

10

Iterating with tags

In this chapter

- Iterating with tags 101
- Universal iteration with tags (iterate anything!)
- Tag-only presentation of a shopping cart
- The JSP1.2 IterationTag

At the end of chapter 8, we used our newly created JavaBean tags to export an Enumeration which was then iterated over with a scriptlet. Let's take another look at this JSP fragment.

```
<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>
```

As you can see (note the highlighted code), although our JavaBean tags greatly reduce the need for scriptlets, we are still unable to avoid them when working with indexed JavaBean properties that have more than one value. In cases of multivalued properties (Enumerations, arrays, etc.) we typically want to loop through (*iterate*) and perform a function with each value in the property. Without a tag to handle this iteration, we're left using a scriptlet like the one here. This is unfortunate since we want to be able to provide our JSP authors with the ability to perform common functions on JavaBeans without prior knowledge of Java. Ideally, we'd like to offer them a very user-friendly JSP custom tag that would work for iteration.

Iteration, especially enumerating some value, can be very declarative, and, as we've seen, declarative tasks are easily performed with tags. For example, by using iteration tags we can modify the previous JSP fragment:

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

```
<iter:foreach id="name"
              type="String"
              object="<%= request %>"
              property="headerNames" />

<tr>
<td> <%= name %> </td>
<td>
    <bean:show object="<%= request %>"
               property="header"
               index="<%= name %>"/>
</td>
</tr>
<iter:foreach>
</table>
```

This is obviously quite an improvement.

Why should we bother creating special iteration tags when a two-line scriptlet hardly seems demanding for a Java developer? Again, we can't forget that the goal of building custom tag libraries is to make it possible for non-Java developers (presentation/HTML developers) to build complex sites. Though iteration using scriptlets may not be complex for the Java programmer, it does require the JSP developer to:

- Know how to iterate on different Java types—Enumerations, Iterators, arrays, and so forth. To further complicate the situation, iteration methods usually return an `Object` that the JSP developer will have to cast.
- Position the curly brackets in the correct location. If the JSP developer forgets a curly bracket, the JSP compilation will fail, usually with a relatively obscure error message.
- Maintain and debug yet another portion of Java code.

As a result, iteration tags are necessary to enhance the effectiveness of our JavaBean tags and to keep our JSPs scriptlet-free.

This chapter explores iteration with tags and shows how to build JSP custom tags that perform iteration for us. We'll start with a brief introduction to iterating with custom JSP tags and discuss their design principles; later, we will develop iteration tags to handle cases in which we wish to iterate over Java's common object containers.

NOTE In this chapter, you will see the word `iterator` used in two distinct ways. When we use the generic term, we mean any multivalued object (be it an `Array`, an implementation of the `java.util.Enumeration` interface or an implementation of the `java.util.Iterator` interface). When we mention `Iterator` we are speaking strictly about the Java interface.

10.1 Iterating with tags 101

Developing custom JSP tags that iterate over some set of values requires us to work, once again, with the familiar `BodyTag` interface. The `BodyTag` interface provides a method call protocol to control the execution of the tag's body—which we'll need in order to repeat the tag's body for every value in the `JavaBean`'s indexed property.

NOTE In JSP1.2 a new `IterationTag` interface was defined and we can also create tags using this interface. You can find information regarding the `IterationTag` later in this chapter.

Figure 10.1 shows how a tag can implement iteration using the `BodyTag` method call protocol.

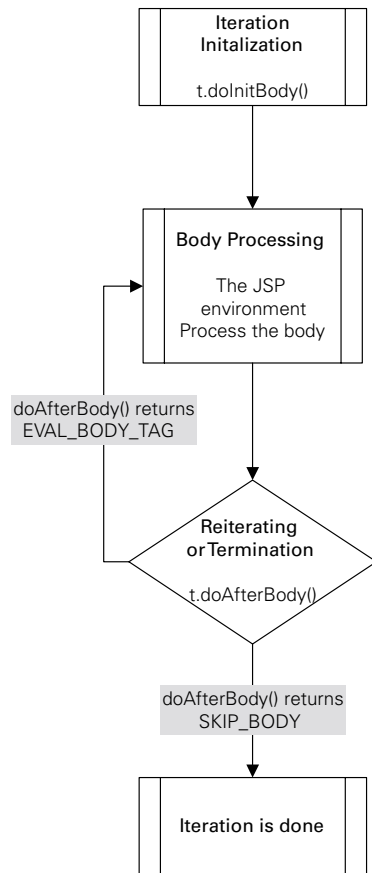


Figure 10.1
Implementing iteration using
the `BodyTag` interface

To further illustrate how iteration is accomplished with tags, we've translated the flow chart in figure 10.1 into (roughly) its Java equivalent.

```
t.doInitBody();
do {
    // The JSP runtime execute
    // the tag's body ...
} while(t.doAfterBody() == BodyTag.EVAL_BODY_TAG);
```

As figure 10.1 and the code fragment show, two methods (table 10.1) take part in implementing the iteration:

Table 10.1 Iteration methods

JSP method	Description
<code>doBodyInit()</code>	Used to initialize preiteration JSP scripting variables and the tags' internal values. For example, if your tag exposes some iterator object as a JSP scripting variable, it will probably use <code>doBodyInit()</code> to export its initial value.
<code>doAfterBody()</code>	Controls the iteration with its return codes: To continue the iteration, <code>doAfterBody()</code> returns a value of <code>BodyTag.EVAL_BODY_TAG</code> (or <code>IterationTag.EVAL_BODY_AGAIN</code> in JSP1.2). To break the iteration, it returns a value <code>BodyTag.SKIP_BODY</code> . This method is also where we re-export the iterator value (the current value of the property on which we are iterating), and where we write the result of the current iteration into the response.

NOTE You can skip the implementation of `doBodyInit()` and perform its work in `doStartTag()`. This will not have any effect on performance and may even simplify your tags. Better yet, since `doStartTag()` is not available in `IterationTag`, code that does not use it will be easier to port to this new tag. In any case, it is a good idea to separate the iteration handling from `doStartTag()` so that `doStartTag()` will only deal with service initialization (e.g., obtaining the object set that we are going to iterate) and `doBodyInit()` will deal with the initialization of the loop.

10.1.1 Iteration example: *SimpleForeachTag*

Now that you know how to implement iteration in your tags, we will take a look at a sample iterative tag and the code that performs iteration.

Our first iteration tag, `SimpleForeachTag`, will take a tag attribute that specifies a list of strings, walk over the string list, and, one by one, export an iterator object that contains the current string value for that iteration round. The following JSP fragment shows a sample usage of this tag:

```
<iter:foreach id="item"
  elements="1,2,3,4">
  The selected item is <%= item %> <br>
</iter:foreach>
```

Executing the above JSP fragment generates the following content:

```
The selected item is 1 <br>
The selected item is 2 <br>
The selected item is 3 <br>
The selected item is 4 <br>
```

Let's look at the code for the `SimpleForeachTag`'s handler (listing 10.1).

