]params = m.getParameterTypes();
                    if((1 == params.length) &&
                        params[0].equals(paramClass)) {
                        rc = m;
                        break;
                    }
                } else {
                    if(null != paramClass ||     ❽
                       !propName.equals(pd[i].getName())) {
                        continue;
                    }
                    rc = pd[i].getReadMethod();
                    break;
                }
            }
        }

        if(null == rc) {
            StringBuffer methodName = new StringBuffer();     ❾
            methodName.append("get");
            methodName.append(propName.substring(0,1).toUpperCase());
            methodName.append(propName.substring(1));
            if(null == paramClass) {
                rc = oClass.getMethod(methodName.toString(),
                                      new Class[0]);
            } else {
                rc = oClass.getMethod(methodName.toString(),
                                      new Class[] {paramClass});
            }
        }
        if(null == rc) {
            // No such method; throw an exception
        }
        sGetPropToMethod.put(key, rc);     ❿
        return rc;
    }
}
```

❶ **Finds the needed method** ❷ **Invokes a nonindexed property** ❸ **Invokes an indexed property** The utility class exports two methods: The first accepts an object, property name, and index, and returns the desired property value from the object by using introspection. The second method introspects the object's class and retrieves the method required to get the property. The first method is not especially

interesting; we already know how a method is called using reflection and the only new issue here is that you see how to provide parameters to the invoked method using an array of objects.

❹ ❿ **Looks for the method in the introspected methods cache** The second method presents several new ideas, starting with the use of a method cache, continuing with introspecting indexed properties, and ending with our own low-level introspection when the default `Introspector` fails us. The method cache was added when we found out how time-consuming introspection is. In fact, in pages loaded with reflection, adding the method cache gave the pages a 33% performance boost. It's important to remember that the key to the cache needs to be a combination of the object's class, property name, and method parameters. This is a rather complex key, so a new method key object was created (when caching methods for indexed `setters`, the key is even more involved). If we fail to find a method in the cache, we will introspect it and, when complete, place it in the cache.

❺ **Method not in the cache, start introspecting.**

❻ **Skip methods that do not match our needs** ❼ **Validate that this method matches our indexed property** ❽ **Skip methods that do not match our needs** Introspecting the class is different from introspection code in listing 8.2, mainly because we now introspect indexed properties. We iterate over the properties descriptor array and differentiate between indexed (instances of `IndexedPropertyDescriptor`) from non-indexed properties, and then check on the indexed property method. The check includes testing the parameter list of the indexed property, because a certain method in Java may be overloaded. For example, think of the following class:

```
class SuperHero{
   Power getSuperPower(int i);
   Power getSuperPower(String s);
}
```

We may want to inspect `getSuperPower(String)`, yet the `Introspector` will probably give us the descriptor of `getSuperPower(int)`.

We will then need to skip this descriptor and hope our luck is better elsewhere.

❾ **Method was not found using the default introspector; try our own low-level introspection** We are finally finished with the property introspection, yet we may not have found the property method. The above example, wherein a specific method has two index types, is a good example of such a case (no, it is not a bug in the default `Introspector`, just our desire to attain more than the simple indexes regulated in JavaBeans). To overcome cases in which the default `Introspector` fails to find the needed method, we employ elementary low-level reflection to look for a method

matching our property name and parameter types. If we find such a method, we assume that it is the one we seek.

⑩ **Places the newly introspected method in the cache.**

ReflectionTag

The class presented in listing 8.3 had nothing to do with tags (in fact, you can use it in any program that employs JavaBeans). The next class, `ReflectionTag`, is an abstract tag handler that integrates the JavaBeans reflection capabilities to the tag's world:

Listing 8.4 Source code for the ReflectionTag base class

```
package book.reflection;

import java.beans.IntrospectionException;
import java.lang.reflect.InvocationTargetException;
import book.util.LocalStrings;
import book.util.ExTagSupport;
import book.util.BeanUtil;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;

public abstract class ReflectionTag extends ExTagSupport {

    public static final String PAGE_ID = "page";
    public static final String REQUEST_ID = "request";
    public static final String SESSION_ID = "session";
    public static final String APPLICATION_ID = "application";

    static LocalStrings ls =
        LocalStrings.getLocalStrings(ReflectionTag.class);

    protected Object obj = null;          ❶
    protected String objName = null;
    protected String objScope = null;

    protected String property = null;     ❷
    protected String index = null;

    public void setObject(Object o)
    {
        this.obj = o;
    }

    public void setName(String name)
    {
        this.objName = name;
    }

    public void setScope(String scope)
    {
        this.objScope = scope;
    }
```

