12

JSP Tag Extensions

From the point of view of the JSP developer, the only significant new feature of JSP 1.1 is support for **tag extensions** (often referred to as **custom tags**). However, this proves to be very significant indeed.

Tag extensions look like HTML (or rather, XML) tags embedded in a JSP page. They have a special meaning to a JSP engine at translation time, and enable application functionality to be invoked without the need to write Java code in JSP scriptlets. Well-designed tag extension libraries can enable application functionality to be invoked without the appearance of programming.

So in this chapter, we'll look at the basics of writing your own tags and then in the next chapter we'll look at some more advanced examples. So in this chapter we will cover:

- Tag extension basics
- The anatomy of a tag extension
- How to deploy a tag library
- How to write custom tag extensions

Let's get started then with what tag extensions are all about.

Tag Extension 101

Consider the `<jsp:forward>` action provided by the JSP specification. This tag dispatches the current request to another page in the current web application. It can be invoked with the following syntax:

```
<jsp:forward page="next.jsp" />
```

We can also to add additional parameters to the request before forwarding with an extended usage of `<jsp:forward>`, which nests one or more `<jsp:param>` tags within the `<jsp:forward>` tag:

```
<jsp:forward page="next.jsp" >
  <jsp:param name="image" value="house.gif" />
</jsp:forward>
```

Tag extensions allow a vast range of new functionality to be added to the JSP language and they can be invoked in a similarly intuitive way. For example, I could create a tag named `<wrox:forward>`, specify what attributes and subtags, if any, it requires, and implement it to perform a custom action before forwarding. Not only can this be added simply into the web page, it enforces separation of code and presentation, decouples the call from the class that implements the functionality associated with the tag, and can be simply incorporated into a design tool.

The key concepts in tag extensions are:

- **Tag name:**
  A JSP tag is uniquely identifed by a combination of **prefix** (in this case `jsp`), and **suffix** (in this case `forward`), separated by a colon.
- **Attributes:**
  Tags may have attributes, which use the XML syntax for attributes. The `<jsp:forward>` tag above has one attribute (`page`), while the `<jsp:param>` attribute has two (`name` and `value`). Attributes may be required or optional.
- **Nesting:**
  Note how the `<jsp:param>` subtag is used in the second example above. Tag extensions can detect nested tags at runtime and cooperate. A tag directly enclosing another tag is called the *parent* of the tag it encloses: in the example above, the `<jsp:forward>` tag is the parent of the `<jsp:param>` tag.
- **Body content:**
  This is anything between the start and end elements in a JSP tag, excluding subtags. A tag extension can access and manipulate its body content. Neither the `<jsp:forward>` nor `<jsp:param>` tags require body content. We will see later an example of a tag that can reverse its body content. It will be invoked like this (the body content is shown in bold):

```
<examples:reverse>
  Able was I ere I saw Elba
</examples:reverse>
```

> **The functionality associated with a tag is implemented by one or more Java classes. The tag handler (the class implementing the tag itself) is a JavaBean, with properties matching the tag's XML attributes. A Tag Library Descriptor (TLD) file is an XML document that describes a tag library, which contains one or more tag extensions. The JSP `taglib` directive must be used to import the tag library's tags in each JSP that wishes to use any of them.**

Why, besides a clever syntax, might we choose to use tag extensions rather than JSP beans? Are tag extensions not simply another way of allowing JSP pages to parcel out work to Java classes?

Due to the richer interaction between the hosting JSP page and tag extensions, tag extensions can achieve directly what beans can only achieve in conjunction with scriptlets. Tag extensions may access the `PageContext`, write output to the output writer, redirect the response, and define scripting variables. As an indication of their power, *all* the standard JSP actions provided via tags of form `<jsp:XXXX>` could be implemented using tag extensions. The behavior of tag handlers is configured by their XML attributes and their body content (which can be the result of evaluating JSP expressions at runtime).

Typical uses of tag extensions are:

- To conceal the complexity of access to a data source or enterprise object from the page (possibly, the page author, who may not be experienced with enterprise data)
- To introduce new scripting variables into the page
- To filter or transform tag content, or even interpret it as another language
- To handle iteration without the need for scriptlets

Tag extensions can be used to deliver a range of functionality limited only by developers' imaginations and sensible programming practice.

Tag extensions differ from beans in that they are common building blocks, not tailored resources for a particular page or group of pages. Tags receive the attributes that control their behavior from the JSP pages using them, not the request to a particular JSP (as in the case of request-bean property mappings). A well-designed tag extension may be used in many JSP pages.

This reusability is particularly important. Since their implementation and interaction with the JSP engine is well defined in the JSP 1.1 specification, libraries of tag extensions can be developed and distributed, on a commercial or open source basis. Generic tags can be developed for particular industries or application types.

Tag extensions, although new to JSP, are a very old concept in dynamic page generation. Products such as Apple's WebObjects and BroadVision have delivered rich functionality through custom tags for years, although in a proprietary context. This experience in the use of custom tags can be valuable to JSP developers.

Tag extensions are a particularly welcome addition to the JSP developer's armory because they are easy to implement. The API surrounding them is relatively simple, and it is possible to use them to achieve results quickly. This is a reflection of the elegance of the design of the tag extension mechanism; in the true Java spirit it delivers rich functionality without excessive complexity.

The examples and most of the discussion in this chapter and the next assume that tag extensions will be used to generate HTML markup. While this is currently their most likely use, tag extensions can be used to generate any content type supported by JSP.

A Simple Tag

Before we look at the API supporting tag extensions and the supporting infrastructure in detail, let's implement a simple tag. The simplest case is a tag without attributes or body content, which outputs some HTML. We'll add some dynamic content to prove that the tag is alive. Say we want to see the following output:

Hello world.
My name is <tag handler implementation class> and it is now <date and time>

We'll call our simple tag `hello`. Here's how we might use it in a JSP. I've named this simple example `hello.jsp`. The first line is a declaration used to import the tag library, which we will discuss in detail later:

```
<%@ taglib uri="/hello" prefix="examples" %>

<html>
  <head>
    <title>First custom tag</title>
  </head>
  <body>
    This is static output.
    <p />
    <i>
      <examples:hello></examples:hello>
    </i>
    This is static output again.
  </body>
</html>
```

All we need to do to implement the tag is to define a tag handler (a Java class implementing the tag's functionality), and provide the tag library descriptor. We can then import the tag library into any JSP that requires it.

The tag handler class must react to callbacks from the JSP engine when the tag is encountered in JSPs. The most important of these are `doStartTag()`, called when the opening of the tag is encountered, and `doEndTag()`, called when the closing tag is encountered. The implementation of `HelloTag` is quite simple, as most of the work of implementing the custom tag is done by the `TagSupport` superclass provided by the JSP 1.1 API. Don't worry if some of the details are puzzling: we'll look at this class, as well as the API it uses, in more detail in a moment. Note that the tag has access to the `PageContext` object of each JSP that uses it.

```
package tagext;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Implementation of a tag to generate a single piece of HTML
public class HelloTag extends TagSupport {
```

```
  // This method will be called when the JSP engine encounters the start
  // of a tag implemented by this class
  public int doStartTag() throws JspTagException {
    // This return value means that the JSP engine should evaluate
    // the contents and any child tags of this tag
    return EVAL_BODY_INCLUDE;
  }

  // This method will be called when the JSP engine encounters the end
  // of a tag implemented by this class
  public int doEndTag() throws JspTagException {
    String dateString = new Date().toString();
    try {
      pageContext.getOut().write("Hello world.<br/>");
      pageContext.getOut().write("My name is " + getClass().getName() +
                              " and it's " + dateString + "<p/>");
    } catch (IOException ex) {
      throw new JspTagException
                ("Fatal error: hello tag could not write to JSP out");
    }

    // This return value means that the JSP engine should continue to
    // evaluate the rest of this page
    return EVAL_PAGE;
  }
}  // class HelloTag
```

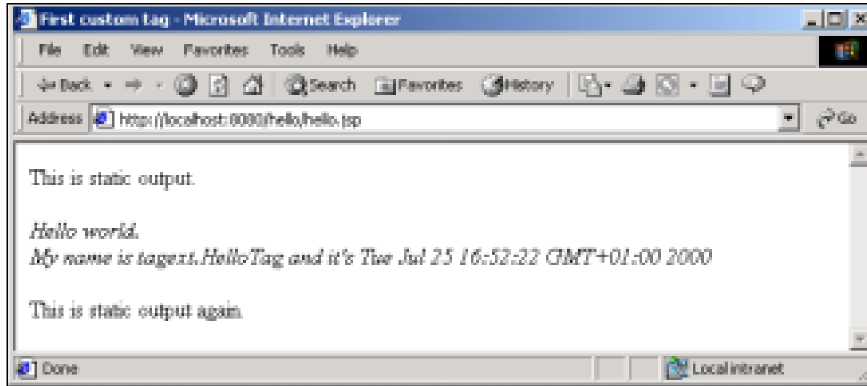
The tag library descriptor, which we'll name `hello.tld`, is required to map the tag to the tag handler class and defines the way JSPs may interact with the `HelloTag` class. The tag library descriptor is an XML document conforming to a DTD specified by Sun Microsystems:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>examples</shortname>

  <info>Simple example library.</info>

  <tag>
    <name>hello</name>
    <tagclass>tagext.HelloTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>Simple example</info>
  </tag>
</taglib>
```

The tag's suffix when used in JSPs must be `hello`, and its prefix is `examples`.

Note that as a tag library can include any number of tags, we'll add extra `<tag>` elements to this tag library descriptor for the remaining examples in this chapter.

Once everything is in the right place (see **Getting it running** below), `hello.jsp` should look like this in your browser:



You might wish to experiment to see what happens when you fail to close the tags, or try to access tags that are not found in your imported tag library. The resulting error messages are not specified in the JSP 1.1 specification, but most implementations should be reasonably informative and helpful.

Getting it Running

I'll discuss deployment options for tag libraries in more detail later, but in the meantime, let's look at how we can get this example running in Tomcat or any other server that supports the JSP 1.1 and Servlet 2.2 specifications.

If you have Tomcat, you can get started very quickly by copying `hello.war` (see instructions below) to the `webapps` directory in the Tomcat installation directory. Once you have done that, start Tomcat and type `http://localhost:8080/tagext/hello.jsp` in your favorite browser. The output as above should come up. Below is included a somewhat more detailed explanation of how this works.

The best option is to package the tag library descriptor, the Java classes that implement the tags in the library, and the JSPs that use the tags in a single, self-contained, Web ARchive file, or WAR, as we have done here. This file can then be loaded into a JSP 1.1-compliant JSP engine and constitutes a self-contained **application**, with its own **context path** on the web server. (A context path is the directory under the server root at which the server publishes the top-level directory of the contents of the WAR. For example, the contents of the root directory of our WAR are published by the web server at `<server root>/tagext`, assuming that `tagext` is the name we give our application when we load it into the server. The WAR's internal directory structure will determine the published directory structure *under* the context root.)

Remember that a WAR is a JAR file, with special directories and a file named `web.xml` located in its `/WEB-INF` directory. The following shows the structure of the WAR file for this simple example; the source can be found under the folder `hello` in the code download:

```
hello.jsp
META-INF/
         MANIFEST.MF
WEB-INF/
        web.xml
        classes/
                tagext/
                       HelloTag.class
        tlds/
             hello.tld
```

There are a few WAR conventions specific to using tag libraries. We should place our tag library descriptor files in the `WEB-INF/tlds` directory. We need to use a special element in the `web.xml` file, `<taglib>`, to let the server know where to find the tag library's TLD within the WAR when JSPs in the WAR attempt to import it. The following is the `web.xml` file for the simple example:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN'
'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>

<web-app>
  <display-name>tagext</display-name>
  <description>Tag extensions examples</description>

  <session-config>
    <session-timeout>0</session-timeout>
  </session-config>

  <!-- Tag Library Descriptor -->
  <taglib>
    <taglib-uri>/hello</taglib-uri>
    <taglib-location>/WEB-INF/tlds/hello.tld</taglib-location>
  </taglib>

</web-app>
```

The easiest way to create the WAR is first to create the directory structure corresponding to the WAR's structure in your development environment. All we then need to do to build the WAR is to then issue the following command in the WAR's root directory. Note that we exclude the `.java` source files, which would unnecessarily inflate our WAR and may cause problems when we attempt to deploy it:

```
jar -cvf hello.war META-INF/MANIFEST.MF WEB-INF/classes/tagext/*.class WEB-
INF/tlds/hello.tld WEB-INF/web.xml *.jsp
```

As mentioned above, deploying this WAR on Tomcat 3.1 simply requires copying the WAR into the `$TOMCAT_HOME/webapps` directory. When Tomcat is started, it automatically unpacks the WAR and creates the application, with the application's name (and context path) being the name of the WAR. There is no need to make any changes to the system or server classpath. Each web application will be given its own classloader at runtime.