Listing 10.1 Source code for the `SimpleForeachTag` handler class

```
package book.iteration;

import java.util.StringTokenizer;
import java.util.LinkedList;
import java.util.List;
import java.util.Iterator;
import book.util.LocalStrings;
import book.util.ExBodyTagSupport;
import javax.servlet.jsp.JspException;

public class SimpleForeachTag extends ExBodyTagSupport {

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

    Iterator elementsList = null;

    protected String elements = null;

    public void setElements(String elements)
    {
        this.elements = elements;
    }

    public int doStartTag()
        throws JspException
    {
        parseElements(); ❶

        if(elementsList.hasNext()) {
            return EVAL_BODY_TAG;
        }
        return SKIP_BODY; ❷
    }

    public void doInitBody()
        throws JspException
    {
        pageContext.setAttribute(id, elementsList.next()); ❸
    }
}
```

```

    }

    protected void parseElements()
        throws JspException
    {
        List l = new LinkedList();
        StringTokenizer st = new StringTokenizer(elements, ",");
        while(st.hasMoreTokens()) {
            l.add(st.nextToken());
        }

        elementsList = l.iterator();
    }

    public int doAfterBody()
        throws JspException
    {
        try {
            getBodyContent().writeOut(getPreviousOut());
            getBodyContent().clear();
        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            log(ls.getStr(Constants.IO_ERROR), ioe);
            throw new
                JspTagException(ls.getStr(Constants.IO_ERROR));
        }
        if(elementsList.hasNext()) {
            pageContext.setAttribute(id, elementsList.next());
            return EVAL_BODY_TAG;
        }

        return SKIP_BODY;
    }

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

    protected void clearServiceState()
    {
        elementsList = null;
    }
}

```

-
- ❶ Parses the list of strings into a Java list and creates an enumerator.
 - ❷ If we have an element in the list, continues the body evaluation; otherwise skips the body (empty iteration).
 - ❸ Sets the iterator variable with the first element in the list.

- ④ **Breaks the string list into a Java list.**
- ⑤ **Writes the results of this iteration back to the user and clears the body buffer.**
- ⑥ **If we have more elements in the list, exports a new iterator value and repeats evaluating the body.**

The work in `SimpleForeachTag` takes place in three designated locations:

- The service phase initialization in `doStartTag()`. The tag initializes the set of objects on which we plan to iterate, and determines if we need to process the body. This is not necessary if the list of objects is empty.
- The loop initialization in `doInitBody()`. The tag exports the needed iterator object by calling `pageContext.setAttribute()` with the name of the object and the object itself. In doing so, we publish the iterator as a scripting variable, so that it ends up in the scope in the JSP (a practice we first came across with JavaBean tags in chapter 8). By exporting the iterator object, other tags and scriptlets can take advantage of it.
- The loop termination/repeating in `doAfterBody()`. The tag writes the results of the last loop into the previous writer (usually the writer that goes to the user) and then clears the body content to prepare it for the next iteration. In the final step, if there are additional items to iterate, the tag exposes a new iterator value and signals the JSP environment to repeat the execution by returning `EVAL_BODY_TAG`.

NOTE When implementing iterations using tags, you do not have to write the results of each loop separately. You may instead wait for the body execution to finish (no more elements on which to iterate) and then write the complete result. Doing so usually results in improved performance, but it may also cause a delay in the user's receipt of the results. For example, consider reading a substantial amount of data from a database and presenting it to the user with some iteration on the result set. Since we are working with a database, completing the iteration may take a while and writing the response only on completion may cause the user to leave the page. Writing the result of each loop incrementally would (depending on buffer size) cause the results to return to the user incrementally, instead of in a large chunk.

SimpleForeachTag's TagExtraInfo

Following the development of `SimpleForeachTag` we must now create its `TagExtraInfo` counterpart. You may recall from our discussions of the `TagExtraInfo` class in chapters 6 and 8, we need to create a subclass of `TagExtraInfo` whenever

we have a tag that exports a scripting variable. Since `SimpleForeachTag` will need to export the values of the iterator, we'll create a `TagExtraInfo` class for it that will inform the runtime of this. We'll call this class `ForeachTagExtraInfo`. Its implementation is in listing 10.2 wherein you see that it merely notifies the JSP runtime that a new scripting variable of type `String` is exported.

Listing 10.2 Source code for the `ForeachTagExtraInfo` class

```
package book.iteration;

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

public class ForeachTagExtraInfo extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data)
    {
        VariableInfo[] rc = new VariableInfo[1];
        rc[0] = new VariableInfo(data.getId(),
                                "java.lang.String",
                                true,
                                VariableInfo.NESTED);

        return rc;
    }
}
```

NOTE Note that the scope defined for the scripting variable is `NESTED`, meaning the variable exists and is accessible only within the body of the tag that exported it. This is important since the variable we export is our iterator, and so should exist only within the body of the loop.

SimpleForeachTag in action

Having written `SimpleForeachTag` and its `TagExtraInfo` we can now write JSP code to work with it. Since this is only the beginning of our iteration tags discussion, we will take that same JSP fragment and make it the content of our JSP as seen in listing 10.3.

Listing 10.3 JSP driver for `SimpleForeachTag`

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

<html>
<body>

<iter:foreach id="item"
```

```
elements="1,2,3,4">
The selected item is <%= item %> <br>
</iter:foreach>

</body>
</html>
```

Now when we execute our JSP, `SimpleForeachTag` will repeat its body four times (one for each string in “elements”); first with 1 as the value of the item (our iterator), and lastly with 4 as its value.

10.2 Generalized iterating tags

In perusing the implementation of `SimpleForeachTag` it appears that most of the work done by the tag is not unique to it. In fact, other than the creation of the `Iterator` object in `parseElements()` all the other code was generic. True, some tags will not want to expose an iterator, and others may want to expose more than a single iterator as a scripting variable (for some other tag-specific purpose), but these tags are not representative of the majority. In most cases, tags will differ only in the objects they iterate (some will iterate over an `Enumeration`, others on `Array`, etc.) but the general structure will stay the same; a single iterator scripting variable will be exposed and updated for each element.

Based on this general iterating structure, we’ll build:

- A generic iteration interface that lets the tag developer specify how to iterate over some set of objects.
- A basic iterator tag that takes a generic iteration object (`Enumeration`, `Array`, etc.) and iterates on it.

Creating these two, generic components will then streamline the creation of various iteration tags. These specialized iteration tags will be custom-built, based on the type of Java object to be contained in the iterator, and the iterator type in which these objects are to be contained. For example, our `SimpleForeachTag` had an iterator type of `java.util.Iterator`, and contained in that iterator was a list of `Strings`. We are now going to build these two components (the class and interface) and modify `SimpleForeachTag` to use this new, more generic infrastructure.

10.2.1 A generic iteration interface

Before looking into the new `ForeachTag`, let’s study the generic iteration infrastructure on which it is constructed, starting with the generic iteration interface as seen in listing 10.4.

Listing 10.4 Source code for the generic iteration interface