```
public void setProperty(String property)
{
    this.property = property;
}

public void setIndex(String index)
{
    this.index = index;
}

public int doStartTag()
           throws JspException
{
    processObject(getPointed());    ❸
    return SKIP_BODY;
}

protected Object getPointed()
           throws JspException
{
    Object value = (null == obj ?
                    getPointedObject(objName, objScope) :
                    obj);                                       ❹
    if(null != property) {
        value = getPointedProperty(value);    ❺
    }
    return value;
}

protected Object getPointedObject(String name,
                                  String scope)
           throws JspException
{
    Object rc = null;
    if(null != scope) {
        rc = pageContext.getAttribute(name,
                                      translateScope(scope));
    } else {
        rc = pageContext.findAttribute(name);
    }
    if(null == rc) {
        // Log and throw a JspTagException
    }

    return rc;
}

protected int translateScope(String scope)
           throws JspException
{
    if(scope.equalsIgnoreCase(PAGE_ID)) {
        return PageContext.PAGE_SCOPE;
    } else if(scope.equalsIgnoreCase(REQUEST_ID)) {
```

```
            return PageContext.REQUEST_SCOPE;
        } else if(scope.equalsIgnoreCase(SESSION_ID)) {
            return PageContext.SESSION_SCOPE;
        } else if(scope.equalsIgnoreCase(APPLICATION_ID)) {
            return PageContext.APPLICATION_SCOPE;
        }

     // Log and throw a JspTagException
    }

    protected Object getPointedProperty(Object v)
                throws JspException
    {
        try {
            Object indexParam = null;
            if(null != index) {
                if(index.startsWith("#")) {    ❻
                    indexParam = new Integer(index.substring(1));
                } else {
                    indexParam = index;
                }
            }
            return BeanUtil.getObjectPropertyValue(v,    ❼
                                                       property,
                                                       indexParam);
        } catch(InvocationTargetException ite) {
            // Log and throw a JspTagException
        } catch(IllegalAccessException iae) {
            // Log and throw a JspTagException
        } catch(IntrospectionException ie) {
            // Log and throw a JspTagException
        } catch(NoSuchMethodException nme) {
            // Log and throw a JspTagException
        }
    }

    protected void processObject(Object v)    ❽
                throws JspException
    {
    }

    protected void clearProperties()
    {
        obj      = null;
        objName  = null;
        objScope = null;
        property = null;
        index    = null;
        super.clearProperties();
    }
}
```

❶ **Points to the object whose property we want to get** We can have either the object itself or its name and scope (optional). There are two ways to specify the object used by the tag: one way is to set the object as a runtime expression value, the other is to specify the name and the scope. These two object specification methods are mutually exclusive, and our TagExtraInfo implementation should take care of this. But we are getting ahead of ourselves.

❷ ❻ **Refers to the property name and the index (optional) in this property** The property value is specified by two attributes: the property name and an index into the property. The index is not mandatory and the tag can handle cases in which the index is not specified. The index, you'll recall, may be a string or an integer; but how can we specify two different types using a single attribute? We cheat! We specify the integer within a string, but prefix its value with a '#' so that the tag knows that the value represents an integer. Why are we giving an advantage to the string? Because strings are far more useful as an index when we are creating web pages. In most cases, we will index our properties using a string, as it was felt that string indexing should be easily done.

❸ **`doStartTag()` fetches the property value and lets the tag process it using `processObject()`.**

❹ **Fetches the object** If the object was configured as a runtime value, either use it, or get a reference to it using the name and the scope.

❺ **As soon as we receive the object, fetches the property value.**

❻ **A '#' prefix denotes an integer** Translates the string to an integer.

❼ **Gets the propery value using the beans utility class.**

❽ **`processObject()` is a method that an extending class can overide to process the value of our object property** As previously stated, `ReflectionTag` is an abstract class whose job is to provide access through reflection to properties in the JSP scripting objects. In this spirit, we defined an empty method named `processObject()` that can be overridden by an extending class, whose only goal is to manipulate the property value. Implementing `processObject()` is not mandatory and, for many cases, it may be better to override `doStartTag()` and use the method `getPointed()` directly; however, for the purpose of our new `ShowTag` (listing 8.5), overriding `processObject()` is enough.

ShowTag

The `ShowTag` handler is the handler for the tag to be used by the JSP author, and inherits from `ReflectionTag` to make use of its introspection work. This tag then retrieves the value for a property and prints it to the user.

Listing 8.5 Source code for the ShowTag handler class