There is another way of deploying WARs in Tomcat, which is much more convenient during development. Tomcat lets us work on a WAR that exists not as a single file, but as an expanded directory structure. This makes development a lot easier. It's not necessary to restart Tomcat when

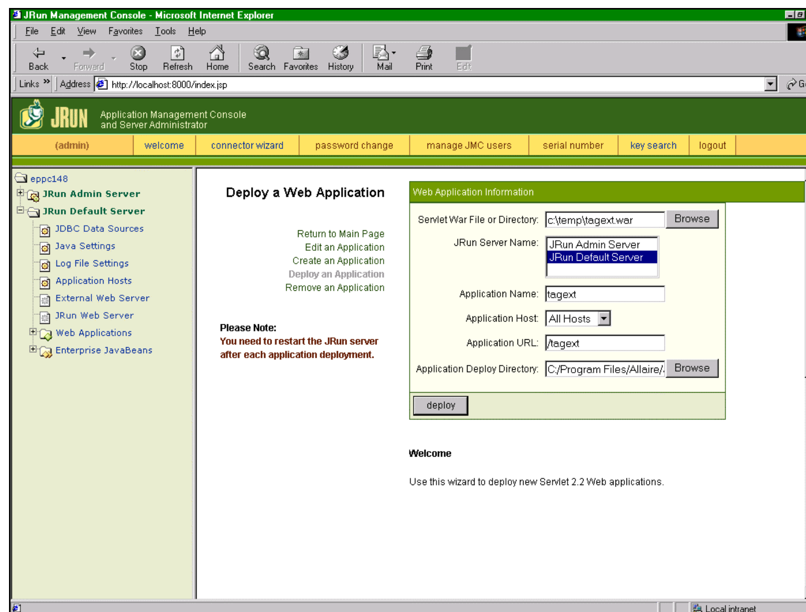
changes are made to the JSPs, and it's not necessary to rebuild the archive every time you make a change, yet it's very easy to build the WAR when it is ready for release. To use this approach, add the following lines to the `server.xml` file, which is located in the `$TOMCAT_HOME/conf` directory. The `<Context>` element is a subelement of `<ContextManager>`.

```
<Context path="/tagext" docBase="<path to root of war>"
         defaultSessionTimeOut="30" isWARExpanded="true"
         isWARValidated="false" isInvokerEnabled="true"
         isWorkDirPersistent="false" />
```

`<path to root of war>` will be an absolute path following the directory convention of your operating system, and need not be under the Tomcat directory tree. I deployed this WAR on Windows NT, using `docBase="c:\ProJavaServer\Chapter12\hello"`. On a Unix system I might have used `docBase="/home/johnsonr/ProJavaServer/Chapter12/hello"`. (Note that Tomcat requires only single backslashes for Windows paths, unlike many Java programs, which require double backslashes.) Note the `isWARExpanded` attribute in the Tomcat `<Context>` element above.

Remember that WARs are portable. Other JSP engines will use different deployment conventions to those of Tomcat, but the WAR itself will not change. In JRun 3.0, the deployment process is as simple, and managed by a web interface. To deploy our application, log into the JRun Management Console (JMC), select the desired server from the tree in the left hand frame (the JRun Default Server is the correct choice for deployment in a new installation), and choose the WAR Deployment link. You will be prompted to select the path to the WAR, the application name – in this case, `tagext` – and the application URL (`/tagext`). Once the form is complete, press the Deploy button. When you see the message confirming successful deployment, restart JRun (also through the JMC), and your new web application will be available. If you used the default configuration, it will be published at `http://localhost:8100/tagext/`.

The following screenshot shows the JRun Management Console's WAR deployment form filled in with the values from my system for the examples in this chapter:



Anatomy of a Tag Extension

Before we return to our simple example, let's cover some basic theory of tag extensions.

A number of components are required to implement a tag extension. The minimal requirement is a tag handler and a tag library descriptor.

- A **tag handler** is a Java bean implementing one of two interfaces defined in the `javax.servlet.jsp.tagext` package, `Tag` or `BodyTag`. These interfaces define the lifecycle events relevant to a tag; most importantly, the calls the class implementing the tag will receive when the JSP engine encounters the tag's opening and closing tags.
- A **tag library descriptor**, as we have seen, is an XML document containing information about one or more tag extensions.

More complex tags will require an additional class extending the abstract class `javax.servlet.jsp.tagext.TagExtraInfo` to provide information about scripting variables that are made available to JSPs through the use of tags.

`TagExtraInfo` subclasses may also perform custom validation of tag attributes. Of course, the classes implementing a tag may require any number of helper classes, which will need to be packaged with the tag for it to be a complete deployable unit.

Before tags can be used in a JSP, the **`taglib`** directive must be used to import a tag library and associate the tags it contains with a prefix.

Let's look at each of these requirements in turn.

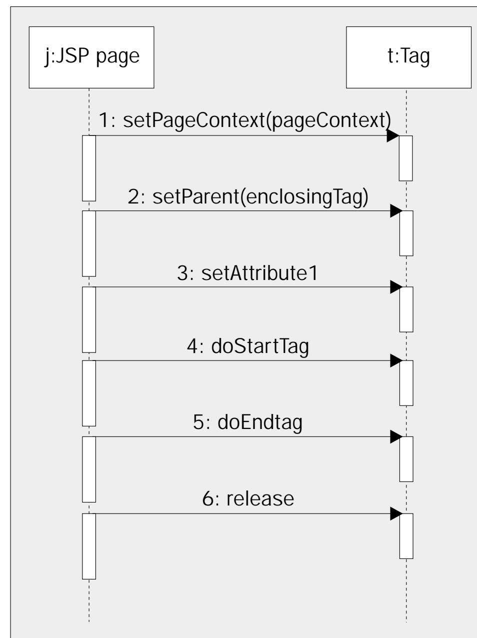
Tag Handlers

When the JSP engine encounters a tag extension in a JSP at translation time, it parses the tag library descriptor to find the required tag handler class, and generates code to obtain, and interact with, the tag handler. The `Tag` or `BodyTag` interfaces, one of which must be implemented by any tag handler, define callbacks that the servlet resulting from the JSP engine's code generation will make to the tag handler instance at runtime.

For performance reasons, JSP engines will not necessarily instantiate a new tag handler instance every time a tag is encountered in a JSP. Instead, they may maintain a pool of tag instances, reusing them where possible. When a tag is encountered in a JSP, the JSP engine will try to find a `Tag` *instance that is not being used, initialize it, use it and release it (but not destroy it), making it available for further use. The programmer has no control over any pooling that may occur. The repeated use model is similar to a servlet lifecycle, but note one very important difference: tag handler implementations don't need to concern themselves with thread safety. The JSP engine will not use an instance of a tag handler to handle a tag unless it is free. This is good news: as with JSP authoring in general, developers need to worry about threading issues less often than when developing servlets.*

The javax.servlet.jsp.tagext.Tag Interface

The `Tag` interface defines a simple interaction between the JSP engine and the tag handler, sufficient for tags that don't need to manipulate their body content. Its core methods are the calls implementing classes will receive when the JSP engine encounters the tag's opening and closing tags, `doStartTag()` and `doEndTag()`. Before we look at the method contracts in more detail, a sequence diagram helps to visualize the calls made to the tag handler by the compiled servlet. Assume that the container already has a tag handler instance available, and in the default state:



Let's look at the messages in more detail:

- The container initializes the tag handler by setting the tag handler's `pageContext` property, which the tag handler can use to access information available to the JSP currently using it.
- The container sets the tag handler's `parent` property. (Parent may be set to `null`, if the tag is not enclosed in another tag.)
- Any tag attributes defined by the developer will be set. This is a mapping from the XML attributes of the tag to the corresponding properties of the tag handler bean. For example, in the case of a tag invoked like this: `<mytags:test name="John" age="43" />`, the container will attempt to call the `setName()` and `setAge()` methods on the tag handler. The container will attempt to convert each attribute to the type of the corresponding bean property: for example, the `String "43"` will be converted to an `int` in this case. If the type conversion fails, an exception will be thrown and must be handled by the calling JSP page. (From a JSP's point of view, there is no difference between an exception thrown by a tag handler and one thrown by an expression of scriptlet in the page.)

- Next, the container calls the tag handler's `doStartTag() method`.
- The container calls the `doEndTag()` method.
- The container calls the `release()` method. This is not equivalent to a finalizer. Tag handlers differ from page beans in that their lifecycle is entirely independent of that of the JSPs that use them. Tag handlers must support repeated use before destruction, possibly in a number of JSPs. The implementation of the `release()` method must ensure that any state that may cause conflict in future uses is reset, and that any resources required during the tag's execution are freed.

Lets look at the `doStartTag()` and `doEndTag()` methods:

```
int doStartTag() throws JspException
```

Called after the tag has been initialized, when the JSP engine encounters the opening of a tag at run time. Its return value should be one of two constants defined in the `Tag` interface: `EVAL_BODY_INCLUDE`, which instructs the JSP engine to evaluate both the tag's body and any child tags it has, or `SKIP_BODY`, which instructs the JSP engine to ignore the body. This can throw a `JspException`, as will most of the methods in the tag handler API when an error condition is encountered; how it will be handled will depend on the JSP page using the tag. Most JSP pages will use an error page, so an exception thrown in a tag will abort the rendering of the page.

```
int doEndTag() throws JspException
```

`doEndTag()` is called when the JSP engine encounters the closing tag of an element at run time. Its return value can be `EVAL_PAGE` or `SKIP_PAGE`. `EVAL_PAGE` will cause the JSP engine to evaluate the rest of the page, `SKIP_PAGE` to terminate evaluation of the page. The `SKIP_PAGE` return value should be used only with very good reason; using tag handlers to terminate page evaluation is even worse than sprinkling random `return` statements in Java code, and may be confusing to the reader. A legitimate use might be to terminate page output if it is established that the user has insufficient privileges to view the whole of the page.

There are also a number of methods that relate to tag nesting, initialization, and reuse:

```
Tag getParent()
void setParent()
```

The specification also requires methods to expose the `parent` property. A tag's parent is the tag that directly encloses it in a JSP, or `null` if there is no enclosing tag. Tag implementations can query their parent at runtime, to obtain context information:

```
void setPageContext (PageContext pc)
```

`setPageContext()` is an initialization method, making the `PageContext` of the JSP available to the tag.