```
package book.iteration;

import javax.servlet.jsp.JspException;

public interface IterationSupport {

    public boolean hasNext()
        throws JspException;

    public Object getNext()
        throws JspException;

}
```

Why do we need another iteration/enumeration interface, as Java already offers plenty. You may also wonder, why a `JspException` is thrown from the methods `hasNext()` and `getNext()`. Shouldn't a generic interface remove JSP related ties? We do this because we want to provide better JSP integration. Let's explore our motivation for this integration.

NOTE We could consider the option of defining a new exception type (such as `IterationException`) that the iteration support methods could throw; but why should we? This code is written for the JSP tags, and we are not going to reuse it. In 99 percent of all cases, you are going to throw a `JspException` as a result of the error. Based on this argument, we've rejected the new exception type idea, and continue to use `JspException` as our error-reporting vehicle.

10.2.2 *IterationTagSupport*

Let's look at the basic iteration tag class, `IterationTagSupport`, and how it uses `IterationSupport`. Before taking a look into the implementation of `IterationTagSupport` as presented in listing 10.5, let's consider how we would like it to work.

What should *IterationTagSupport* do?

Most emphatically, the generic iteration tag class should automatically take care of iteration-related issues such as flow control, as well as exporting default iterator variables. In addition, it must be able to:

- Create an `IterationSupport` object out of the elements provided as a tag attribute. This can be accomplished by defining a method that our specialized iteration tags can override and that `IterationTagSupport` will call during its `doStartTag()`. By specialized tag we mean the special version of the tag that

is custom built to handle a particular iterator type and a particular type of object in that iterator.

- Export a different set of JSP variables. Whenever `IterationTagSupport` wants to export its iterator value, it should call yet another method that can be overridden by the specialized tag (but the default implementation of the variable exportation method should export only a single iterator).

IterationTagSupport's implementation

`IterationTagSupport` was created with a few methods that may be overridden by specialized iteration tags.

Listing 10.5 Source code for the generic iteration tag handler

```
package book.iteration;

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

public abstract class IterationTagSupport
    extends ExBodyTagSupport {

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

    IterationSupport elementsList = null;

    public int doStartTag()
        throws JspException
    {
        fetchIterationSupport();
        if(elementsList.hasNext()) {
            return EVAL_BODY_TAG;
        }
        return SKIP_BODY;
    }

    public void doInitBody()
        throws JspException
    {
        exportVariables();
    }

    public int doAfterBody()
        throws JspException
    {
        try {
            getBodyContent().writeOut(getPreviousOut());
            getBodyContent().clear();
        }
    }
}
```

```

        } catch(java.io.IOException ioe) {
            // User probably disconnected ...
            // Log and throw a JspTagException
        }

        if(elementsList.hasNext()) {
            exportVariables();
            return EVAL_BODY_TAG;
        }

        return SKIP_BODY;
    }

    protected abstract void fetchIterationSupport() ❶
        throws JspException;

    protected void exportVariables() ❷
        throws JspException
    {
        pageContext.setAttribute(id, elementsList.getNext());
    }

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

    protected void clearServiceState() ❹
    {
        elementsList = null;
    }
}

```

-
- ❶ **First override point. The specialized tag must implement this method to create and set an `IterationSupport` object** The first method that tags can and must override is `fetchIterationSupport()`. This abstract method is the location wherein the overriding tag should implement the creating and setting of the `IterationSupport` object and any specialized iteration tag must provide such objects to make the generic infrastructure work. If problems arise within `fetchIterationSupport()`, it can throw a `JspException` that the generic implementation will pass to the JSP runtime.
- ❷ **Second override point. The specialized tag may want to export additional objects** The second method that can be overridden is `exportVariables()`, which is where the generic iteration tag exports the iterator (based in the `id` attribute). An overriding tag may override this method to add more variables. For example, a certain tag iterates a hash table and wants to export both the key to the table and the value itself. In this case you would like to add the exportation of the value variable along with the default iterator.

- ③ **Override if you have additional attributes in the specialized tag (you probably do).**
- ④ **Override if you have additional service state in the specialized tag.**

Listing 10.5 shows that the general structure of `IterationTagSupport` is very similar to the one presented in `SimpleForeachTag`. The tag is merely a generic iteration infrastructure with several methods to override as explained in the annotations. Note also that `IterationTagSupport` extends our now familiar `ExBodyTagSupport`, and therefore inherits its functionality.

An improved `ForeachTag` which uses `IterationTagSupport`

We've mentioned several times the concept of a specialized tag, by which we infer a tag that uses our generic interface and class for a specific iterator and object type. Let's now look at one such specialized tag, `ForeachTag`, which uses `IterationTagSupport` to support an `Iterator` containing a list of `Strings` (see listing 10.6).

Listing 10.6 Source code for the `ForeachTag` handler class

```
package book.iteration;

import java.util.StringTokenizer;
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;
import book.util.LocalStrings;
import book.util.ExBodyTagSupport;
import javax.servlet.jsp.JspException;

public class ForeachTag extends IterationTagSupport {

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

    protected String elements = null;

    public void setElements(String elements)
    {
        this.elements = elements;
    }

    protected void fetchIterationSupport()
        throws JspException
    {
        List l = new LinkedList();
        StringTokenizer st = new StringTokenizer(elements, ",");
        while(st.hasMoreTokens()) {
            l.add(st.nextToken());
        }
        elementsList = new IteratorIterationSupport(l.iterator());
    }
}
```

①

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

class IteratorIterationSupport implements IterationSupport { ❸
    Iterator i = null;

    IteratorIterationSupport(Iterator i)
    {
        this.i = i;
    }

    public boolean hasNext()
        throws JspException
    {
        return i.hasNext();
    }

    public Object getNext()
        throws JspException
    {
        return i.next();
    }
}
```

- ❶ Parsing the list specification string and making an `IterationSupport` out of it.
- ❷ Clearing the additional tag property.
- ❸ Implementing an `IterationSupport` class that uses a Java `Iterator` object.

The new `ForeachTag` has most of its code implementing its tag-specific functionality, that is, dealing with an `Iterator` of `Strings`. Also of note in our implementation is the additional `IteratorIterationSupport` class we created, which is simply an implementation of the generic `IterationSupport` that works on the `java.util.Iterator` interface. We can imagine a similar class that works on `Arrays` and even another for `Enumerations` (or perhaps one that handles all?). The `IteratorIterationSupport` class is not, of course, unique to `ForeachTag` and we will be able to reuse it many times in other specialized tags.

We now have a way to easily create iteration tags that iterate on all sorts of objects. We'll flex the power of this infrastructure in the next section in creating a tag that is capable of iterating on just about anything.