```
package book.reflection;

import book.util.LocalStrings;
import javax.servlet.jsp.JspException;

public class ShowTag extends ReflectionTag {

    static LocalStrings ls =
        LocalStrings.getLocalStrings(ShowTag.class);

    protected void processObject(Object v)      ❶
               throws JspException
    {
        try {
            writeHtml(pageContext.getOut(), v.toString());
        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            // Log and throw an exception
        }
    }
}
```

❶ **Overrides `processObject()` to print the property value.**

TagExraInfo for ShowTag

The last portion of code left unseen is the `TagExtraInfo` that we shall attach to the `ShowTag`. In fact, since `ShowTag` does not add any new attribute or syntactic constraints, we can actually take a `TagExtraInfo`, as developed for `ReflectionTag`, and use it for `ShowTag` (listing 8.6).

Listing 8.6 Source code for the ReflectionTagExtraInfo class

```
package book.reflection;

import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;
import javax.servlet.jsp.tagext.VariableInfo;

public class ReflectionTagExtraInfo
    extends TagExtraInfo {

    public boolean isValid(TagData data)
    {
        Object o = data.getAttribute("object");
        if((o != null) && (o != TagData.REQUEST_TIME_VALUE)) {    ❶
            return false;
        }
```

```
            String name = data.getAttributeString("name");
            String scope = data.getAttributeString("scope");

            if(o != null) {
                if(null != name || null != scope) {
                    return false;
                }
            } else {
                if(null == name) {
                    return false;
                }

                if(null != scope &&
                   !scope.equals(ReflectionTag.PAGE_ID) &&
                   !scope.equals(ReflectionTag.REQUEST_ID) &&
                   !scope.equals(ReflectionTag.SESSION_ID) &&
                   !scope.equals(ReflectionTag.APPLICATION_ID)) {
                    return false;
                }
            }

            if((null != data.getAttribute("index")) &&
               (null == data.getAttribute("property"))) {
                return false;
            }
            return true;
        }
}
```

❶ **The object attribute must be the product of a runtime expression.**

❷ **If the object was provided through a runtime expression, the name and scope attributes should not be used.**

❸ **If the object is not provided through a runtime expression, we must provide a variable name.**

❹ **The scope value must be one of the four defined scopes.**

❺ **We cannot provide an attribute index without specifying a property.**

ShowTag's TLD

All that is left for us to do before using `ShowTag` in a JSP file is to create a TLD (listing 8.7). Note in this listing that not all the attributes are mandatory. This loss of control is required by the flexible functionality that was required from the tag.

Listing 8.7 Tag library descriptor entry for ShowTag

```
<tag>
    <name>show</name>
```

```
    <tagclass>book.reflection.ShowTag</tagclass>
    <teiclass>book.reflection.ReflectionTagExtraInfo</teiclass>
    <bodycontent>empty</bodycontent>
    <info>
        Show a certain object property value.
    </info>
    <attribute>
        <name>object</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>scope</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>index</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>property</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

ShowTag in action

We can create a JSP file that uses our new tag to show bean properties; the JSP driver (listing 8.8) uses our tag to explore the values present in the request and response objects (this is possible since both objects are JavaBeans).

Listing 8.8 JSP file that uses ShowTag

```
<%@ page import="java.util.*" %>
<%@ page errorPage="error.jsp" %>
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/beans-taglib"
    prefix="bean" %>
<html>
<body>
<%-- javax.servlet.jsp.jspRequest is the JSP attribute
```

```
     name of the request object
  --%>
<br> <bean:show name="javax.servlet.jsp.jspRequest"          ❶
                property="locale"/>

<table>
<tr>
<th> Header </th> <th> Value </th>
</tr>

<% Enumeration e = request.getHeaderNames();           ❷
    while(e.hasMoreElements()) {
        String name = (String)e.nextElement();
%>
<tr>

<td> <%= name %> </td>

<td>
    <bean:show object="<%= request %>"                 ❸
               property="header"
               index="<%= name %>"/>
</td>
</tr>
<%
    }
%>
</table>
<%-- javax.servlet.jsp.jspResponse is the JSP attribute
     name of the response object
  --%>
<br> <bean:show name="javax.servlet.jsp.jspResponse"         ❹
                scope="page"
                property="committed"/>

</body>
</html>
```

❶ **Shows the request locale.**

❷ **Creates a table of header names and values. Starts by enumerating the header names.**

❸ **Prints the header value** We are using runtime expression to define the object we work with and to provide the index into the header property.