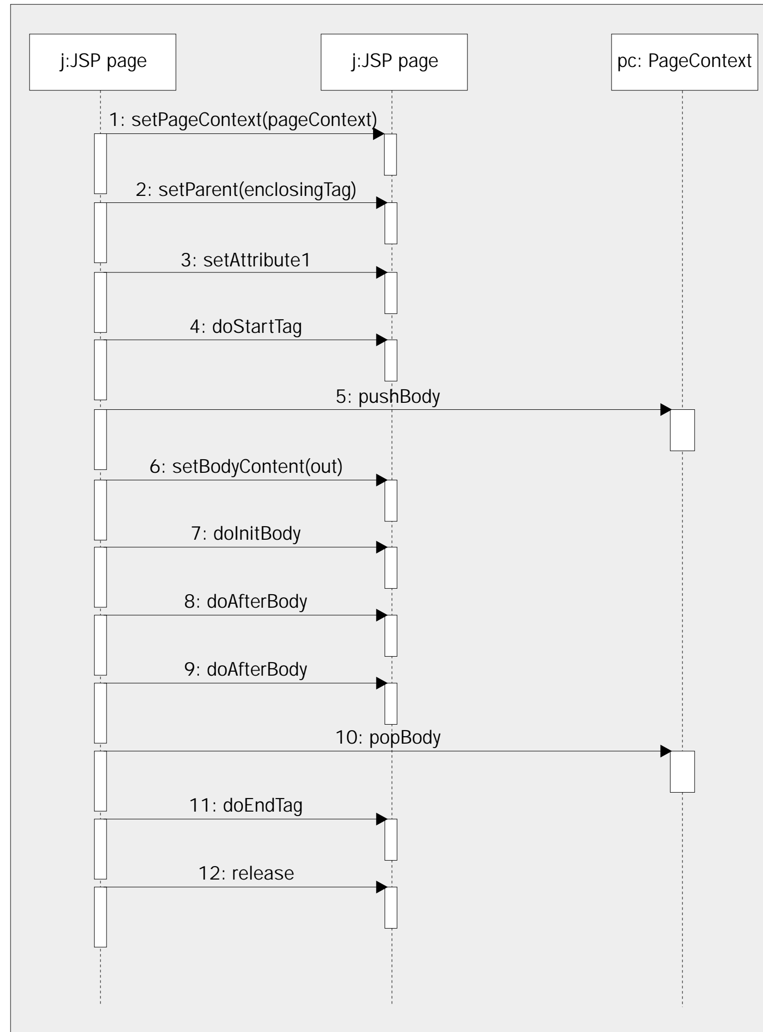
```
void release()
```

`release()` is a call to the tag handler to release any resources. Examples of this may be closing JDBC connections or open sockets that the handler requires for its function and to clear any state associated with it. The second is an often overlooked task in this context.

The javax.servlet.jsp.tagext.BodyTag Interface

`BodyTag` extends `Tag`, adding extra callbacks and other methods allowing the developer to work with the content of the tag.

A sequence diagram displays the interaction between calling JSP and tag handler in the case of a body tag; as you can see, it is somewhat more complex than that of the `Tag` interface:



The extra steps involve the preservation of the JSP's `JSPWriter` (messages 5 and 10), and the possibility of repeated calls to the `doAfterBody()` method, which enables the `BodyTag` implementation to take control of the tag's execution at runtime.

The more significant methods of the `BodyTag` interface are listed below.

`int doInitBody() throws JspException`

Called after the tag has been initialized and `doStartTag()` has been called. Its return value should be `EVAL_BODY_TAG`, in which case the tag's body content and any child tags will be evaluated, or `SKIP_BODY`, in which case the body will be ignored. Watch out that you don't try to return `EVAL_BODY_INCLUDE` from a `BodyTag`'s `doInitBody()` or `doStartTag()` methods. The JSP engine will throw a `JspException` if it detectes this.

`int doAfterBody() throws JspException`

`doAfterBody()` is called each time the tag's body has been processed. The return values are `EVAL_BODY_TAG` and `SKIP_BODY`. `EVAL_BODY_TAG` directs the JSP engine to evaluate the tag's body and any child tags *again* (resulting in at least one more call to this method), `SKIP_BODY`, causes processing of the body content to terminate. This can be used to conditionally loop through the tag content.

`void setBodyContent(BodyContent bodyContent)`

Initialization method to set the class used to manipulate body content.

The javax.servlet.jsp.tagext.BodyContent Class

The `BodyContent` class is the key to `BodyTag` functionality. `BodyContent` is a subclass of `JspWriter` that can be used to manipulate the body content of `BodyTag` implementations and store it for later retrieval. The `getBodyContent()` method of `BodyTagSupport` returns the `BodyContent` instance associated with a particular tag.

To understand the way in which the `BodyContent` class works, consider how `JspWriter` objects are handled in JSPs using `BodyTag`s: messages 5 and 10 from the sequence diagram above. Before the `BodyTag` begins to evaluate its body content, the generated JSP implementation class includes the following line:

```
out = pageContext.pushBody();
```

After the `BodyTag`'s methods have been called, it includes a matching call:

```
out = pageContext.popBody();
```

What this means is that each `BodyTag` has a kind of play area, enabling it to manipulate its `BodyContent` without automatically affecting the `JspWriter` of the enclosing JSP page or tag. To generate output, the `BodyTag` needs to write the contents of its `BodyContent` into its enclosing writer explicitly (see below). This is the key difference between `BodyTag`s and `Tag`s: `Tag` implementations have no such flexibility, and therefore cannot modify or suppress their body content, although they can prevent it from being evaluated altogether by returning `SKIP_BODY` in their implementation of `doStartTag()`.

The most interesting methods in the `BodyContent` class are:

`void clearBody()`

Clears the body content. Useful if we want to manipulate the body content before writing it out.

`JspWriter getEnclosingWriter()`

Returns the enclosing `JspWriter`; this may be the writer of an enclosing tag, or the writer of a JSP itself. We normally use this method to get a JSP writer to which we can write the body content stored in a body tag when we have finished manipulating it. For example, we have used the following lines of code in the `doEndTag()` method of a number of `BodyTag` implementations in this chapter:

```
BodyContent bodyContent = getBodyContent();
if (bodyContent != null) {
  bodyContent.getEnclosingWriter().write(sbOut.toString());
}
```

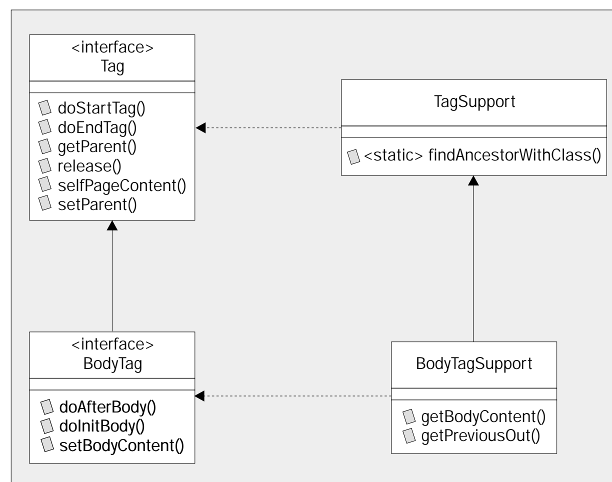
This ensures that if there is any body content held in the body tag, it will be written to the enclosing JSP writer.

`String getString()`

This returns the content already held in the body content, as a `String`. This is useful if we need to examine what has been added to the body content with each iteration of a loop.

Convenience Classes

Some of the methods in `Tag` and `BodyTag` will be implemented the same way in most tags. So the `javax.servlet.jsp.tagext` package includes two convenience implementations of `Tag` and `BodyTag`: `TagSupport` and its subclass `BodyTagSupport`. Classes implementing tag extensions will normally be derived from one of these. The class diagram below shows the relationship between these classes and the `Tag` and `BodyTag` interfaces:



TagSupport and BodyTagSupport are concrete, not abstract classes, so they provide complete implementations of the corresponding interfaces, which do nothing except return the appropriate values to cause the JSP engine to continue rendering the page. So developers can safely omit methods that they are not interested in. Developers don't usually concern themselves with handling the parent property and setPageContext(). The release() method can also be omitted if it is not necessary to free resources or return the tag to its default state. The methods that a developer *will* normally want to override are doStartTag() and doEndTag() for all tags, and doInitBody() and doAfterBody() for BodyTags specifically.

TagSupport also makes an important convenience variable available to subclasses: pageContext (the saved PageContext which was set by the JSP engine when the tag was first used in a page). BodyTagSupport provides a getBodyContent() method, necessary to obtain a tag's BodyContent before manipulating it.

> *Like me, you might find the names TagSupport and BodyTagSupport confusing. These classes are standard implementations of the Tag and BodyTag interfaces. It should be pointed out that this naming is inconsistent with Sun's usual practice and Java convention; I think of them as TagImpl and BodyTagImpl.*

The javax.servlet.jsp.tagext.TagExtraInfo Class

Metadata classes extending the TagExtraInfo abstract class may be associated with tag handlers to provide extra information to a JSP engine. This optional association is specified in the tag library descriptor.

We will look at the implementation of a TagExtraInfo class in more detail later in this chapter, but the two methods you are most likely to override are:

```
VariableInfo[] getVariableInfo(TagData td)
```

This method is used to return information about scripting variables that the tag makes available to JSPs using it. It returns an array of VariableInfo objects, which contain information about the name of each scripting variable and its fully qualified class name. It describes whether or not the variable should be declared or whether the tag will merely overwrite the value of an existing variable, and the scope of the variable.

```
boolean isValid(TagData data)
```

This is sometimes used to validate the attributes passed to a tag at translation time. As an example, consider a tag with four attributes. Three may be optional, but if one of the optional attributes is specified the whole three must be present. There is no way to specify this behavior in a tag library descriptor. However, the isValid() method of the appropriate TagExtraInfo subclass could be implemented to return false if the parameters supplied at translation time are invalid. The default implementation of isValid() in TagExtraInfo always returns true. Note that *runtime* attribute validation is entirely different, and is the responsibility of the tag handler concerned.

Objects Available to Tag Handlers

All tag handlers have access to more context information than do most beans. This is available through the PageContext object they are passed on initialization. As you'll recall, javax.servlet.jsp.PageContext is a convenient holder for information about the runtime of a JSP page, including the request and response objects, and references to objects such as beans associated with the JSP.

This amount of access equals power. Note, though, that it is poor style to modify `request` and `response` directly from a tag handler. Custom tags should be thought of as generic building blocks intended for use in a wide variety of contexts. In practice tags should not be concerned with the parameters passed to the JSP page. Although a tag handler *can* access request parameters, relying on doing so will greatly reduce its reusability.

The Simple Example Revisited

To catch our breath after all this theory, let's look again at the Java implementation of the simple example we introduced earlier. We see that the tag handler extends `TagSupport`, and so gets most of its functionality for free. It has no state and accesses no file or other resources, so there is no need to override `release()`. We use `doEndTag()` to access the `PageContext`, obtain a `JspWriter`, and generate output:

```
package tagext;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

//  Implementation of a tag to generate a single piece of HTML.
public class HelloTag extends TagSupport {

//  This method will be called when the JSP engine encounters the start
//  of a tag implemented by this class
  public int doStartTag() throws JspTagException {
    // This return value means that the JSP engine should evaluate
    // the contents and any child tags of this tag
    return EVAL_BODY_INCLUDE;
  }

//  This method will be called when the JSP engine encounters the end
//  of a tag implemented by this class
  public int doEndTag() throws JspTagException {
    String dateString = new Date().toString();
    try {
      pageContext.getOut().write("Hello world.<br/>");
      pageContext.getOut().write("My name is " + getClass().getName() +
                                 " and it's " + dateString + "<p/>");
    }
    catch (IOException ex) {
      throw new JspTagException
                ("Fatal error: hello tag could not write to JSP out");
    }

    // This return value means that the JSP engine should continue to
    // evaluate the rest of this page
    return EVAL_PAGE;
  }
}  // class HelloTag
```

Note that we need to check for `IOException`s when generating output. Any exception encountered while processing the tag must be wrapped as a `JspException` if it is to be rethrown; it is good practice to use the `javax.servlet.jsp.JspTagException` subclass of `JspException`. (Note that, confusingly, this *isn't* in the same package as the other classes specific to tag handlers.)

We could actually have omitted the `doStartTag()` method. I include it for completeness, but in fact it does exactly what its superclass `TagSupport`'s `doStartTag()` method does: instruct the JSP engine to evaluate the tag's content and any subtags.

Tag Library Descriptors

Tag Library Descriptors or **TLDs** are XML documents with a `.tld` extension that describe one or more tag extensions. TLDs must conform to the Document Type Definition (DTD) included in the JSP 1.1 specification. Many of the elements are intended to provide support for JSP authoring tools, although such tools are yet to be widely available in the market.