10.3 IterateTag

The previous section presented a generic iteration tag infrastructure that we will now use to develop a new iteration tag (named `IterateTag`) which will be able to iterate over the following types of objects:

- Arrays of all types
- Enumerations—objects of type `java.util.Enumeration`
- Iterators—objects of type `java.util.Iterator`.

We're going to put this functionality into a single tag so its users will be able to use one tag for all their iteration chores. They will be able to reference the object they want to iterate in the same way as in chapter 8, using Java reflection. In fact, we'll reuse the reflection code we saw in chapter 8's `ReflectionTag` to accomplish this. In doing so, our tag will be able to take any bean property value and iterate its objects. For example, we will be able to take a shopping cart with a method such as:

```
public Enumeration getProducts();
```

and iterate on the `Enumeration` value returned from it.

10.3.1 Design considerations for IterateTag

Given that we have the generic iteration infrastructure, and that we have a previously built basic reflection tag, implementing our tag should be a breeze (almost codeless, you might expect). But this is not quite the case because a Java class cannot inherit two superclasses (no multiple inheritance, if you recall). Also, our `ReflectionTag` did not implement `BodyTag`; instead, it implemented the `Tag` interface, so it cannot serve as a base class for an iteration-related tag. As a result, our iteration tag will have to reimplement the reflection code that we previously developed. There are ways to share the implementation code between the tags, but for simplicity's sake, we will merely copy and paste the needed code.

10.3.2 Wrapping iterators

We will use the `ReflectionTag` code from chapter 8 to procure the referenced object from within the iteration tag, but we still need to decide what to do with it; meaning, how are we going to wrap it within an `IterationSupport`? We choose to create an `IterationSupport` implementation for each of the different iterator types (`Iterator`, `Enumeration`, and `Array`), then wrap the object within the matching `IterationSupport` implementation. An `IterationSupport` wrapper for the `Iterator` interface was covered in the previous section, so let's now look at the individual wrappers for `Array` and `Enumeration`.

ArrayIterationSupport

The first `IterationSupport` wrapper class we implement will be for Arrays. Implementing `IterationSupport` is not usually too much of a challenge, yet this case is different due to the requirement to be iterable on any type of Array (i.e., an Array of Strings, an Array of Dates, etc.). Normally, when the array element type is known, indexing the array elements is a snap, but how do you do that when the element type is unknown?

The answer, as you might have guessed, is reflection. The reflection package contains an `Array` class with static methods for manipulating array elements and querying the array's length. We make use of this reflection class in our implementation of `ArrayIterationSupport`, as seen in listing 10.7.

Listing 10.7 Source code for the `ArrayIterationSupport` utility class

```
package book.iteration;

import java.lang.reflect.Array;
import javax.servlet.jsp.JspException;

class ArrayIterationSupport implements IterationSupport {

    protected Object a = null;
    protected int    pos = 0;

    ArrayIterationSupport(Object a)
    {
        this.a = a;
        this.pos = 0;
    }

    public boolean hasNext()
        throws JspException
    {
        return (pos < Array.getLength(a));    ❶
    }

    public Object getNext()
        throws JspException
    {
        if(hasNext()) {
            Object rc = null;
            rc = Array.get(a, pos);    ❷
            pos++;
            return rc;
        }
        // Throw an exception
    }
}
```

- ❶ Using Array's static method to find the length of the input array.
- ❷ Using Array's static method to get an indexed value.

The functionality rendered by the Array class is enough for us to be able to have full access to all the array's attributes and elements.

EnumerationIterationSupport

The IterationSupport class supporting Enumerations, EnumerationIterationSupport, is very straightforward, since both the IterationSupport and Enumeration interfaces are so similar (see listing 10.8)

Listing 10.8 EnumerationIterationSupport

```
package book.iteration;
import java.util.*;

public class EnumerationIterationSupport implements IterationSupport
{
    Enumeration elements;

    public EnumerationIterationSupport(Enumeration e)
    {
        elements = e;
    }

    public boolean hasNext()
        throws JspException
    {
        return elements.hasMoreElements();    ❶
    }

    public Object getNext()
        throws JspException
    {
        return elements.nextElement();    ❷
    }
}
```

- ❶ Using Enumeration's method to determine if more elements exist.
- ❷ Using Enumeration's method to retrieve the current object.

10.3.3 Implementing IterateTag

The next step is the implementation of IterateTag (listing 10.9) in which we'll see how all the wrappers, reflection logic, and our generic iteration framework combine in its creation (note that for clarity reasons we snipped the reflection code out of the code listing).

Listing 10.9 Source code for the IterateTag handler class

```
package book.iteration;

import java.beans.IntrospectionException;
import java.lang.reflect.InvocationTargetException;
import java.util.Enumeration;
import java.util.Iterator;
import book.reflection.ReflectionTag;
import book.util.LocalStrings;
import book.util.BeanUtil;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;

public class IterateTag extends IterationTagSupport {

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

    // Reflection related properties and properties setters
    // were removed from this section.

    protected void fetchIterationSupport()
        throws JspException
    {
        Object o = getPointed(); ❶

        if(o instanceof Iterator) {
            elementsList =
                new IteratorIterationSupport((Iterator)o);
        } else if(o instanceof Enumeration) {
            elementsList =
                new EnumerationIterationSupport((Enumeration)o);
        } else if(o.getClass().isArray()) {
            elementsList = new ArrayIterationSupport(o);
        } else {
            // Throw an exception to inform that we cannot
            // iterate this object
        }
    }

    // The reflection code below this line
    // was removed from this listing
}
```

- ❶ `getPointed()` retrieves the object the tag should iterate on. This method is inherited from `ExBodyTagSupport`.
- ❷ Gets the referenced object and wraps it within the appropriate `IterationSupport` implementation.

Apart from the reflection related code which we've omitted (we've seen how this code works in chapter 8), `IterateTag`'s implementation consists of a single method implementation: `fetchIterationSupport()`. This method merely checks the object that is passed as the tag attribute and selects an appropriate iterator and `IterationSupport` wrapper, based on the object's type.

IterateTagExtraInfo

Accompanying the `IterateTag` is the `IterateTagExtraInfo` whose implementation is fairly effortless. Once again, we need to create this `TagExtraInfo` object for our `IterateTag` because we will be exporting a scripting variable from it. From an attribute and variable exportation point of view, `IterateTag` and `ExportTag` (as presented in chapter 8) are quite similar. The only difference is that our current variable is exported as a `NESTED` variable, meaning its scope only exists within the tag's body. Because they are so similar, all we need to do is inherit `ExportTagExtraInfo` (again, from chapter 8) and modify the `VariableInfo` it returns to reflect a `NESTED` variable. As listing 10.10 shows, this is exactly what we did.