❹ **Gets the value of the response's committed property** It should be `false`, since writing back to the user has not started.

Throughout the entire JSP file we never defined any of the objects used as a JavaBean; `ShowTag` will treat any object we give it as a bean, and this lets us take the regular request and response objects and use them without prior definition. This

sample JSP file presents different usage patterns for the tag. Sometimes it is employed in an elementary way. For example, we only specify the JSP attribute name associated with the object (optionally the scope) and the name of the property. There might be a case, however, in which the tag is used with runtime expressions and indexed properties; in fact, we produce a fine table with the header names and values by using the tag with the header indexed property.

This section showed how easy it is to print the values of JavaBeans properties back to the response flowing to the user. Now we move on to build the second tag of our library, which allows us to export new JavaBeans from our tags and have them serve as scripting variables in the page.

8.4 Exporting bean values from tags

Exporting new scripting variables from a tag is a handy feature. For example, a JDBC connection tag can connect to a database and export a JDBC connection as a scripting variable so that JSP scriptlets further down the page may use the connection object. Though useful, exporting a new scripting variable is more than a minor maneuver for the JSP engine; it first needs to know:

- The Java type of the new scripting variable so that a correct Java scripting variable will be declared for the newly exported object.
- The duration (scope) of the scripting variable. Sometimes you want your Java scripting variable to last until the end of the page; in other cases you may want it to exist within the body of the tag. The JSP environment needs to be informed of that.
- The name of the scripting variable.

This reflective information must arrive at the JSP environment in order to take advantage of it while translating the JSP file to a servlet.

The methods to provide this information were defined in the JSP specification, which we will present now. Next, we take a look at a tag whose job it is to export JavaBean property values as new JSP scripting variables.

8.4.1 Informing the runtime of exported scripting variables

The JSP specifications define a simple way to inform the JSP runtime of the exported scripting variables, by overriding yet another method in `TagExtraInfo`. Up until now, the only method we overrode in `TagExtraInfo` was `isValid()`, which we used to validate tag attribute syntax and values. The `TagExtraInfo` class also allows you to indicate any scripting variables your tag will export by overriding `getVariableInfo()`.

The signature for `getVariableInfo()` is presented below:

```
public VariableInfo[] getVariableInfo(TagData data);
```

As you can see, `getVariableInfo()` accepts a `TagData` object which stores the values for the tag's attributes. The method returns an array of objects of type `VariableInfo`, whose methods and static variables are presented in listing 8.9.

Listing 8.9 Methods and static fields in VariableInfo

```
public class VariableInfo {

    public static final int NESTED = 0;
    public static final int AT_BEGIN = 1;
    public static final int AT_END = 2;

    public VariableInfo(String varName,
                        String className,
                        boolean declare,
                        int scope) { ... }

    public String getVarName() { ... }
    public String getClassName() { ... }
    public boolean getDeclare() { ... }
    public int getScope() { ... }
}
```

The job of a `VariableInfo` object is to provide variable information, starting with the variable's name and ending with its scope. A developer wishing to export scripting variables from a tag should first override `getVariableInfo()` and, within the method, use the tag's attributes to decide the exact variables to be exported. Next, for each scripting variable, the JSP author should create a `VariableInfo` instance with the desired variable name, type, and scope, and return an array containing these `VariableInfos`. Based on this information, the JSP page translator will emit Java code to implement the newly exposed scripting variables in a way that scriptlets and other JSP entities can access them.

Constructing a VariableInfo

How de we create a `VariableInfo` object? The parameters to the `VariableInfo`'s constructor have the following semantics:

- The `varName` parameter has two roles. It informs the JSP runtime of the name under which the newly generated JSP attributes (that hold the value of the new scripting variable) are kept; and it tells the JSP runtime the name of the variable it should emit into the generated servlet code. In the second role, the name of the exported scripting variable should match the rules of the scripting

language employed in the page; for example, if the language is Java, then the name value may not include the "." character.