The root element is `<taglib>`. It's defined in the DTD by:

```
<!ELEMENT taglib
  (tlibversion, jspversion?,
  shortname, uri?, info?,
  tag+) >
```

- `tlibversion` is the version of the tag library implementation. This is defined by the author of the tag library.
- `jspversion` is the version of JSP specification the tag library depends on. At the time of writing the value you should use is 1.1 (the default). The element is optional.
- `shortname` is a simple default name that could be used by a JSP authoring tool; the best value to use is the preferred prefix value: that is, a suggestion as to a prefix to use when importing the tag library. Although there is no way of enforcing this, hopefully developers using the library will follow this suggestion, and consistency will be achieved between all users of the tag library. The `shortname` should not contain whitespace, and should not start with a digit or underscore.
- `uri` is an optional URI uniquely identifying this tag library. If it is used, the value will normally be the URL of the definitive version of the tag library descriptor.
- `info` is an arbitrary text string describing the tag library. Think of it as the equivalent of a Javadoc comment relating to an entire class or package; the authoring tool may display it when the tag library is imported.

The `<tag>` element is the most important. It's defined in the DTD as:

```
<!ELEMENT tag
  (name, tagclass, teiclass?,
  bodycontent?, info?, attribute*) >
```

- `name` is the name that will identify this tag (after the tag library prefix).
- `tagclass` is the fully qualified name of the tag handler class that implements this tag. This class must implement the `javax.servlet.jsp.tagext.Tag` interface.
- `teiclass` stands for `TagExtraInfo` class, and defines the subclass of `javax.servlet.jsp.tagext.TagExtraInfo` that will provide extra information about this tag at runtime to the JSP. Not all tags require a `TagExtraInfo` class.

- bodycontent is an optional attribute specifying the type of body content the tag should have. Three values are legal: tagdependent, JSP, and empty. The default (and most useful) is JSP, which means that the tag's body content will be evaluated at run time like any other JSP content. tagdependent signifies that the JSP engine should *not* attempt to evaluate the content, but accept that while it may not understand it, it means something to the tag handler, and should therefore be passed unchanged. empty is useful when a tag should not have any body content. If this value is used, and the tag is not empty, JSP translation will fail.

<attribute> sub-elements describe each attribute accepted (or required) by the tag. The DTD definition is:

```
<!ELEMENT attribute
  (name, required?, rtexprvalue?) >
```

- name is the name of this attribute, as it will appear in JSPs using the tag.
- required specifies whether or not this attribute is mandatory. The valid values are true (the attribute is required), and false (the default, signifying an optional attribute). The attribute may have a default value.
- rtexprvalue specifies whether the attribute value can be the result of a JSP expression, or whether it has a fixed value at translation time when the tag is used in a JSP. Valid values are true and false. Again, the default is false, meaning that expressions are forbidden. If rtexprvalue is true, the following will be legal:

```
<examples:mytag attrib="<%=myObject.getValue()%>">
```

  Allowing attributes to take expression values can be very useful. Setting attributes by the use of expressions allows their behavior to be determined at runtime. For example, very often tag attributes will be set to the value of properties of JSP beans. This relies on the use of a JSP expression.

The simple example's TLD was very straightforward. As this tag takes no attributes and has no associated TagExtraInfo class, only the bare minimum of elements is required:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
        PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>examples</shortname>

  <info>Simple example library.</info>

  <tag>
    <name>hello</name>
    <tagclass>tagext.HelloTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>Simple example</info>
  </tag>
</taglib>
```

Although the XML structure is not complex, and at present TLDs will usually be written by hand, as JSP 1.1 becomes more widely supported tool support can be expected. This will synchronize TLDs and the relevant Java classes, avoiding time-wasting trivial errors.

Using Tag Extensions in JSP Pages

Unlike the standard actions such as `<jsp:forward>`, custom tags must be explicitly imported into JSP pages that wish to use them. The syntax for the `taglib` directive is shown below:

```
<%@ taglib uri="http://www.tagvendor.com/tags/tags.tld" prefix="examples" %>
```

The `uri` attribute tells the JSP engine where to find the TLD for the tag library. The `prefix` attribute tells the JSP engine what prefix will be given to tags from this library in the remainder of the JSP.

A JSP may import any number of tag libraries. The `taglib` directive will cause an exception at translation time if the tag library cannot be located; the first attempted access to any tag defined in the TLD will cause an exception at runtime if all the classes required to support the tag implementation cannot be loaded.

Once the tag library has been imported into the page, tags in a library can be called as follows:

```
<examples:someTag name="Rod">
...
</examples:someTag>
```

The way in which custom tags are used in JSPs is an example of Sun's efforts to introduce XML conventions into JSP syntax. Note that, unlike HTML attributes, the attributes of custom tag *must* be enclosed in double quotes, in accordance with the XML specification. (Of course it is good practice to write XML compliant HTML markup, but browsers do not currently enforce it.) Tag prefixes use the same syntax as XML namespaces.

When a tag requires no body content, it is best to use the XML shorthand to make this explicit:

```
<examples:hello name="Rod" />
```

Tag prefixes are defined in JSPs, not, as one might expect, in tag libraries. Choice of prefix is a matter for developers, but consistency among JSP pages importing the same tag library is advisable. It is best to adopt the value of the `shortname` element in the tag library. The prefixes `jsp:`, `jspx:`, `java:`, `javax:`, `servlet:`, `sun:`, and `sunw:` are reserved. It's perhaps unfortunate that Sun has not defined a unique naming system such as the Java package naming system for tag library prefixes. Choosing a prefix unique to a company or organization is advisable: for example, instead of using the potentially clashing short name `tables`, it might be advisable to use `myCompany_tables`.

Deploying and Packaging Tag Libraries

There are three main ways of deploying and using tag libraries with a JSP engine. JSP developers must be familiar with all three, because each of them calls for slightly different syntax in the `taglib` directive. (This means that JSPs need to be modified slightly if they are to be deployed in a different way – surely an oversight in the JSP specification.)

No Packaging

The first and simplest means of deployment is simply placing the tag library descriptor under the server's document root, and the Java classes required to implement the tags in the server (or system) classpath. There is no attempt to package a tag library or an application. In this case, the `taglib` directive will look like this:

```
<%@ taglib uri="./hello.tld" prefix="examples" %>
```

The `uri` is simply a path on the host server, which may be relative (as in this example) or absolute. In this approach, the tag library desciptor (although not the classes implementing the tag handler) is always publicly available: anyone could view it by simply typing in its URL. This approach is easy to work with, but can create problems at deployment time: the JSP engine or system's classpath has to be hand edited to include the classes implementing the tag.

WAR

In a second approach to deployment, the tag library descriptor, the Java classes required to implement the tags, and the JSPs that use the tag library can be shipped together as a *web application*, in a Web ARchive file (better known a WAR file). This is the approach we've taken in this chapter. It is very attractive because it offers painless portability between servers and very easy deployment. In this case, the `taglib` directive will look like this:

```
<%@ taglib uri="/hello" prefix="examples" %>
```

Note that we don't specify the actual filename of the TLD, so there is no need to use the TLD extension. The server knows where to look in the current web application's WAR for a `.tld` file matching this URI because the mapping from URI to file location is specified in the `web.xml` file in a `<taglib>` element. The complete `web.xml` file for our simple example looked like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN'
'http://java.sun.com/j2ee/dtds/web-app_2.2.dtd'>

<web-app>
  <display-name>tagext</display-name>
  <description>Tag extensions examples</description>

  <session-config>
    <session-timeout>0</session-timeout>
  </session-config>

  <taglib>
    <taglib-uri>/hello</taglib-uri>
```

```
    <taglib-location>/WEB-INF/tlds/hello.tld</taglib-location>
  </taglib>

</web-app>
```

The <taglib> element contains two subelements: <taglib-uri> specifies the URI that should be used in JSPs wishing to use the tag library; <taglib-location> specifies the path to the tag library descriptor relative to the web.xml file. Note that this path need not be publicly available to users of the web server: the server will not publish anything that is in the WEB-INF directory.

Remember the important directories in the WAR:

- WEB-INF:
  This contains the web.xml file, in which the TLD URI-location mapping must be specified.
- WEB-INF/classes:
  This contains Java classes required to implement tag libraries or otherwise support the functionality of the web application.
- WEB-INF/lib:
  This contains JAR files containing additional classes required to support the functionality of the web application.
- WEB-INF/tlds:
  By convention (although not mandated in any specification) this contains the tag library descriptors (but not tag handler classes) that will be made available to JSPs in the web.xml file. The TLDs could actually be placed anywhere in the WAR (so long as a mapping is included in the web.xml file), but adhering to this convention makes it easier to comprehend the WAR's structure.

Tag Library JAR

In a third approach to packaging and deployment, a tag library may be distributed in a JAR file whose META-INF subdirectory contains the tag library descriptor. The JAR file should also contain the classes required to implement the tags defined in the tag library, but *not* the JSPs that use the tag library. In this case, the taglib directive in JSP pages should refer to this JAR, which must be available to the JSP engine via a URL (or mapped URL). This enables custom tags to be supplied in self-contained units – a vital precondition for the successful distribution of third-party custom tags.

The taglib directive will look like this:

```
<%@ taglib uri="/tagext/hellotags.jar" prefix="examples" %>
```

The easiest way to JAR tag extension classes is to create a META-INF directory containing the tag library descriptor (renamed taglib.tld if necessary) under the root of your Java package hierarchy (that is, parallel to com). The jar tool can then be invoked easily from the root directory. The example below shows the process of creating a JAR file from the second group of examples in the next chapter, and the eventual contents of the file. The viewJar directory is the base of the Java package tree (here containing only the jspstyle package):

A recursive directory listing reveals the following files:

```
\viewJar\
         jspstyle\
                  CellTag.class
                  CellTagExtraInfo.class
                  HeadingCloseTag.class
                  HeadingOpenTag.class
                  HeadingTag.class
                  HeadingTagExtraInfo.class
                  ListTag.class
                  ListTagExtraInfo.class
                  NameValueModel.class
                  NameValueTag.class
                  NameValueTagExtraInfo.class
                  RowCloseTag.class
                  RowOpenTag.class
                  RowsTag.class
                  RowsTagExtraInfo.class
                  RowTagsExtraInfo.class
                  StyledXMLTag.class
                  TableTag.class
         META-INF\
                  taglib.tld
```

The JAR file can be created running the command:

```
jar -cvf viewslib.jar jspstyle/*.class /META-INF
```

in the `\viewJar` directory.

The JAR's contents, shown by the command:

```
jar –tvf viewslib.jar
```

will be:

```
META-INF/
         MANIFEST.MF
         taglib.tld
jspstyle/
         CellTag.class
         CellTagExtraInfo.class
         HeadingCloseTag.class
         HeadingOpenTag.class
         HeadingTag.class
         HeadingTagExtraInfo.class
         ListTag.class
         ListTagExtraInfo.class
         NameValueModel.class
         NameValueTag.class
         NameValueTagExtraInfo.class
         RowCloseTag.class
```

```
RowOpenTag.class
RowsTag.class
RowsTagExtraInfo.class
RowTagsExtraInfo.class
StyledXMLTag.class
TableTag.class
```

Note that although I wrote this particular example using Windows NT, the `jar` tool (which is written in Java) uses identical syntax when running on other operating systems.

Warning: Tomcat 3.1 can produce translation errors if `.java` source files are included in tag library JARs.

Combination of WAR and JAR

A combination of the second and third delivery methods is often useful. For example, consider a web application that uses a tag library that may also be of value in other web applications. The best approach is to package the tag library as a JAR, place this JAR in the `/META-INF/lib` directory of the web application's WAR, and create a mapping in the WAR's `web.xml` file to the tag library's TLD. This approach works well; the only issue is that the TLD must be extracted from the tag library JAR for it to be picked up by the `web.xml` file or JSPs in the WAR. Unfortunately the `<taglib>` element's mapping does not work directly to a JARed tag library. The tag library JAR can be directly imported into JSPs if it is publicly accessible, under the web application's root, but this is somewhat less elegant. It's seldom a good idea to publish more information than is strictly necessary.

This combination approach will be used later in the next chapter, as we'll need to reuse some of the tags to support the application developed in Chapter 14.

Writing Tag Extensions

Once the initial concepts are grasped, implementing tag extensions is surprisingly easy.