Listing 10.10 Source code for the `IterateTagExtraInfo` class

```
package book.iteration;

import book.reflection.ExportTagExtraInfo;
import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;
import javax.servlet.jsp.tagext.VariableInfo;

public class IterateTagExtraInfo extends ExportTagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data)
    {
        VariableInfo[] rc = new VariableInfo[1];
        rc[0] = new VariableInfo(data.getId(),
                                guessVariableType(data),
                                true,
                                VariableInfo.NESTED); ❶
        return rc;
    }
}
```

- ❶ Returns a `NESTED` variable.

IterateTag's TLD

The last step in our implementation of `IterateTag` is its tag library descriptor entry as seen in listing 10.11.

Listing 10.11 Tag library descriptor entry for IterateTag

```
<tag>
  <name>iterate</name>
  <tagclass>book.iteration.IterateTag</tagclass>
  <teiclass>book.iteration.IterateTagExtraInfo</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
    Iterate over an Object. The object can be an array,
    Iterator or Enumeration.
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
    <rtextprvalue>>false</rtextprvalue>
  </attribute>
  <attribute>
    <name>type</name>
    <required>>false</required>
    <rtextprvalue>>false</rtextprvalue>
  </attribute>
  <attribute>
    <name>object</name>
    <required>>false</required>
    <rtextprvalue>>true</rtextprvalue>
  </attribute>
  <attribute>
    <name>name</name>
    <required>>false</required>
    <rtextprvalue>>false</rtextprvalue>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>>false</required>
    <rtextprvalue>>false</rtextprvalue>
  </attribute>
  <attribute>
    <name>index</name>
    <required>>false</required>
    <rtextprvalue>>true</rtextprvalue>
  </attribute>
  <attribute>
    <name>property</name>
    <required>>false</required>
    <rtextprvalue>>false</rtextprvalue>
  </attribute>
</tag>
```

The tag library entry is almost identical to the one we had for `ExportTag`. The only significant difference is that `ExportTag` had an empty body, whereas `IterateTag` has, of course, a JSP body.

10.4 Look, Mom! No scriptlets—IterateTag in action

Armed with `IterateTag` we can now greatly improve our JSP development and even reach the point at which scriptlets are no longer needed. To illustrate, we present a real world example wherein a JSP file shows a user the content of his or her shopping cart. For this example, the shopping cart is kept inside a session variable that the JSP file retrieves to create a table containing the current products in the cart.

The methods provided by the shopping cart and the cart items are available in listing 10.12.

Listing 10.12 The methods exposed by the cart and cart elements

```
public class Cart implements Serializable {
    public int getDollars();
    public int getCents();
    public boolean isEmpty();
    public Enumeration getProducts();
    public Enumeration getProductNames();
    public CartElement getProduct(String key);
    public CartElement []getProductValues();
    public void addProduct(String key, CartElement ince);
    public void removeProduct(String key);
}

public class CartElementImp implements CartElement {
    public int getDollars();
    public void setDollars(int dollars);
    public int getCents();
    public void setCents(int cents);
    public int getQuantity();
    public void setQuantity(int quantity);
    public void setName(String name);
    public String getName();
}
```

10.4.1 Printing the shopping cart with scriptlets

Assuming we have the cart in the session state and we want to display the cart's content in some tabular format (figure 10.2), we *could* create a scriptlet-littered JSP file, such as the one seen in listing 10.13.

Listing 10.13 JSP file that uses scriptlets to present the cart information

```

<%@ page errorPage="error.jsp" %>
<%@ page import="book.util.*,java.util.*" %> ❶

<html>
<body>

<%
    Cart cart = (Cart)session.getAttribute("cart"); ❷
    if(!cart.isEmpty()) {
%>
Your cart contains the following products:

<table>
<tr><th>Product</th> <th>Quantity</th> <th>Price</th> </tr>
<% java.util.Enumeration e = cart.getProducts();
    while(e.hasMoreElements()) {
        CartElementImp p = (CartElementImp)e.nextElement();
%>
    <tr>
        <td> <%= p.getName() %></td>
        <td> <%= p.getQuantity() %> </td>
        <td> <%= p.getDollars() %>.<%= p.getCents() %>$ </td>
    </tr>
<% } %>
    <tr>
        <td> Totals <td>
        <td> <%= cart.getDollars() %>.<%= cart.getCents() %>$<td> ❸
    </tr>
</table>

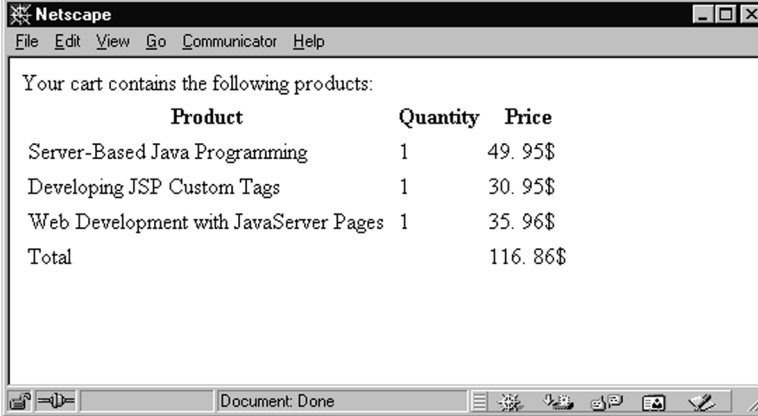
<% } else { %>
Your cart is empty.
<% } %>

</body>
</html>

```

- ❶ **Importing classes to be used in the scriptlets.**
- ❷ **Gets a reference to the cart.**
- ❸ **Enumerates the products and presents their properties.**
- ❹ **Presents the total price (property of the cart).**

Listing 10.13 serves as a basic example for a piece of JSP code that, once introduced to the scriptlets, is no longer manageable by anyone but a Java programmer. The file is replete with the familiar Java curly brackets, Java flow control statements, and casting and import statements—all of which are difficult for a non-Java programmer



Your cart contains the following products:

Product	Quantity	Price
Server-Based Java Programming	1	49.95\$
Developing JSP Custom Tags	1	30.95\$
Web Development with JavaServer Pages	1	35.96\$
Total		116.86\$

Figure 10.2 Cart presentation output

to grasp. Instead of this chaos, we can use the `IterateTag` we just developed to substantially improve the JSP.

10.4.2 Printing the shopping cart with `IterateTag`

All of the scriptlets in listing 10.13 can be eliminated by making use of our new `IterateTag` as in listing 10.14. Executing the JSP code on a sample cart content yielded the response presented in figure 10.2.