- The `className` parameter specifies the type of the exported object. Currently, tags can only export objects, so this is the fully qualified class name. You cannot export primitive types such as `float` values (they should be wrapped in some Java object type such as `java.lang.Float`).
- By default, the JSP engine will declare our variable as a new scripting variable in the JSP. If we want, instead, to assign a value to an existing scripting variable with our tag, we set the `declare` parameter to `false`, informing the JSP runtime that this variable has already been declared and to not declare it as a new variable.
- The `scope` parameter specifies the scope of the scripting variable that the page translator needs to emit into the Java servlet being generated for the JSP file. The `scope` parameter specifies, for example, whether the scripting variable will be known through all of the generated servlet or only within the tag's body (more on this soon).

Scripting variable scope

The last parameter to the `VariableInfo`'s constructor is the variable's scope. The possible values for this parameter (the different scope types as defined in the JSP specification) are listed in table 8.6.

Table 8.6 Possible scope types and their uses

Scope name	Scope in the generated servlet	Use
NESTED	Between the starting and closing marks of the custom tag, as presented in figure 8.1.	The JSP runtime emits a variable declaration in the block of code between the calls to `doStartTag()` and `doEndTag()`. Such `NESTED` scope variables are very useful in tags that perform iterations and need an iteration index of some sort.
AT_BEGIN	Between the starting mark of the custom tag and the end of the JSP file, as presented in figure 8.1.	The JSP runtime emits a variable declaration so that the variable will live in the block of code between the calls to `doStartTag()` and the end of the JSP file. This way you can define scripting variables whose scope spans the entire JSP file and whose value is specified in `doStartTag()`.
AT_END	Between the ending mark of the custom tag and the end of the JSP file, as presented in figure 8.1.	The JSP runtime emits a variable declaration so that the variable will live in the block of code between the calls to `doEndTag()` and the end of the JSP file. This way you can define scripting variables whose scope spans the entire JSP file and whose value is specified in `doEndTag()`.

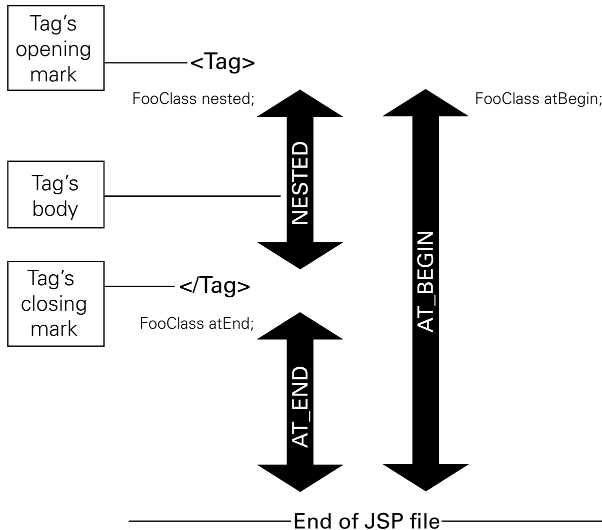


Figure 8.1 Scripting variable scopes illustrated

VariableInfo in action: TestTag example

Let's try to crystallize some of the information presented in this section with an example. We'll look at a simple `BodyTag` example called `TestTag` that exposes three scripting variables:

- sName—A variable of type `java.lang.String` with `ET_BEGIN` scope.
- iName—A variable of type `java.lang.Integer` with `ET_END` scope.
- fName—A variable of type `FooClass` with `NESTED` scope.

Let's look at the `TagExtraInfo` object we need to create for this class in order to export these three variables (listing 8.10).

Listing 8.10 Source code for TestTei

```
import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;
import javax.servlet.jsp.tagext.VariableInfo;

public class TestTei extends TagExtraInfo {

    public VariableInfo[] getVariableInfo(TagData data)
    {

        VariableInfo[] rc = new VariableInfo[3];
```

```
        rc[0] =  new VariableInfo("sName",
                                  "java.lang.String",
                                  true,
                                  VariableInfo.AT_BEGIN);
        rc[1] =  new VariableInfo("iName",
                                  "java.lang.Integer",
                                  true,
                                  VariableInfo.AT_END);
        rc[2] =  new VariableInfo("fName",
                                  "FooClass",
                                  true,
                                  VariableInfo.NESTED);

        return rc;
    }
}
```

This is an example of how to create and return an array of `VariableInfo` objects, each with its own type name and scope, in order to tell the JSP runtime engine the variables our tag will expose.

