JavaServer Pages(TM) Tutorial

Welcome to the JavaServer Pages^{TM} (JSP^{TM}) technology, the cross-platform method of generating dynamic content for the Web.

If you have reached this learn-by-example tutorial, you are probably new to the technology. You might be a Web developer or enterprise developer who wants to use JavaServer Pages to develop dynamic Web applications. The sections in this tutorial contain a series of topics and example applications that teach you how to use the essential features of JavaServer Pages technology:

Installing and Running the Example Applications

The example applications described in this tutorial are packaged so that they can be easily installed and run on the Tomcat JSP and server implementation. To run the examples:

1. Download and install Tomcat
2. Download the example applications

   The complete binary and source code for three of the examples are packaged in the Web application archives `helloworld.war`, `hellouser.war`, and `email.war`, contained in the zip archive `examples.zip`. To install the applications in Tomcat, download `examples.zip` into the directory *TOMCAT_HOME/*`webapps` and unzip the archive. The fourth example, `number guess`, is already installed in Tomcat in the directory *TOMCAT_HOME/*`webapps/examples/jsp/num`.

3. Configure Tomcat for the example applications

   When an archived Web application is accessed, Tomcat 3.2 automatically unpacks it into the directory *TOMCAT_HOME/*`webapps`*/appName* and adds the context for each archive to the server startup file. If you are using an earlier version of Tomcat you will need to:

   - Unpack the Web application archive with the command `jar xvf` *appName*`.war`.
   - Add the following line to the file *TOMCAT_HOME/*`conf/server.xml` for each application:

     ```
     <Context path="/appName" docBase="webapps/appName" debug="0"
     reloadable="true" />
     ```

   When a Web application archive is unpacked, its contents are deposited into the directories listed in the following table. This directory layout is required by the Java Servlet specification and is one that you usually will use while developing an application.

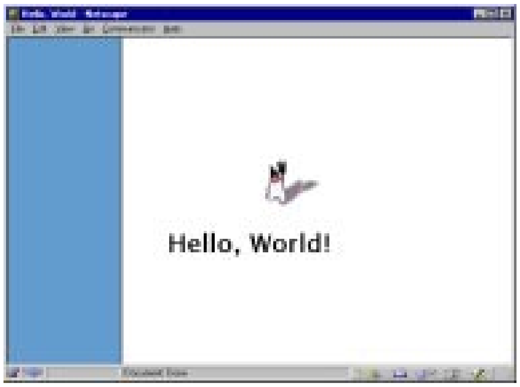
Directory	Contents
appName	JSP, HTML, and image files
appName/`WEB-INF/classes`	classes accessed by JSP files

4. Open the URL of the first page of each example in a Web browser:
   - `http://localhost:8080/helloworld/helloworld.jsp`
   - `http://localhost:8080/hellouser/hellouser.jsp`
   - `http://localhost:8080/examples/jsp/num/numguess.jsp`
   - `http://localhost:8080/email/email.jsp`

A First JSP Application

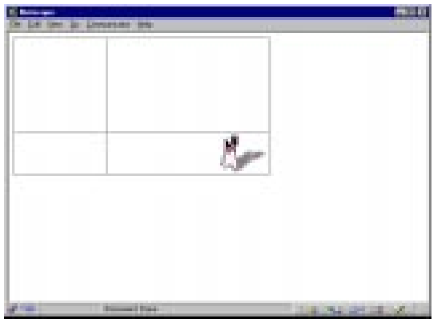
FIGURE 1-1 shows what is perhaps the simplest JSP application one could write. It continues the illustrious computer science *Hello, World* tradition. CODE EXAMPLE 1-1 and CODE EXAMPLE 1-2 show how the example is put together.

FIGURE 0-1 Duke Says Hello



CODE EXAMPLE 0-1 The Duke Banner *(dukebanner.html)*

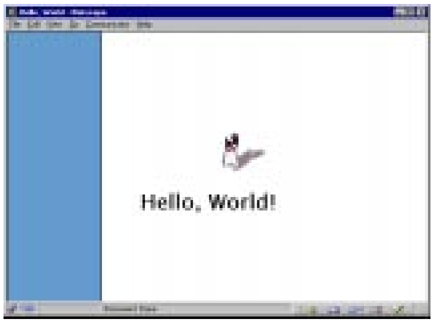
```
<table border="0" width="400" cellspacing="0" cellpadding="0">
<tr>
<td height="150" width="150"> &nbsp; </td>
<td width="250"> &nbsp; </td>
</tr>
```



```
<tr>
<td width="150"> &nbsp; </td>
<td align="right" width="250">
<img src="duke.waving.gif">
</td>
</tr>

</table>
<br>
```

CODE EXAMPLE 0-2 The JSP Page *(helloworld.jsp)*

```
<%@ page info="a hello world example" %>

<html>
<head><title>Hello, World</title></head>

<body bgcolor="#ffffff" background="background.gif">
<%@ include file="dukebanner.html" %>
```



```
<table>
<tr>
<td width=150> &nbsp; </td>
<td width=250 align=right>
<h1>Hello, World!</h1> </td>
</tr>
</table>

</body>
</html>
```

The Page Directive

The `page` directive is a JSP tag that you will use in almost every JSP source file you write. In *helloworld.jsp*, it's the line that looks like this:

```
<%@ page info="a hello world example" %>
```

The `page` directive gives instructions to the JSP container that apply to the entire JSP source file. In this example, `page` specifies an informative comment that will become part of the compiled JSP file. In other cases, `page` might specify the scripting language used in the JSP source file, packages the source file would import, or the error page called if an error or exception occurs.