Listing 10.14 JSP file that uses custom tags to present the cart information

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

<html>
<body>

<cond:with name="cart" property="empty">
<cond:test condition="eq true">
    Your cart is empty.
</cond:test>
<cond:test condition="eq false">
```

Your cart contains the following products:

```

<table>
  <tr><th>Product</th> <th>Quantity</th> <th>Price</th> </tr>
  <iter:iterate name="cart" property="products" id="product"> ❷
    <tr>
      <td><bean:show name="product" property="name"/> </td>
      <td><bean:show name="product" property="quantity"/></td> ❸
      <td><bean:show name="product" property="dollars"/>.
        <bean:show name="product" property="cents"/>${</td>
    </tr>
  </iter:iterate>
  <tr>
    <td>Totals<td>
    <td><bean:show name="cart" property="dollars"/>. ❹
      <bean:show name="cart" property="cents"/>${<td>
  </tr>
</table>

</cond:test>
</cond:with>
</body>
</html>

```

- ❶ References all the TLDs we use.
- ❷ Enumerates the products (using the enumeration property).
- ❸ Presents the product's properties.
- ❹ Presents the total price (property of the cart).

Comparing listings 10.13 and 10.14 shows the advantages of using custom tags. Listing 10.14 is much simpler: all the curly brackets, type casting, and the like are gone, and it is readable by almost anyone. Moreover, all tag supporting tools can manipulate the file and we feel certain that they will be able to get along with our custom tags. Listing 10.13 is littered with endless scriptlets to the point that developing the page without a programmer's help is very difficult. Which page would you prefer to have your HTML coder maintain?

10.5 Making it easier on the JSP author

As convenient as the JSP might be in listing 10.14, there is still something that bothers us from a usability standpoint; namely, the printing of the value of a bean property to the user is too cumbersome. To illustrate, look at the following JSP fragment:

```
<iter:iterate name="cart" property="products" id="product">
```

```

<tr>
  <td><bean:show name="product" property="name"/> </td>
  <td><bean:show name="product" property="quantity"/></td>
  <td><bean:show name="product" property="dollars"/>.
    <bean:show name="product" property="cents"/>${</td>
</tr>
</iter:iterate>

```

Seeing all those `<bean:show>` tags begs the question: why do we need so much overhead associated with using the bean tag and pointing to the property in the product? We know that we are interested in the product object (since we're iterating on it) yet our `<bean:show>` tag forces us to pass it as a name attribute for every property we print to the user. Can't we make access to bean-based, nonindexed properties in an iterator less complicated (or friendlier)? We can, but how?

Improving access to nonindexed JavaBean properties

The first thought that comes to mind is to create a tag with a single attribute that points to the property name. When running, this tag will fetch the iterator object from the iteration tag and query its property value. The following JSP fragment shows a revised version of the previous JSP fragment that uses this simplified tag.

```

<iter:iterate name="cart" property="products" id="product">
<tr>
  <td><bean:showp property="name"/> </td>
  <td><bean:showp property="quantity"/></td>
  <td><bean:showp property="dollars"/>.
    <bean:showp property="cents"/>${</td>
</tr>
</iter:iterate>

```

This is an improvement; however, we still are not entirely satisfied with the new JSP fragment, largely because the number of keystrokes we've saved is not especially significant. To make the syntax for retrieving a property extremely terse, we don't want to use a tag at all; we want something that is even more minimal. Syntax such as the following is clearly an improvement for the JSP author, especially if they're building a number of JSPs with property access in iterators.

```

<iter:iterate name="cart" property="products" id="product">
  <tr>
    <td> <$ name $> </td>
    <td> <$ quantity $> </td>
    <td> <$ dollars $>.<$cents$>${</td>
  </tr>
</iter:iterate>

```

In this JSP fragment we no longer use tags to present the property values of the iterator. Instead, a property value in the current iterator is referenced by using a special

directive with field placement syntax `<${property-name}>`. Using this field placement could be a time-saver, but how would we implement it? Up to this point, everything we created was a tag; this new proprietary directive is not. The way to implement this functionality is to modify our iteration tags to perform a pass on their body content and translate these field placement directives into values that should replace them. By processing the body in this way, we can easily swap any special directive we want with some other value; in this case, the value of a JavaBean's nonindexed property.

10.5.1 Building a better tag

Remember that the iterator tags implement the `BodyTag` interface; hence, the iteration tags can have direct access to their body *before* they write it to the response stream. All the tag has to do is implement some body parsing in `doAfterBody()`, in which the tag will replace our field placement directives with the actual field values.

Implementing the substitution of field placement directives with their actual values should be done in a generic manner, for several reasons:

- It is not safe to assume that we will always want to use the field placement directives. For example, certain users may not want to use proprietary syntax. In such cases we do not want to take the performance penalty associated with parsing the body. Thus we require the ability to disable/enable substitutions on the fly.
- We can imagine many different objects on which we may iterate, as well as many field types that we may want to show, from JavaBean properties to database columns. We want to build a generic solution such that we do not implement the body parsing differently for each case.
- We may develop many different iteration tags and most of them will need the (extremely nifty) field substitution feature, and we do not want to implement the related substitution logic more than once.

10.5.2 The design

To attain these goals, we distribute the implementation of the field substitution into the following units:

- Body parsing—This part of our solution searches for field references and identifies them. We'll implement this functionality in `IterationTagSupport`, our iteration tag superclass. This will make all tags derived from `IterationTagSupport` capable of performing field substitution.
- Field fetching—This is the part of our solution that retrieves a field's value when one is found. Whenever `IterationTagSupport` parses and identifies a

field reference, it will use an object that implements an interface we'll call `FieldGetter`. This interface will allow us to get the value of the referenced field from the current iterator. Since `FieldGetter` will be an interface, we can create many different implementations of it, such as one that fetches a database column value, or another that gets bean properties. This will become clearer when we see the code.

- Setting the `FieldGetter`—Combining the first two portions of our design, we see that any specialized implementation of `IterationTagSupport` will need a specialized version `FieldGetter`, corresponding to the type of objects the iterator contains. The specialized iteration tag will know the type of objects that it exposes as iterators and will therefore know what type of `FieldGetter` to use. If no `FieldGetter` is used, the tag will not implement any field substitution, hence avoiding the associated performance costs from parsing the body. This accomplishes our previously mentioned goal of making the field substitution optional for performance reasons.

This design should accomplish all our defined goals. Our abstract design will become much more comprehensible as we look at our implementation and an example.

10.5.3 `FieldGetter` and `ReflectionFieldGetter`

Let's start by looking at the `FieldGetter` interface, which provides one method to set the object whose fields we'll want to retrieve and a second method to get those fields from the object. We present this interface in listing 10.15, along with an implementation of it called `ReflectionFieldGetter` whose job is to implement a `FieldGetter` that gets JavaBeans properties (through reflection).

Listing 10.15 Source code of `FieldGetter` and `ReflectionFieldGetter`

```
package book.util;

import java.beans.IntrospectionException;
import java.lang.reflect.InvocationTargetException;

public interface FieldGetter {

    public void setObject(Object o) ❶
        throws IllegalArgumentException;

    public Object getField(String fieldName) ❷
        throws IllegalAccessException;
}

public class ReflectionFieldGetter implements FieldGetter {

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

```

protected Object o;

public void setObject(Object o) ❸
    throws IllegalArgumentException
{
    this.o = o;
}

public Object getField(String fieldName) ❹
    throws IllegalAccessException
{
    try {
        return BeanUtil.getObjectPropertyValue(o,
                                                fieldName,
                                                null); ❺
    } catch(InvocationTargetException ex) {
    } catch(IllegalAccessException ex) {
    } catch(IntrospectionException ex) {
    } catch(NoSuchMethodException ex) {
    }

    // Throw an exception
}
}

```

- ❶ Generic method to set the object whose fields we'll later retrieve.
- ❷ Generic method to get an object's field by name.
- ❸ For `ReflectionFieldGetter`, `setObject` will be set with a `JavaBean`.
- ❹ For `ReflectionFieldGetter`, `getField` uses reflection (seen in chapter 8) to get a field from the `JavaBean`.