NOTE The JSP specification instructs tag developers to use a special attribute named `id` to let the user name the variable exported by the tag. (Usually a single tag will export only a single object.) The sample provided in this section does not follow this rule; in part because there are several different variables to export, and because we wanted to make it as simple as possible. Do your best to follow this instruction, meaning that if you want your tag `foo` to export a variable, let your user specify this variable name by using an attribute named `id` in the following manner: `<foo id="bar"/>`. The implications of these instructions are that your `TagExtraInto` implementation should look into the attribute's data, grab the value of the `id` attribute, and use it as the name (first parameter to the `VariableInfo`'s constructor) returned in the `VariableInfo`.

To see how the `TestTag` class affects the servlet code generated at translation time, let's look at the source code generated by Tomcat's JSP translator for `TestTag` when used with our `TestTag` class (listing 8.11).

Listing 8.11 Java that was generated for TestTag

```
TestTag testtag = new TestTag();                 ❶
testtag.setPageContext(pageContext);
testtag.setParent(null);
java.lang.String sName = null;          ❷
```

```
try {
    int rc = testtag.doStartTag();
    sName = (java.lang.String) pageContext.getAttribute("sName");      ❸
    if (rc == Tag.EVAL_BODY_INCLUDE)
        // Throw exception. TestTag implements BodyTag so
        // it can't return Tag.EVAL_BODY_INCLUDE
    if (rc != Tag.SKIP_BODY) {
        try {
            if (rc != Tag.EVAL_BODY_INCLUDE) {
                out = pageContext.pushBody();
                testtag.setBodyContent((BodyContent) out);
            }
            testtag.doInitBody();
            do {
                FooClass fName = null;                        ❹
                fName = (FooClass)
                pageContext.getAttribute("fName");

                sName = (java.lang.String)    ❺
                    pageContext.getAttribute("sName");

                // Evaluate body ...

            } while(testtag.doAfterBody() == BodyTag.EVAL_BODY_TAG);
            sName = (java.lang.String)    ❻
                pageContext.getAttribute("sName");
        } finally {
            if (rc != Tag.EVAL_BODY_INCLUDE)
                out = pageContext.popBody();
        }
    }
    if (testtag.doEndTag() == Tag.SKIP_PAGE)
        return;
} finally {
    testtag.release();
}
java.lang.Integer iName = null;                                      ❼
iName = (java.lang.Integer)pageContext.getAttribute("iName");
```

❶ Initializes the `test` tag. Tomcat does not pool its tag handlers, but rather instantiates them.

❷ Declaring `sName` outside of the tag body so that it will be known through the page.

❸ Setting a value into `sName` right after the call to `doStartTag()` to set an initial value.

❹ Declaring and initializing `fName`. Since it is a `NESTED` variable it will be declared and initialized once per iteration.

❺ Updating `sName`'s value each iteration (so the tag can modify it over and over again).

❻ Updating `sName`'s value one more time when we are bailing out of the body processing loop.

❼ Declaring and initializing `iName`; this is done only once, after the tag processing is done.

Each of the variables is generated within a Java scope that matches the defined variable scope; for example, `fName` is defined within the curly brackets of the body evaluation loop. The JSP runtime updates the values of the exported variables from the `PageContext` whenever one of the tag's methods is called so that the tag can modify the value of the scripting variables as often as possible.

Our next step will be to use the information in this section to develop a real-world tag that exports objects into the JSP environment.

8.4.2 The ExportTag

`ShowTag` that we developed earlier in this chapter took the value of the certain bean property and printed it to the response flowing to the user. This capability is handy, but what if the property you want to print is not a primitive value or some type with a reasonable string conversion method that can be easily printed to the user? Or, what if this property is a bean of its own and we only want to echo some of its properties to the user? In these cases, `ShowTag` falls short of meeting our needs and we require the help of some other tag to export the complex property value into the JSP environment where `ShowTag` can take this bean property and print it the way we want. For this purpose, `ExportTag` was developed.