Processing Attributes

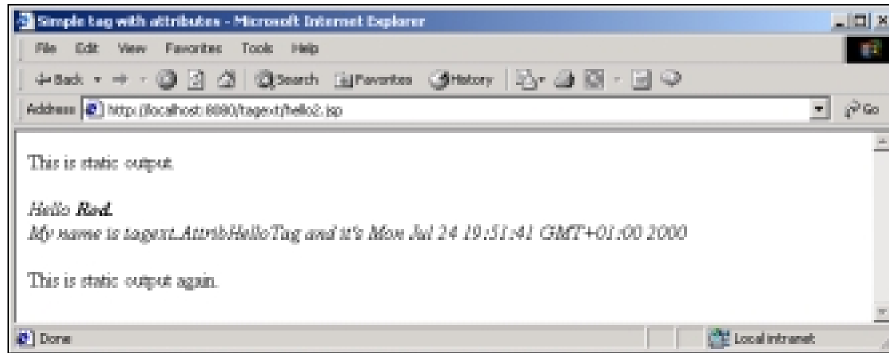
Our simple example is all very well, but it doesn't take advantage of the dynamic potential of custom tags. We *could* interrogate the `PageContext` to implement context-specific behavior, but there are far better alternatives.

The easiest way to parameterize tags is to pass in XML attributes.

How do we make our tags handle attributes? The answer, not surprisingly, is that attributes in a TLD `tag` element map onto bean properties of the corresponding tag handlers. The mapping of attributes onto tag handler properties is, as we might expect, handled by the JSP engine using reflection and not only does it work with primitive types, we can pass *any* type to a tag handler. (Draft versions of the JSP 1.1 specification included a `type` subelement of the `attribute` TLD element; this has now been removed.)

Attributes can be either required or optional. This is specified in the TLD, as is whether attributes can take the value of JSP expressions at runtime, as we mentioned earlier.

Let's suppose we decide to pass a name as an attribute, and change our simple example to display the following:



First, we need to write a tag handler with a `name` property. With this minor change, it's pretty much like `HelloTag`:

```
package tagext;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Hello tag accepting a name attribute.
public class AttribHelloTag extends TagSupport
{
  private String  name;

  // Property getter for name
  public String getName() {
    return name;
  }

  // Property setter for name
  public void setName(String name) {
    this.name = name;
  }

  public int doEndTag() throws JspTagException {
    String dateString = new Date().toString();
    try {
      pageContext.getOut().write("Hello <b>" + name + "</b>.<br/>");
      pageContext.getOut().write("My name is " + getClass().getName() +
                                 " and it's " + dateString + "<p/>");
    }
    catch (IOException ex) {
      throw new JspTagException("Hello tag could not write to JSP out");
    }
    return EVAL_PAGE;
  }
}
```

You should save this in `WEB-INF/classes/tagext`. Don't be tempted to omit the property 'getter', which at first sight seems unnecessary. The JavaBeans specification requires both getter and setter for a bean property, and some JSP engines may rely on the presence of both methods at translation time. (JRun 3.0, unlike Tomcat, seems to rely on getters to determine property types.)

Now we must add a tag entry to our TLD describing the new tag, and specifying that it requires an attribute, `name`:

```
<tag>
  <name>helloAttrib</name>
  <tagclass>tagext.AttribHelloTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>Simple example with attributes</info>
  <attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

The JSP container will throw an exception if the required `name` attribute is not specified, and the attribute can be set with the runtime value of an expression, as well as with a static string (the value of `rtexprvalue` being `true`).

The calling JSP, `hello2.jsp`, is identical to `hello.jsp`, except for the way in which we invoke the tag itself:

```
<%@ taglib uri="/hello" prefix="examples" %>


<html>
  <head>
    <title>Simple tag with attributes</title>
  </head>
  <body>
    This is static output.
  <p />
  <i>
    <examples:helloAttrib name="Rod">
    </examples:helloAttrib>
  </i>

    This is static output again.
  </body>
</html>
```

Attributes are an excellent way of controlling tag behavior at runtime, and are especially valuable in ensuring that tags are generic and reusable.

Elegant as the attribute/property mechanism is, there is one annoying problem with passing `String` attributes to tags. Specifying some characters in attributes is messy. The double quote character, for example, is (for obvious reasons) illegal in an attribute, and we must use the entity reference `"` if we want to include it. This rapidly becomes unreadable if the data includes multiple quotation marks. Attributes are also unsuited to handling lengthy values, for reasons of readability.

So there are limits to what can sensibly be achieved with attributes. Where complicated markup is concerned, consider the alternatives:

- Processing markup and expressions in the body of the tag, possibly repeatedly
- Defining a subtag that configures its ancestor. This is an advanced strategy, which we'll look at later in the following chapter
- Implementing the tag to read its markup from a template file or URL

The most elegant of these solutions, where feasible, is to manipulate the tag body. This will only be useful if the tag defines scripting variables that the tag body can use.

> *There is a curious and confusing inconsistency in JSP syntax when non-`String` tag attributes are the results of JSP expressions. Let's suppose we want to pass an object of class `examples.Values` (a kind of list) to a tag extension. The syntax:*
>
> ```
> <wrox:list values="<%=values%>"
> ```
>
> *is problematic, because we know from the JSP specification that an expression "is evaluated and the result is coerced to a `String` which is subsequently emitted into the current `out JspWriter` object". In the case of the custom tag above, however, the value of the expression is* not *coerced to a `String`, but passed to the tag handler as its original type.*

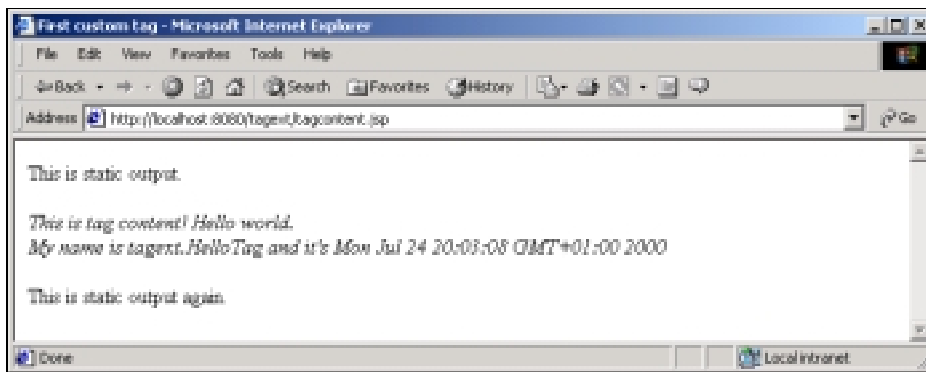
Body Content

So far we have used tags only to generate markup, we haven't considered what custom tags may do to their body content (anything found between their start and end tags).

Let's suppose we modify `hello.jsp` to add some code inside one of the tags, like this:

```
<examples:hello>
  This is tag content!
</examples:hello>
```

The result, shown by `tagcontent.jsp`, is that the content we have added is output in addition to any output generated by the tag. Whether the content appears before or after the tag's output depends on whether we chose to do tag output in the `doStartTag()` or `doEndTag()` methods. In the case of the `HelloTag`, the output will appear *before* the tag's own output, as shown in `tagcontent.jsp`:



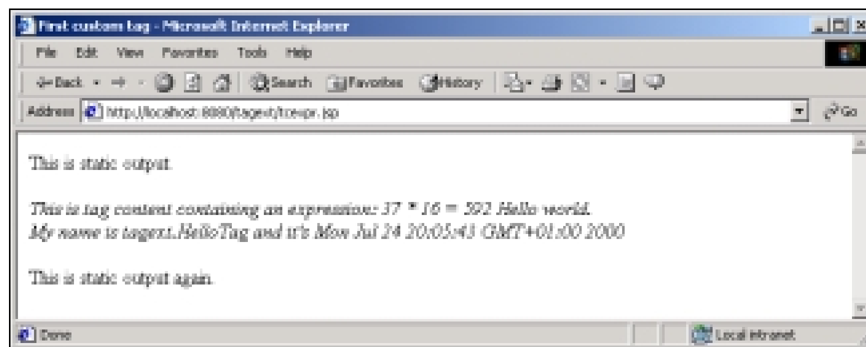
Tag content will be treated at run time as JSP (so long as the body content is set in the TLD to be JSP), so it may include expressions. Try modifying the last example like this:

```
<examples:hello>
This is tag content containing an expression: 37 * 16 = <%= 37 * 16 %>
</examples:hello>
```

Scriptlets are also legal, as is any other valid JSP content. The following will produce the same result:

```
<examples:hello>
<% int a = 37; %>
<% int b = 16; %>
This is tag content containing expressions and scriptlets:
       <%=a%> * <%=b%> = <%= a * b %>
</examples:hello>
```

The output in either case as shown in `tcexpr.jsp` is:



To do anything really useful with body content, however, we need to make our tags define scripting variables, or manipulate body content.

Tags Introducing Scripting Variables

We don't always want tags to produce output themselves. It's very useful for custom tags to introduce new scripting variables into the page, allowing calling JSPs to control the presentation without performing the processing involved in determining the content. Accordingly, there is a mechanism to allow tag extensions to define scripting variables.

Three steps are required to introduce scripting variables in a custom tag:

- Specify a `TagExtraInfo` class in the tag's entry in the TLD
- Implement the `TagExtraInfo` class to define the names and types of the variables
- Write code to add the variables to the `PageContext` in the tag handler itself

Let's look at these steps in turn.

Specifying a TagExtraInfo Class

As we have seen, a `TagExtraInfo` class can be associated with the tag by adding a `<teiclass>` element to the appropriate `<tag>` element in the TLD. It is good practice to follow the simple convention of naming your `TagExtraInfo` subclass for tag `xxxxTag` as `xxxxTagExtraInfo`. (Some developers prefer the shorter, but less self-documenting, form `xxxxTEI`.)

Implementing the TagExtraInfo Class

The implementation of the `TagExtraInfo` class is straightforward, once we understand the concepts involved. We will need to override the `getVariableInfo()` method for this:

```
VariableInfo[] getVariableInfo(TagData td)
```

The key to understanding how the scripting variable mechanism works is the `VariableInfo` class. We are only interested in the single constructor:

```
VariableInfo(String varName, String className, boolean declare, int scope);
```

Let us examine each parameter in turn:

- **Variable Name** (`varName`) is the name by which the scripting variable will be accessed in the JSP. It must be a legal Java identifier.
- **Class Name** (`className`) should be the fully qualified name of the variable's type. In practice, it will not be tested by the JSP engine at translation time, but used directly in code generation. For example, if `String` is specified for variable `name`, the JSP engine will generate a declaration such as `String name;` in the Java class representation of the JSP page.
- **Declare** (`declare`) is a `boolean` parameter that controls whether a *new* variable is to be created, or whether the tag will simply update the value of a variable already in the calling JSP page's `PageContext`. It is generally better practice to create new variables, although this may cause a translation-time error if the JSP has already used the variable name. Name conflicts can also arise if the same tag is nested, and descendants try to create new variables with the same name while an earlier one is still in scope. If the `declare` parameter is `true`, and a new variable is to be created, the JSP engine can generate Java code to declare a variable in the same way as a scriptlet may declare a variable: in the `_jspService()` method. In either case, the JSP will obtain the value for the variable set by the tag handler by looking in the `PageContext`.
- **Variable Scope** (`scope`). There are three types of scope defined for variables introduced within custom tags: `NESTED`, `AT_BEGIN`, and `AT_END`. If `NESTED` scope is specified, the variables are available to the calling JSP only within the body of the defining tag. (They will remain visible even if other tags are invoked within the defining tag.) If `AT_BEGIN` scope is specified, the variables will be available to the remainder of the calling JSP after the start of the defining tag. If `AT_END` scope is specified, the variables will be available to the remainder of the calling JSP after the end of the defining tag. Unless this is a strong reason for using the variables after the tag has been closed, the preferred scope is `NESTED`. In JSP pages, as in programming generally, additional variables introduce complexity.

Changes to the Tag Handler

We must not forget to modify the tag handler itself. Before they can be available to JSPs using the tag, variables must be added to the `PageContext` like this:

```
pageContext.setAttribute("variableName", myObject);
```

An Example

Suppose we've decided we'd like our `Hello` tag to be more configurable. Having "Hello" and other English text hard coded makes it little use in other language environments. Suppose we decide to use the tag to provide all the dynamic values we've seen it output (name, class name and date), but control the presentation entirely in the JSP.

First, we need to write a `TagExtraInfo` implementation to define the scripting variables; save this in the usual directory:

```
package tagext;

import javax.servlet.jsp.tagext.*;

// Class defining variables available to JSPs using VarHelloTag.
public class VarHelloTagExtraInfo extends TagExtraInfo {

  public VariableInfo[] getVariableInfo(TagData data) {
    return new VariableInfo[] {

      // The use of NESTED scope means that these scripting variables
      // will only be available inside the VarHelloTag.
      new VariableInfo("name", "java.lang.String", true, VariableInfo.NESTED),
      new VariableInfo("className", "java.lang.String", true,
                       VariableInfo.NESTED),
      new VariableInfo("date", "java.util.Date", true, VariableInfo.NESTED)
    };
  }
}
```

The tag handler itself is very simple, but differs from those we've seen so far in that it generates no markup. All it does is add values for the variables defined in the `TagExtraData` class to the `PageContext` in its `doStartTag()` method. (As these variables are only available within the tag, it would be useless to add them in the `doEndTag()` method.)