`FieldGetter` has two methods: `setObject()` that tells the getter which object we are going to query for a field and `getField()` to procure the field's value. When using a `FieldGetter`, instantiate it, then set an object into the `FieldGetter` using `setObject()`, and then call `getField()` to get the values of the wanted fields. For error notification, `FieldGetter`'s methods can throw exceptions (e.g., if the object set into the `FieldGetter` implementation is not of the right type, say a `ResultSet` for a database-aware `FieldGetter`). To further clarify `FieldGetter`, listing 10.15 also shows the implementation of `ReflectionFieldGetter` which implements the `FieldGetter` functionality for `JavaBeans` by using the reflection API. Remembering the types of objects `IterateTag` enumerates, it is reasonable to assume that it is going to step over beans in its iterations.

10.5.4 Integrating FieldGetter with IterationTagSupport

Having established the nature of the `FieldGetter`, how do we integrate it into the iteration process? The answer is in the updated implementation of `IterationTagSupport` wherein `FieldGetter` was integrated. An updated listing of `IterationTagSupport` is in listing 10.16 (for clarity, unmodified code was omitted and whenever new and old code are mixed, the new code is in bold).

Listing 10.16 An updated `IterationTagSupport` with `FieldGetter` integration

```
package book.iteration;

import java.io.Reader;
import java.io.IOException;
import book.util.LocalStrings;
import book.util.FieldGetter;
import book.util.ExBodyTagSupport;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;

public abstract class IterationTagSupport
    extends ExBodyTagSupport {

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

    protected IterationSupport elementsList = null;
    protected Object current;
    protected FieldGetter fGetter = null;
    // Some unmodified code was removed

    public int doAfterBody()
        throws JspException
    {
        try {
            if (null == fGetter) {
                getBodyContent().writeOut(getPreviousOut());
            } else {
                populateFields();
            }
            getBodyContent().clear();
        } catch (java.io.IOException ioe) {
            // User probably disconnected ...
            // Log and throw a JspTagException
        }

        if (elementsList.hasNext()) {
            exportVariables();
            return EVAL_BODY_TAG;
        }

        return SKIP_BODY;
    }
}
```

```

    }

    protected void populateFields()
        throws JspException
    {
        String field = null;
        try {
            Reader r = getBodyContent().getReader();
            JspWriter w = getPreviousOut();

            fGetter.setObject(current); ③

            int ch = r.read();
            while(-1 != ch) { ④
                if('<' == ch) {
                    ch = r.read();
                    if('$' == ch) {
                        /* found a field reference */
                        field = readFieldName(r);
                        w.print(fGetter.getField(field)); ⑤
                        ch = r.read();
                    } else {
                        w.write('<');
                    }
                } else {
                    w.write(ch);
                    ch = r.read();
                }
            }
        } catch(IllegalAccessException e) {
            // Throw a JspTagException
        } catch(IOException ioe) {
            // Throw a JspTagException
        }
    }

    protected String readFieldName(Reader r)
        throws JspException, IOException
    {
        StringBuffer sb = new StringBuffer();
        int ch = r.read();
        while(-1 != ch) {
            if('$' == ch) { ⑥
                ch = r.read();
                if('>' == ch) {
                    /* found a field ending mark */
                    return sb.toString().trim();
                } else {
                    sb.append((char)ch);
                }
            } else {
                sb.append((char)ch);
                ch = r.read();
            }
        }
    }

```

```

    }
  }
  // Throw a JspTagException (parse error, directive
  // was not terminated)
}

// Some unmodified code was removed
protected void exportVariables()
    throws JspException
{
    current = elementsList.getNext(); ⑦
    pageContext.setAttribute(id, current);
}

// Some unmodified code was removed

protected void clearServiceState()
{
    elementsList = null;
    current = null;
    fGetter = null;
}
}

```

-
- ① ⑦ **Two new instance variables for the field substitution** The majority of new code that was added has to do with parsing the body and propagating the current iterator value to the field substitution code. Propagating the value of the current iterator is needed because `doAfterBody()` does not know the value. Implementing the propagation involves adding an instance variable to carry the iterator value as well as initialize this value whenever a new iterator value is exported.
 - ① ② **If a field getter is available, field substitution is on** Now that the iterator value is available for all methods, we can use `doAfterBody()` to process the body. Body processing is turned on whenever a value is set to the class `FieldGetter` member, `fGetter`, which informs `IterationTagSupport` that field substitution is required and `populateFields()` is being called.
 - ③ **Sets the current iterator into the field getter to make it possible to get field values from the iterator** ④ **Searches for a directive starting prefix (<\$)** ⑤ **Reads the field name and prints its value using the getter** ⑥ **Looks for the directive-terminating sequence (\$>)** `populateFields()` and `readFieldName()` are those that actually implement the field substitution. `populateFields()` parses through the body looking for the substitution directive-starting prefix. Whenever `populateFields()` finds this directive it will ask `readFieldName()` to read the rest of the directive (including its suffix) and return the name of the field referenced therein. Once `populateFields()` holds the referenced field name, it uses the `FieldGetter` to obtain the field's value, print it, and continue parsing the body (looking for other directives).

- 7 Stores the current iterator for later use in `doEndBody()`.

10.5.5 Updating `IterateTag` to perform field substitution

Now that the modifications to `IterationTagSupport` are complete, the road to field substitution is open. All we need is to modify `IterateTag` and make it set the `ReflectionFieldGetter` into `IterationTagSupport` in order to turn on field substitution. The modifications to `IterateTag` are presented in listing 10.17 (unmodified code was omitted and new code is in bold).

Listing 10.17 An updated `IterateTag` handler class with field substitution support

```
package book.iteration;
// Some unmodified code was removed
import book.util.LibraryConfig;
// Some unmodified code was removed
public class IterateTag extends IterationTagSupport {
    // Some unmodified code was removed
    protected void fetchIterationSupport()
        throws JspException
    {
        Object o = getPointed();
        if(o instanceof Iterator) {
            elementsList =
                new IteratorIterationSupport((Iterator)o);
        } else if(o instanceof Enumeration) {
            elementsList =
                new EnumerationIterationSupport((Enumeration)o);
        } else if(o.getClass().isArray()) {
            elementsList = new ArrayIterationSupport(o);
        } else {
            // Throw an exception to inform that we cannot
            // iterate this object
        }
        if(LibraryConfig.isFieldPlacementInUse()) {
            fGetter = new ReflectionFieldGetter();
        }
    }
    // Some unmodified code was removed
}
```

Only `fetchIterationSupport()` was modified to add the `ReflectionFieldGetter` into `IterationTagSupport` according to a property in the library configuration.