`ExportTag` acts very much like `ShowTag` except, instead of printing the property to the response, it exports the property as a JSP scripting variable. This lets us use the variable in scriptlets, just as if we'd defined it in a scriptlet or utilized the `<jsp:useBean>` standard tag to define it. To facilitate this task, we implemented the two objects presented in table 8.7:

Table 8.7 The ExportTag implementation objects

Java class	Description
`ExportTag`	Extends `ReflectionTag` to export the value gathered through reflection. This tag has two additional attributes: `id` for the user to choose a name for an exported bean, and `type` for the user to specify a type for the exported object.
`ExportTagExtraInfo`	Extends `ReflectionTagExtraInfo` to add an implementation for the method `getVariableInfo()`, which grabs the value of the `id` and `type` attributes and returns a `VariableInfo` with the desired name and type as well as `AT_BEGIN` scope.

It may seems as though `ExportTag` is a waste of time; why not just improve `ShowTag` such that it will be able to print the property of a property? The answer is twofold: First, this improved `ShowTag` will be too complex (just how many attributes will it take to print an indexed property of an indexed property)? Second, `ExportTag` offers

clear advantages by being able to export a complex attribute (such as an enumerator) to be used later by our scriptlets. For example, the following JSP fragment:

```
<%
    Enumeration e = request.getHeaderNames();
%>
```

can be replaced by an alternative tag usage that looks similar to:

```
<beans:export id="e" type"enum"
              object="<%= request =>"
              property="headerNames"/>
```

Though this may be a matter of taste, the ability to grab complex properties that will be accessible for other JSP entities to work on (including other custom tags that do not know a thing about reflection) opens a great opportunity for synergy.

ExportTag handler

As a first step, let's look into `ExportTag` handler class (listing 8.12), which is actually the simpler part of the implementation of `ExportTag`.

Listing 8.12 Source code for the ExportTag handler class

```
package book.reflection;

import book.util.LocalStrings;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;

public class ExportTag extends ReflectionTag {

    static LocalStrings ls =
        LocalStrings.getLocalStrings(ExportTag.class);

    public void setType(String type)
    {
        // Unused, needed only for the translation phase
    }

    protected void processObject(Object v)
                throws JspException
    {
        pageContext.setAttribute(id,                          ❶
                                 v,
                                 PageContext.PAGE_SCOPE);
    }

    protected void clearProperties()
    {
        id = null;
        super.clearProperties();
    }
}
```

❶ Exports the object into the JSP runtime by setting its value into the page scope with the `id` serving as the name.

Listing 8.12 does not show anything new, but the property `setters` in `ExportTag` may seem strange. What happened to the `type` property (we didn't keep it) and why don't we need to implement a property `setter` for the `id` property?

The answer is easy. First, we do not use the `type` property during the service phase (only while translating the page), so keeping its value in the tag handler is not needed. As for the `id` attribute setter, the `TagSupport` base class implements a `setter` method for the property `id` (probably because it was regulated in the JSP specification as the recommended way to name the exported variables).

TagExtraInfo for ExportTag

The second class we implemented for the `ExportTag` is its `TagExtraInfo` implementation shown in listing 8.13. In its base, `ExportTagExtraInfo` should extend `ReflectionTagExtraInfo` to implement `getVariableInfo()`. This implementation is interesting because it shows how you can use the tag's attributes to define your exported variable parameters.

Listing 8.13 Source code for ExportTagExtraInfo

```
package book.reflection;

import java.util.Hashtable;
import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;
import javax.servlet.jsp.tagext.VariableInfo;

public class ExportTagExtraInfo extends ReflectionTagExtraInfo {

    static Hashtable types = new Hashtable();

    static {                                                      ❶
        types.put("iterator", "java.util.Iterator");
        types.put("enum", "java.util.Enumeration");
        types.put("string", "java.lang.String");
        types.put("boolean", "java.lang.Boolean");
        types.put("byte", "java.lang.Byte");
        types.put("char", "java.lang.Character");
        types.put("double", "java.lang.Double");
        types.put("float", "java.lang.Float");
        types.put("int", "java.lang.Integer");
        types.put("long", "java.lang.Long");
        types.put("short", "java.lang.Short");
    }

    public VariableInfo[] getVariableInfo(TagData data)
    {
        VariableInfo[] rc = new VariableInfo[1];
```

```
        rc[0] =  new VariableInfo(data.getId(),          ❷
                                  guessVariableType(data),      ❸
                                  true,
                                  VariableInfo.AT_BEGIN);
        return rc;
    }
    protected String guessVariableType(TagData data)
    {
        String type = (String)data.getAttribute("type");

           if(null != type) {
            type = type.trim();
            String rc = (String)types.get(type);          ❹

               if(null != rc) {
                return rc;
            }

            if(type.length() > 0) {
                return type;
            }
        }
        return "java.lang.Object";
    }
}
```