```
package tagext;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// Simple tag handler with one attribute.
//  Generates no markup, but defines three scripting
// variables: name, className and date.
```

```
// @see VarHelloBodyTagExtraInfo
public class VarHelloTag extends TagSupport {
  private String  name;

  public String getName() {
    return name;
  }

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

  public int doStartTag() throws JspTagException {
    // Make the variables available to calling JSPs
    pageContext.setAttribute("name", name);
    pageContext.setAttribute("className", getClass().getName());
    pageContext.setAttribute("date", new Date());
    return EVAL_BODY_INCLUDE;
  }
}
```

The TLD entry is similar to those we have seen: only the addition of the `<teiclass>` element specifying the `TagExtraInfo` class associated with the tag handler is required to make the scripting variables available to calling pages:

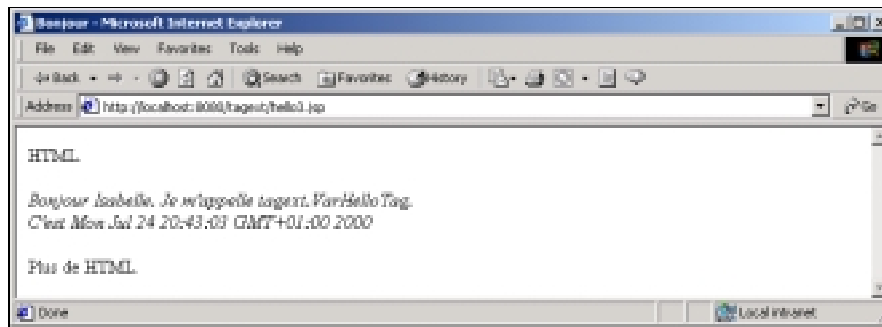
```
<tag>
  <name>helloVars</name>
  <tagclass>tagext.VarHelloTag</tagclass>
  <teiclass>tagext.VarHelloTagExtraInfo</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>Simple example defining scripting variables</info>
  <attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

The calling JSP, `hello3.jsp`, can now do all the work of rendering the output:

```
<%@ taglib uri="/hello.tld" prefix="examples" %>

<html>
  <head>
    <title>Bonjour</title>
  </head>
  <body>

    HTML.
  <p/>

  <i>
```

```
      <examples:helloVars name="Isabelle">
        Bonjour <%=name%>. Je m'appelle <%=className%>.<br/>
        C'est <%=date%><p/>
      </examples:helloVars>
    </i>

      Plus de HTML.

    </body>
  </html>
```

The screen output will be:



Note that we don't *need* to use the scripting variables we've created. The contents of the `examples:helloVars` tag could be static, in which case they it would be output unchanged, or the tag could be empty, in which case it would produce no HTML output at all.

> **Note that because the scripting variables introduced by tags must be placed in the `PageContext` by the JSP engine, only object variables can be created. This is a minor annoyance when we would really like a primitive type. Also remember that, even if they are never accessed, the values of all the scripting variables must still be computed and placed in the `PageContext`.**

Body Tags

What we've done so far with body content and scripting variables, using the `Tag` interface, is all very fine, but it still doesn't help us to do anything really exciting with a tag's body content. What if we want to suppress the content under some circumstances, filter it, or repeat it a number of times?

To do this, we need a richer API. Remember the `BodyTag` interface? The two methods we need are `doInitBody()` and `doAfterBody()`. `doAfterBody()` is especially useful, as it enables us to decide after each time the tag's body content has been processed whether to continue processing it, or move onto the end tag.

Body Tags and Iteration

One of the most common uses of body tags is to handle iteration. Control flow is handled much more cleanly in Java classes (such as tag handlers) than in JSPs, so placing iteration in custom tags can improve JSPs dramatically.

Let's modify our `VarHelloTag` to say hello to a number of people, using the `name` variable, and make the other scripting variables we've already used, `className` and `date`, available only *after* the looping of the tag has been completed.

We'll need to change the property passed in from a single `String` (`name`) to an indexed type. I've used `java.util.List`. We'll also need an instance variable to control our iteration over the list. (I could have obtained an `Iterator` from the `List`, but I also want to make the current position in the list available as a scripting variable.) I've added an `index` variable for this.

Handling the output of this tag requires a more complex implementation than we've seen before. I've used a `StringBuffer` to hold the output until we're ready to write it out, after list processing is complete.

The iteration is handled by the `doAfterBody()` method, which returns `EVAL_BODY_TAG` until the list has been exhausted. With each iteration, the value in the name variable is reset. At runtime, the body content of this tag will be evaluated for each element in the list, with the variable's value always up to date. The `doEndTag()` method is implemented to output the content we've built up in the `StringBuffer`:

```
package tagext;

import java.io.IOException;
import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
*  Body tag taking a list of names and iterating over its
*  content for each name.
*  No content will be generate for each iteration; however
*  the scripting variable <i>name</i> will be set to each
*  name in turn. After the close of this tag, scripting
*  variables <i>className</i> and <i>date</i> will be available
*  to JSPs using it.
*/
public class VarHelloBodyTag extends BodyTagSupport {
  // List of names passed in
  private List      names;

  // Where we're up to in iterating over the body content
  private int        index;

  // Output we're building up while iterating over the body content
  private StringBuffer  output = new StringBuffer();


  //  Getter for the names property/attribute
  public List getNames() {
    return names;
  }
```

```
//  Setter for the names property/attribute
public void setNames(List names) {
  this.names = names;
}

  public int doStartTag() throws JspTagException {
  if (names.size() > 0 ) {
    setLoopVariables();
    return EVAL_BODY_TAG;
  }
  // If we get here, we have an empty list and this tag should
  // ignore any body content
  return SKIP_BODY;
  }

/**
*  The JSP engine will call this method each time the body
*  content of this tag has been processed. If it returns
*  SKIP_BODY, the body content will have been processed for the
*  last time. If it returns EVAL_BODY_TAG, the body will be processed
*  and this method called at least once more.
*  <p/>We store content in a StringBuffer, rather than write
*  output directly.
*/
public int doAfterBody() throws JspTagException {
  BodyContent bodyContent = getBodyContent();
  if (bodyContent != null) {
    output.append(bodyContent.getString());
    try {
      bodyContent.clear();
    }
    catch (IOException ex) {
      throw new JspTagException("Fatal IO error");
    }
  }

  // If we still haven't got to the end of the list,
  // continue processing
  if (++index < names.size()) {
    setLoopVariables();
    return EVAL_BODY_TAG;
  }
  // If we get to here, we've finished processing the list
  return SKIP_BODY;
}

/**
*  Called after processing of body content is complete.
*  We use it to output the content we built up during processing
*  of the body content.
*/
public int doEndTag() throws JspTagException {
  try {
    bodyContent.getEnclosingWriter().write(output.toString());
```

```
      }
      catch (IOException ex) {
        throw new JspTagException("Fatal IO error");
      }

      // We've finished processing.
      // Set variables for the rest of the page
      pageContext.setAttribute("className", getClass().getName());
      pageContext.setAttribute("date", new Date());

      // Process the rest of the page
      return EVAL_PAGE;
    }

    //  Make variable available for each iteration
    private void setLoopVariables() {
      pageContext.setAttribute("name", names.get(index).toString());
      pageContext.setAttribute("index", new Integer(index));
    }

  }
```

Note the call to `getBodyContent()`, to obtain the JSP content generated by each pass over the tag's body content.

This tag requires an associated `TagExtraInfo` class:

```
package tagext;

import javax.servlet.jsp.tagext.*;

//  Variable information for the VarHelloBodyTag.
public class VarHelloBodyTagExtraInfo extends TagExtraInfo {

  //  Return an array of variables set by the VarHelloBodyTag.
  public VariableInfo[] getVariableInfo(TagData data) {
    return new VariableInfo[] {
      new VariableInfo("name", "java.lang.String", true, VariableInfo.NESTED),
      new VariableInfo("index", "java.lang.Integer", true,
                       VariableInfo.NESTED),
      new VariableInfo("className", "java.lang.String", true,
                       VariableInfo.AT_END),
      new VariableInfo("date", "java.util.Date", true, VariableInfo.AT_END)
    };
  }

}
```

Note the two scopes used, to distinguish between the loop variables and the variables to be made available after the tag has been processed.

The tag library entry is:

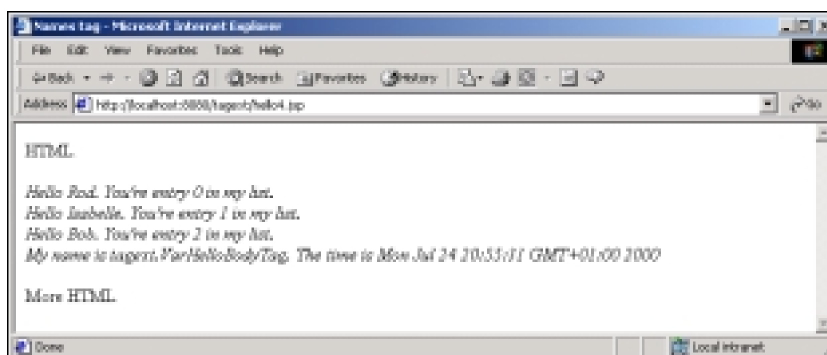
```
<tag>
  <name>hellos</name>
  <tagclass>tagext.VarHelloBodyTag</tagclass>
  <teiclass>tagext.VarHelloBodyTagExtraInfo</teiclass>
  <bodycontent>JSP</bodycontent>
```

```
    <info>Simple iterative example</info>
    <attribute>
      <name>names</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
```

In the calling JSP page I've defined a `List` and added a few elements to give the tag something to display:

```
  <%@ taglib uri="/hello" prefix="examples" %>

  <%
    // Normally we don't declare variables, in JSPs,
    // but this example should be self-contained
    java.util.List names = new java.util.LinkedList();
    names.add("Rod");
    names.add("Isabelle");
    names.add("Bob");
  %>

  <html>
    <head>
      <title>Names tag</title>
    </head>
    <body>
      HTML.
      <p />

      <i>
        <examples:hellos names="<%=names%>">
          Hello <%=name%>. You're entry <%=index%> in my list.<br/>
        </examples:hellos>
        My name is <%=className%>.
        The time is <%=date%><p/>
      </i>

      More HTML.

    </body>
  </html>
```

The output will be:



Although we developed it from our previous examples, this is close to a generic solution for list iteration, isn't it? It conceals the looping from the JSP and makes the successive list values and indices available. With minor changes, this class could take a `Collection`, or a Swing `ListModel`, and make it available to any JSP.

The main reason that this tag is so generic is that it doesn't generate markup. The more the JSP can control its output, the more useful a tag extension is.

Body Tags That Filter Their Content

Another idiomatic use of body tags is to perform filtering or other processing on their body content. This could be a simple text transformation, or could even interpret the tag's content as a custom language. The following simple example takes the tag's body content and writes it, reversed, into the calling JSP page.