10.5.6 Field substitution in action

Once the tweaking of the iteration code is behind us, we can modify our original JSP (which printed the shopping cart) and adapt it to use field substitution. The end result of this adaptation is shown in listing 10.18 and, as you shall see, the loop that populates the HTML table with cart items has been simplified.

Listing 10.18 A JSP file that uses field substitution

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

<html>

<body>
<cond:with name="cart" property="empty">
<cond:test condition="eq true">
    Your cart is empty.
</cond:test>
<cond:test condition="eq false">
    Your cart contains the following products:

    <table>
        <tr><th>Product</th> <th>Quantity</th> <th>Price</th> </tr>
        <iter:iterate name="cart" property="products" id="product">
            <tr>
                <td> <$ name $> </td>
                <td> <$ quantity $> </td>
                <td> <$ dollars $>.<$cents$>$ </td>
            </tr>
        </iter:iterate>
            <tr>
                <td> Totals <td>
                <td> <bean:show name="cart" property="dollars"/>.
                    <bean:show name="cart" property="cents"/>$ <td>
            <tr>
        </table>
    </cond:test>
</cond:with>
</body>
</html>
```

This section showed more than a mere ease-of-use enhancement to the iteration task. It showed how to add your own proprietary additions to the JSP syntax. Some developers may reject the idea of working with proprietary JSP additions, since this syntax will not be useful in other settings. However, the additions presented in this chapter are based on custom tags, and since custom tags are a standard JSP feature, the field replacement features developed here will run on all JSP engines. Although our creation's nature is indeed proprietary, our tags and their additions can run anywhere. The simplicity of our field substitution syntax and the time it will save JSP authors who use it are well worth the expense of a bit of proprietary syntax.

10.6 JSP1.2 and *IterationTag*

This chapter created iteration tags using `BodyTag`, but using `BodyTag` for iteration includes within it a hidden performance hazard due to its buffering overhead.

As noted in chapter 6, when using `BodyTag` the JSP runtime places the body into an intermediate buffer (the `BodyContent` object) and leaves it up to the tag to actually do something with the results of the body execution. In our iteration tags, what we did with these results was to copy them into the response flowing to the user, thereby suffering needless buffering overhead. Granted, using the buffer made it possible to develop ease of use techniques such as field placement, but if the JSP file developer decides not to use field placement, why suffer the performance penalty?

10.6.1 *IterationTag*

This performance penalty was solved in JSP1.2 with the introduction of the `IterationTag`, which can repeatedly execute its body for as long as it returns `EVAL_BODY_AGAIN` from `doAfterBody()`. Hence, all we need do is take the iteration framework that was developed in this chapter and have it work with the JSP1.2 `IterationTag`.

All our iteration-related code was part of a single class, `IterationTagSupport`, which is where we implemented our `doStartTag()`, `doBeforeBody()`, and `doAfterBody()`. All the tags that work with the iteration framework have only to extend `IterationTagSupport` and provide an implementation for a few methods. At this point, we only need to port `IterationTagSupport`, which requires the following steps:

- Remove any code portion related to the field placement (no buffering means no field placement).
- Return `EVAL_BODY_INCLUDE` from `doStartTag()` so that the JSP runtime includes the body's results into the stream flowing to the client.

- Export variables in `doStartTag()` instead of `doBeforeBody()`, since `IterationTag` does not have a `doBeforeBody()` method.
- Return `EVAL_BODY_AGAIN` from `doAfterBody()` as per the JSP1.2 specification.

When we have finished, our iteration tags can take advantage of the `IterationTag` interface and its improved performance. Listing 10.19 presents such an adaptation of `IterationTagSupport` to the JSP1.2 `IterationTag` interface.

Listing 10.19 `IterationTagSupport` adapted to the JSP1.2 `IterationTag`

```
package book.iteration;

import book.util.LocalStrings;
import book.util.ExTagSupport;
import book.util.StringUtil;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;

public abstract class IncludedIterationTagSupport
    extends ExTagSupport {

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

    protected IterationSupport elementsList = null;
    protected Object current;

    public int doStartTag()
        throws JspException
    {
        fetchIterationSupport();
        if(elementsList.hasNext()) {
            exportVariables();
            return EVAL_BODY_INCLUDE;
        }
        return SKIP_BODY;
    }

    public int doAfterBody()
        throws JspException
    {
        if(elementsList.hasNext()) {
            exportVariables();
            return EVAL_BODY_AGAIN;
        }

        return SKIP_BODY;
    }

    protected abstract void fetchIterationSupport()
        throws JspException;
}
```

```
protected void exportVariables()
    throws JspException
{
    current = elementsList.getNext();
    pageContext.setAttribute(id, current);
}

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

protected void clearServiceState()
{
    elementsList = null;
    current = null;
}
}
```

`IncludedIterationTagSupport` presented in Listing 10.19 is much less complicated than `IterationTagSupport`. This simplicity comes partially from the removal of the field placement code, and partially from the fact that we no longer need to handle the `BodyContent` buffer and write its content back to the user.

To summarize, all tags developed in this chapter should be able to run unmodified in JSP1.2 (as `BodyTag` is supported there). However, tags wishing to take advantage of the new `IterationTag` interface should abandon the field placement as a means of populating the iterator's fields, since the tags can then extend our new `IncludedIterationTagSupport` and gain performance improvements.

10.7 Summary

Iteration is a crucial task in almost any web application, yet until the arrival of custom JSP tags, it could only be accomplished using scriptlets. As we stated, iteration scriptlets render the JSP code difficult to read and maintain, and even worse, place a premium on the content developer's knowledge of Java. Custom tags fix these problems at a reasonably low price.

We also presented a generic way to develop iteration tags. In fact, the code developed for this chapter can be used in your own daily work (e.g., iteration on something that is not an `Array`, `Enumeration`, or `Iterator`) with a relatively small time investment. Simply extend `IterateTag` or `IterationTagSupport`, override a method, and gain full access to the custom tag iteration functionality.

As a last phase in enhancing the quality and ease-of-use of our iteration tags, body content processing was added to the iteration tags to make using the iterator

properties easier. This body content processing is by no means unique to iteration tags. In fact, you can implement it in any tags that extend the `BodyTag` interface and have complete control over their body. Body content processing in this way can speed up the work of the JSP developer, by allowing you to introduce simple, proprietary syntax in your JSPs. It should be considered an appealing alternative to using smaller custom tags, such as the show tags that we developed, especially in cases in which the size of the parsed content is small compared to the size of the entire page.

Next we'll see how we can integrate custom tags with a database to provide simple tag-based access to a database server.