❶ ❹ **Prepares a translation table for the primitive types as well as a few shortcuts for common types** What is clear from `ExportTagExtraInfo` is that we go a long way to prevent users from specifying primitive types for the exported scripting variable! This is because for now, in JSP1.1 and 1.2, you are not allowed to export primitive types, only objects (less painful than you might think, but something to keep in mind). Other than that, take a look at how we used a `type` attribute to specify the exported bean's `type`. Whenever you export a bean from your tag, you will rarely know the exported type in advance (meaning we export an arbitrary bean) and, since the runtime object that we will reflect is not available during the translation, we will have to convey the type using some attribute (as demonstrated in listing 8.13). The last item to note here is that the `id` property receives special treatment in the `TagData` class which provides a special `getId()` method for easily obtaining the value of the `id` attribute (saving us a call to `data.getAttributeString("id")`).

❷ **Specifies the value of the `id` property as the name of the exported variable (as specified in the specification).**

❸ **Guesses the exported variable type using the type attribute.**

❹ **Looks for the type in the translation table. If it exists in the translation table, uses the lookup value.**

ExportTag's TLD

The final piece in the implementation of `ExportTag` is the tag library entry as presented in listing 8.14.

Listing 8.14 Tag library descriptor entry for ExportTag

```
<tag>
    <name>export</name>
    <tagclass>book.reflection.ExportTag</tagclass>
    <teiclass>book.reflection.ExportTagExtraInfo</teiclass>
    <bodycontent>empty</bodycontent>
    <info>
        Exports an object property into the JSP environment
    </info>
    <attribute>
        <name>id</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>type</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    ... Additional attributes as defined for the ShowTag
</tag>
```

The `id` and `type` attributes in listing 8.14 were both defined as nonruntime expression values for a simple reason: we need these values during the translation process. We did not make the `type` attribute mandatory for the `ExportTag`, because often the type of the exported variables is not that important; for example, when another reflection-driven tag is processing the exported object. In these cases, `ExportTagExtraInfo` provides a default type (`java.lang.Object`), and eases the job of the JSP developer that does not deal with the exact details.

ExportTag in action

The last step of the `ExportTag` tour is the JSP file that uses it, which is actually a modified version of the JSP file that drove `ShowTag`.

Listing 8.15 A JSP driver for ExportTag

```
<%@ page errorPage="error.jsp" %>
<%@ taglib
    uri="http://www.manning.com/jsptagsbook/beans-taglib"
    prefix="bean" %>
<html>
```

```
<body>
<table>
<tr>
<th> Header </th> <th> Value </th>
</tr>

<bean:export id="e"
             type="enum"
             object="<%= request %>"
             property="headerNames" />

<% while(e.hasMoreElements()) {
        String name = (String)e.nextElement();
%>
<tr>

<td> <%= name %> </td>

<td>
    <bean:show object="<%= request %>"
               property="header"
               index="<%= name %>"/>
</td>
</tr>
<%
    }
%>
</table>
</body>
</html>
```

The modified portion of the JSP driver is in bold and is relatively straightforward. However, using `ExportTag` is shaded by the fact that we need some (relatively complex) scriptlet to iterate over the exported enumeration. This iteration scriptlet makes the `ExportTag` seem both clumsy (we need to specify a type for the exported value) and useless (if we already have a scriptlet, why add this tag?). We will return to this sample again in chapter 10 when we implement an enumerator that reduces the need for the iteration scriptlet and overcomes this limitation.

8.5 Summary

We have seen how easy it is to integrate beans and tags to produce a winning combination. The tags in this chapter free the JSP developer from the need to use scriptlets such as the following:

```
<%= obj.getPropertyName("index") %>
<% ClassType t = obj.getPropertyName("index"); %>
```

thus reducing the amount of real Java code in our JSPs. By removing Java syntax from JSPs, we further our cause of decoupling presentation logic and business logic; and by making the syntax cleaner and easier, reduce the possibility of writing incorrect or error-prone code. Future chapters show how to perform conditioning and iteration through tags, and how the availability of these bean-property related tags makes it possible to write out JSP files with minimal coding. We will also use bean integration in other future tags (such as conditioning) for which the know-how acquired in this chapter will prove worthwhile.