Implementing the reversal is trivial; all we need to do is obtain the body content as a `String`, use it to initialize a `StringBuffer`, and call the `StringBuffer`'s `reverse()` method before writing out the resulting `String`:

```
package tagext;

import java.io.IOException;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

//  Simple body tag to reverse its content
public class ReverseTag extends BodyTagSupport {

  /*
  *  Called after processing of body content is complete.
  *  We use it to obtain the tag's body content and write
  *  it out reversed.
  */
  public int doEndTag() throws JspTagException {
  BodyContent bodyContent = getBodyContent();
    // Do nothing if there was no body content
    if (bodyContent != null) {
      StringBuffer output = new StringBuffer(bodyContent.getString());
      output.reverse();
      try {
        bodyContent.getEnclosingWriter().write(output.toString());
      }
      catch (IOException ex) {
        throw new JspTagException("Fatal IO error");
      }
    }

    // Process the rest of the page
    return EVAL_PAGE;
  }
}
```

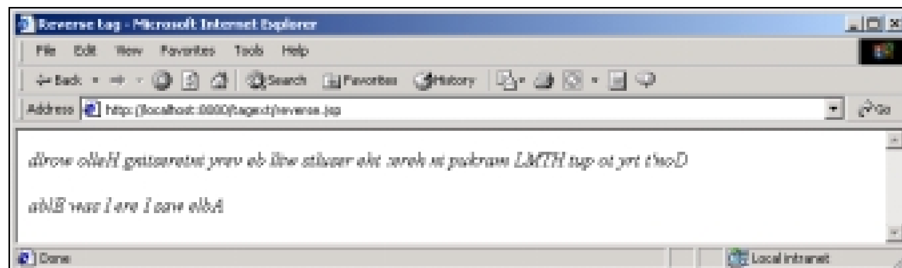
The tag library entry is also very simple. There are no variables requiring a `TagExtraInfo` class, and no attributes:

```
<tag>
  <name>reverse</name>
  <tagclass>tagext.ReverseTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>Simple example</info>
</tag>
```

A simple test JSP, `reverse.jsp`:

```
<%@ taglib uri="/hello" prefix="examples" %>

<html>
  <head>
    <title>Reverse tag</title>
  </head>
  <body>
    <i>
      <examples:reverse>
        Don't try to put HTML markup in here:
        the results will be very interesting
        Hello world
      </examples:reverse>

      <p />

      <examples:reverse>
        Able was I ere I saw Elba
      </examples:reverse>
    </i>
  </body>
</html>
```

This should produce the following output:



The result of placing HTML markup in the tag body will be unpredictable; hence I've used the tag twice so that I can introduce some formatting markup between the tag instances.

There are many useful applications for filtering body tags. One, which will be discussed in the next chapter, is performing an XSLT transform on XML content.

Tag Nesting

One might expect that specifying tag nesting of custom tags would be done in the TLD. However, TLDs don't allow for this, and nesting must be implemented by cooperating tag handler classes. Fortunately, the API helps us in this respect by providing methods on tag handlers we can use to obtain information about their parents and other ancestors. Although programmers of tag handlers must ensure that they enforce their desired tag nesting, the dynamic discovery of tag nesting at runtime allows for greater flexibility than would be possible if nesting were mandated in a static file. However, the absence of a formal grammar like a DTD or XML schema does place a responsibility on developers to ensure that any cooperation between tags is well documented.

Why might we use tag nesting? A common reason is to handle iteration (nested tags can simulate nested loops). Another is to let nested tags benefit from the context of the enclosing tag or tags.

Suppose we have additional information we want to display for the people named in our list, but that this information is expensive to retrieve from a database or legacy system. So we don't want the `hellos` tag to retrieve these additional fields with every iteration of the loop. (This need to retrieve regardless of usage is a disadvantage of using scripting variables.)

One solution is to use a descendant tag that draws its context from the enclosing tag and performs the additional lookups only when required: that is, only when the descendant tag is used. Let's implement a `NameTag` that requires no attributes, but retrieves additional information for the user its parent is currently processing. This information, nationality and city, will be exposed through scripting variables. Note that the child tag, like any body content, will be evaluated each time the parent iterates over its body content.

The JSP code invoking this functionality might look like this (using a scriptlet to define and display the extra information only when desired):

```
<examples:hellos names="<%=names%>" >
  Hello <%=name%>. You're entry <%=index%> in my list.
  <% if (condition) { %>
    <examples:nameInfo>
      <b>Nationality:</b> <%=nationality%> <b>City:</b> <%=city%>
    </examples:nameInfo>
  <% } %>
  <br/>
</examples:hellos>
```

To implement this, we'll first need to add a method to `VarHelloBodyTag` that exposes the necessary context, `String getName()`.

While we're at it, we'll create an interface `NameContext` that contains this new method. This way, we could make `VarHelloTag`, which already defines a `getName()` method, implement the interface and provide the necessary context for our new subtags.

> **With tag extensions, as always, remember to program to interfaces rather than concrete classes.**

The `NameContext` interface is trivial:

```
package tagext;

// Interface to provide context for nested tags
public interface NameContext {
  String getName();
}
```

So is the modification to `VarHelloBodyTag` (beyond making it implement `NameContext`) – simply the implementation of the new method:

```
//  Method from NameContext interface required to provide context to
// nested tags
public String getName() {
  return names.get(index).toString();
}
```

In a real application, we'd do some error checking here.

Now let's look at the new `NameTag` tag handler. Its main tasks are to obtain the context from an enclosing tag, and to retrieve the additional information. Note how it enforces correct nesting. There is a `getParent()` method in the `Tag` interface, but it is usually preferable to use `findAncestorWithClass()`. We don't want to limit the context in which we can use our tags. Can we guarantee that another tag or tags might not stand in the hierarchy between the two cooperating tags? If one or more do, and we have hard coded reliance on a particular parent tag class or interface, the nested tag will fail to find the ancestor it requires to provide its context.

For simplicity, I've hard coded the additional data in the class, in a hash table. For the sake of the example, imagine that this data is actually very expensive to retrieve. The extra information about people in the list will be their nationality, and the city they currently live in. My friends tend to travel quite a bit so these pieces of information may not be obviously related:

```
package tagext;

import java.io.IOException;
import java.util.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

/**
*  Tag nested within a tag implementing the NameContext interface
*  to provide additional information about the relevant person.
*  This information will be exposed through the <i>nationality</i>
*  and <i>city</i> scripting variables.
*  <br/>This tag requires no attribute, as it initializes itself from
*  the appropriate ancestor tag.
*  <p/>This tag produces no markup.
*/
public class NameTag extends TagSupport {
  // Data store
  private HashMap  infoHash = new HashMap();
```

```
//  Populate the data store. In a real application, this data would
//  be sourced from a database or another part of the application.
public NameTag() {
  infoHash.put("Rod", new PersonalInfo("Australian", "London"));
  infoHash.put("Isabelle", new PersonalInfo("French", "Gabon"));
  infoHash.put("Bob", new PersonalInfo("Australian", "Sydney"));
}

public int doStartTag() throws JspTagException {
  String nationality = "Unknown";
  String city = "Unknown";

  // Test whether this tag has an ancestor of the required type,
  // which we can use to obtain a name to lookup.
  // Note that using the findAncestorWithClass static method is more
  // flexible than using getParent(). getParent() will fail if
  // one or more tags separate this tag from the desired tag in
  // the runtime hierarchy of tag handlers.
  NameContext nameContextAncestor = (NameContext)
          TagSupport.findAncestorWithClass(this, NameContext.class);

  // The exception thrown here will be handled by the JSP engine as normal.
  // This will normally mean redirection to an error page.
  if (nameContextAncestor == null) {
    throw new JspTagException
          ("NameTag must only be used within a NameContext tag");
  }

  // If we get here, we have a valid ancestor from which we can obtain
  // a context.
  String name = nameContextAncestor.getName();
  PersonalInfo pi = (PersonalInfo) infoHash.get(name);
  if (pi != null) {
    nationality = pi.getNationality();
    city = pi.getCity();
  }
  pageContext.setAttribute("nationality", nationality);
  pageContext.setAttribute("city", city);
  return EVAL_BODY_INCLUDE;
}


// Inner class containing additional data retrieved for each name
private class PersonalInfo {
  private String  nationality;
  private String   city;

  public PersonalInfo(String nationality, String city) {
    this.nationality = nationality;
    this.city = city;
  }

  public String getNationality() {
    return nationality;
  }
```

```
    public String getCity() {
      return city;
    }
  }

}
```

The `NameTag` class will require a simple `NameTagExtraInfo` class, which should contain no surprises. It simply publishes variables not published by the enclosing `VarHelloBodyTag`. Note that these scripting variables will only be available within the name tag itself, although those published by the enclosing tag will still be visible:

```
package tagext;

import javax.servlet.jsp.tagext.*;

//  Variable information for the NameTag.
//   @author  Rod Johnson
public class NameTagExtraInfo extends TagExtraInfo {

  //  Return an array of variables set by the VarHelloBodyTag.
  public VariableInfo[] getVariableInfo(TagData data) {
    return new VariableInfo[] {
      new VariableInfo("nationality", "java.lang.String", true,
                       VariableInfo.NESTED),
      new VariableInfo("city", "java.lang.String", true,
                       VariableInfo.NESTED),
    };
  }
}
```

We'll also need a new entry in our TLD file:

```
<tag>
  <name>nameInfo</name>
  <tagclass>tagext.NameTag</tagclass>
  <teiclass>tagext.NameTagExtraInfo</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>Simple example of tag nesting</info>
</tag>
```

Let's now create a JSP to use the new tag, so that the additional lookup is performed part of the time. Of course normally application logic would determine this, but for the sake of the example we will simply decide this randomly.

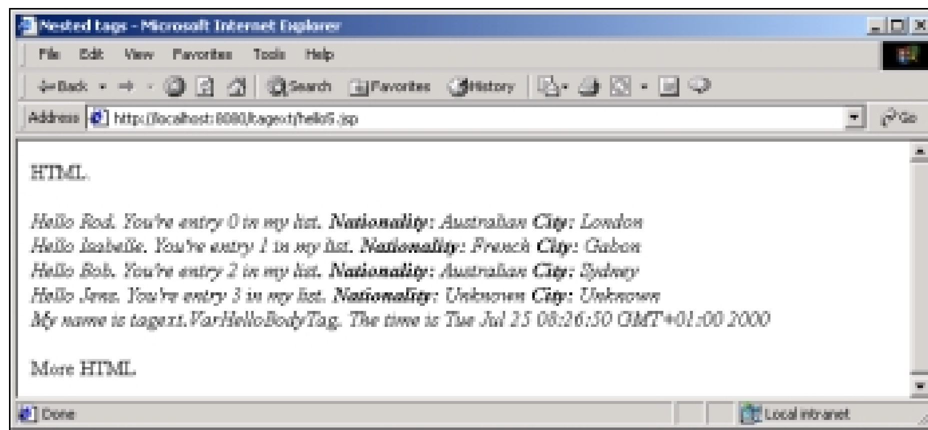
The JSP, `hello5.jsp`, is very similar to the previous example:

```
<%@ taglib uri="/hello" prefix="examples" %>

<%
  // Normally we don't declare variables, in JSPs,
  // but this example should be self-contained
  java.util.List names = new java.util.LinkedList();
  names.add("Rod");
  names.add("Isabelle");
  names.add("Bob");
  names.add("Jens");
```

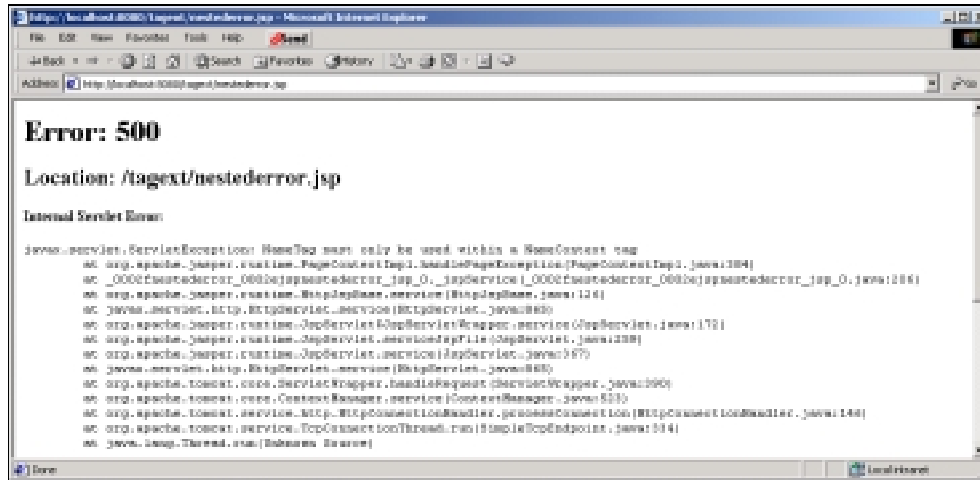
```
%>

<html>
  <head>
    <title>Nested tags</title>
  </head>
  <body>
    HTML.
    <p/>

    <i>
      <% java.util.Random rand = new java.util.Random(); %>
      <examples:hellos names="<%=names%>" >
        Hello <%=name%>. You're entry <%=index%> in my list.
        <% if (rand.nextInt(3) != 0) { %>
            <examples:nameInfo>
              <b>Nationality:</b> <%=nationality%> <b>City:</b> <%=city%>
            </examples:nameInfo>
        <% } %>
        <br/>
      </examples:hellos>
      My name is <%=className%>.
      The time is <%=date%><p/>
    </i>
      More HTML
  </body>
</html>
```

I've also added a new entry, Jens, to the list of names, to show what happens when no extra information is available. When I wrote the `NameTag` class I decided to handle this situation gracefully, as it didn't justify throwing an exception that would abort the rendering of any JSP using the tag. Instead, the placeholder value `"Unknown"` is placed in the `PageContext`. Note the use of a scriptlet inside the `<examples:hellos>` tag to perform the necessary conditional logic. This doesn't produce any output, and is evaluated each time the `<example:hellos>` tag processes its body content. Only if the condition is true – this is quasi-random in this example – will the subtag be evaluated, its variables calculated and added to the `PageContext` and its body content evaluated.

The output should look like this:



Try changing the example to move the `<examples:nameInfo>` tag outside the scope of a `<examples:hellos>` tag. Your JSP engine should provide a helpful error message, including the wording of the exception we made the `NameTag` throw in this case. The following is the page displayed by Tomcat 3.1:



Once again we have seen the power of the ability to define scripting variables. Because the implementations of the tags don't contain any markup, we could easily use these tags to generate a content type other than HTML.

Handling Errors

What should tag handlers do if they encounter an error condition? The answer depends on whether the error is serious enough to invalidate the work of the calling JSP, and whether the tag is likely to be important enough to justify causing the calling page to redirect to an error page.

If the error is minor, the best solution is to output suitable error markup, or nothing at all, depending on the tag's purpose. An HTML comment may be added explaining the error in more detail. If the error is major, the best approach is to make the tag handler method that detects the problem throw a `JspTagException`. This will, in most cases, cause the calling page to redirect to an error page.

Tag Extension Idioms

In order to stress the mechanics of tag handlers themselves, I have deliberately used simple examples to this point. However, the range of functionality that can be delivered with tag extensions is very wide.

Some of the most important possibilities are:

- **Generating HTML output**
  This is the simplest use of custom tags. It has some merit in that a standard building block is available, and can be changed simultaneously everywhere it occurs if necessary. However, in general, using Java code to generate HTML is clumsy and inflexible.

- **Using template content to generate HTML output**
  A more sophisticated variant of the above, avoiding having messy markup generation in Java code and allowing modification of the generated markup without recompilation. Usually a mechanism will need to be designed and documented for controlling variable substitution and any other ways in which the template is made dynamic. An abstract class extending `TagSupport` or `BodyTagSupport` could provide standard template lookup and variable interpolation for use across a system. This is a common idiom, and can be very useful in real-world systems.

- **Defining scripting variables**
  This is often a superior alternative to using tags to generate markup directly, especially when iteration is concerned. All computations and control flow is controlled by the tag handler classes, rather than the calling JSP, but the scripting variables are used within the JSP to allow markup generation to be changed easily. As a general rule, tags that define variables should not generate markup.

- **Transforming or interpreting tag body content**
  This is an especially powerful use of tag extensions, with virtually unlimited potential. For example, a tag could establish a database connection and execute SQL contained in its body to display the results. A tag could implement an interpreter for a language, providing entirely new syntax and functionality within JSPs. (For example, it would be conceivable to design an `asp` tag that would interpret a subset of ASP for running legacy code.) In some of these cases, the `tagdependent` value should be used to describe the body content in the TLD.

Note that some of the possibilities of this idiom are not compatible with writing maintainable code. JSP is a standard, understood by a whole development community, whereas your tag content may not be, unless you choose some other standard such as SQL. It is also possible to create nested tags that interact in surprising ways; this also reduces readability.

- **A gatekeeper role**
  A custom tag can check for login or some other condition, and redirect the response if the result is unsatisfactory.

- **Concealing access to enterprise objects or APIs that should not be visible from JSPs**
  This is an easy way of providing a JSP interface to enterprise data. However, note that it is not consistent with good J2EE design to write tags that access databases via JDBC. Consider the alternative of a true n-tier architecture, in which the tag handlers access business objects such as session EJBs.

- **Exposing complex data**
  Custom tags can be used to expose data (for example, in a list or table) that might otherwise require complicated JSP logic to display.

- **Handling iteration**
  This is a simple and practical way of avoiding a profusion of scriptlets in JSPs. In this case it is especially important to make tags as generic as possible.

Remember that custom tags are building blocks, and will be most useful when they can be reused easily. The following principles help to make tag extensions reusable:

- Make tags as configurable as possible. This can be achieved through tag attributes and using nesting to provide context. Optional attributes can be used where default values can be supplied.

- Avoid generating HTML in your tag handler unless absolutely necessary.
- When tags handlers *must* generate HTML, ensure that the generated HTML can be used in a wide variety of contexts: try to avoid <html>, <form>, and other structural tags. Consider reading the HTML from a template file.
- Avoid making custom tags do unexpected things to the `request` and `response`. Just because tag handlers can access these objects through the `PageContext` does not mean it's a good idea. For example, how obvious will it be to a reader of a JSP using a custom tag that the tag may redirect the response? Unless the tag's documentation is scrupulous, figuring this out might require plunging into the tag's Java implementation. Consider another example: if a tag were to flush the JSP's output buffer, the JSP engine would be unable to redirect to an error page if anything went wrong in the rendering of the rest of the JSP. (The page would be in an illegal state, attempting redirection after it had written to the HTTP response.) This would limit the usefulness of the tag and, again, the behavior would be a challenge for a JSP developer to figure out.

When and how to use tag extensions is further discussed in Chapter 14, Writing Maintainable JSP Pages. Remember, however, that some caution is called for in using custom tags. Using too many tag extensions can make JSPs unreadable: the end result will be your own language, which may be the most efficient approach to solving a particular problem, but won't be intelligible to an outside observer, especially if your tags cooperate in complex ways.

Summary

JSP 1.1 tag extensions, or custom tags, are a powerful extension to the JSP model. Their use is limited only by the ingenuity of developers, and they will become an essential building block of well-engineered JSP interfaces.

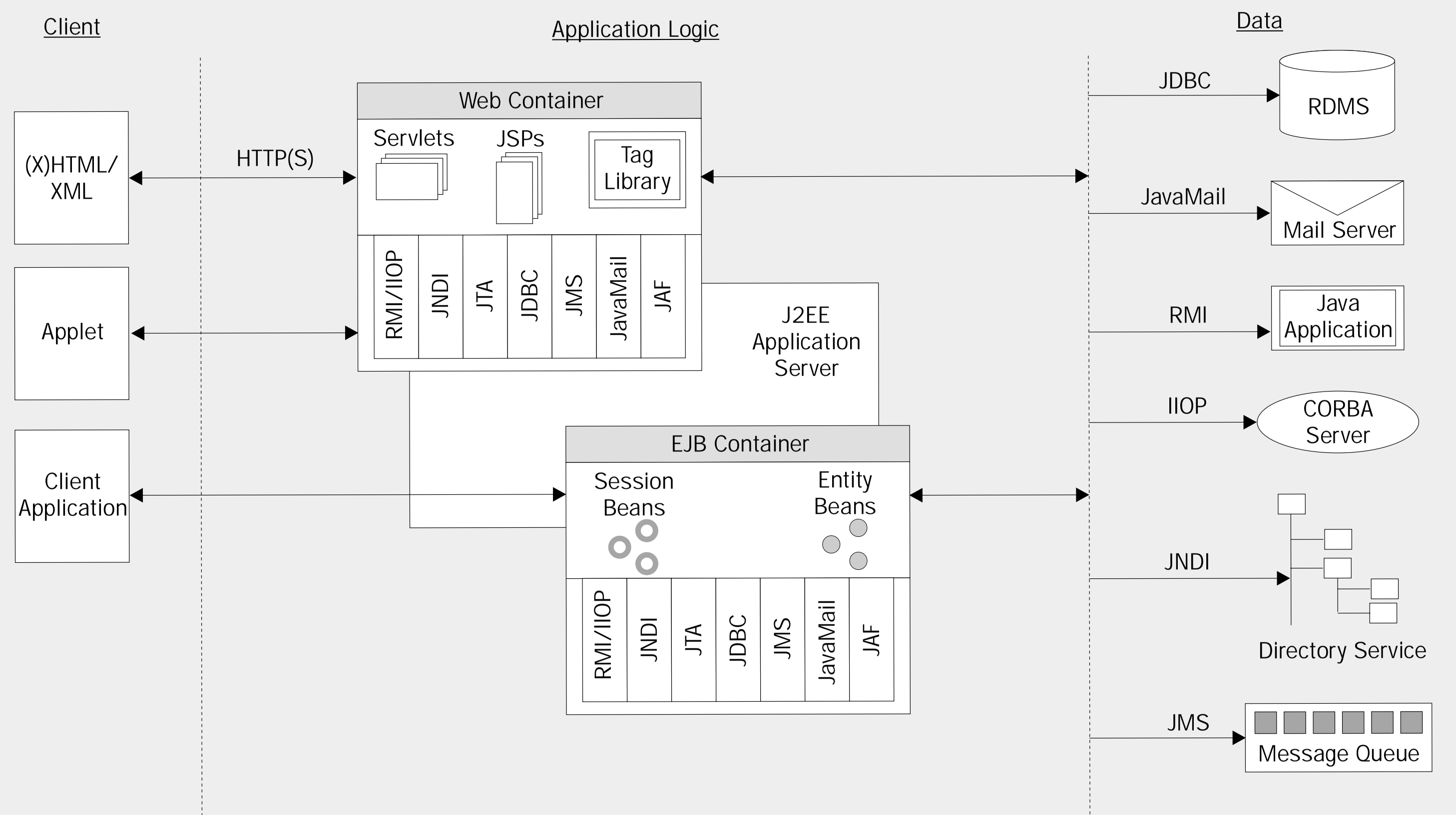
Tag extensions can access the JSP `PageContext`. Their behavior can respond dynamically to the XML attributes they are invoked with, and their body content. They are implemented using:

- Java classes implementing tag behavior
- XML Tag Library Descriptor (TLD) files describing one or more tags, and the attributes they require
- Optional extra classes defining scripting variables introduced into the page by the custom tag

Tag extensions are a valuable way of separating presentation from content in JSP interfaces. Since they are a standard part of JSP 1.1, libraries of third party tags can be developed, and will become increasingly valuable building blocks in JSP development.

Importantly, tag extensions are very easy to develop, once the initial concepts are grasped. The tag extension mechanism achieves the goals expressed in the JSP 1.1 specification, that it should be portable, simple, expressive, and built upon existing concepts and machinery.

In the next chapter we will go on to look at some more complex examples of tag extensions.


Client
Application Logic
Data
(X)HTML/
XML
Applet
Client
Application
HTTP(S)
Web Container
Servlets
JSPs
Tag
Library
RMI/IIOP
JNDI
JTA
JDBC
JMS
JavaMail
JAF
J2EE
Application
Server
EJB Container
Session
Beans
Entity
Beans
RMI/IIOP
JNDI
JTA
JDBC
JMS
JavaMail
JAF
JDBC
RDMS
JavaMail
Mail Server
RMI
Java
Application
IIOP
CORBA
Server
JNDI
Directory Service
JMS
Message Queue