You can use the `page` directive anywhere in the JSP file, but it's good coding style to place it at the top of the file. Because it's a JSP tag, you can even place it before the opening `<html>` tag.

The Include Directive

The `include` directive inserts the contents of another file in the main JSP file, where the directive is located. It's useful for including copyright information, scripting language files, or anything you might want to reuse in other applications. In this example, the included file is an HTML table that creates a graphic banner.

You can see the content of the included file by viewing the page source of the main JSP file while you are running *Hello, World*. The included file does not contain `<html>` or `<body>` tags, because these tags would conflict with the same tags in the calling JSP file.

A Note About the JSP Tags

As you use the examples in this chapter, remember that the JSP tags are case sensitive. If, for example, you type `<%@ Page %>`, instead of `<%@ page %>`, your tag will not be recognized, and the JSP implementation will throw an exception. Some of the attributes on the tags take class names, package names, pathnames or other case-sensitive values as well.

If you have any doubts about the correct spelling or syntax of any JSP tag, see the *JavaServer Pages Syntax Card*.

How To Run the Example

Install the example as described in “Installing and Running the Example Applications” on page 2. Then, open a Web browser and go to:

```
http://localhost:8080/helloworld/helloworld.jsp
```

Handling HTML Forms

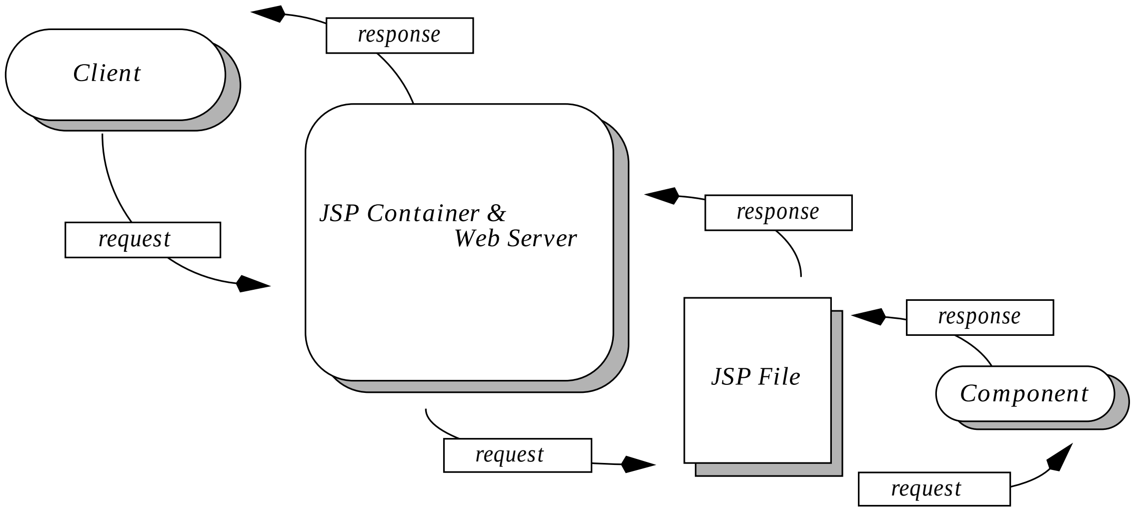
One of the most common parts of an electronic commerce application is an HTML form in which a user enters some information. The information might be a customer's name and address, a word or phrase entered for a search engine, or a set of preferences gathered as market research data.

What Happens to the Form Data

The information the user enters in the form is stored in the `request` object, which is sent from the client to the JSP container. What happens next?

FIGURE 1-2 represents how data flows between the client and the server (at least when you use Tomcat; other JSP containers may work a little differently).

FIGURE 0-2 How Data is Passed Between the Client and the Server



The JSP container sends the `request` object to whatever server-side component (JavaBeans^{TM} component, servlet, or enterprise bean) the JSP file specifies. The component handles the request, possibly retrieving data from a database or other data store, and passes a `response` object back to the JSP container. The JSP container passes the `response` object to the JSP page, where its data is formatted according the page’s HTML design. The JSP container and Web server then send the revised JSP page back to the client, where the user can view the results in the Web browser. The communications protocol used between the client and server can be HTTP, or it can be some other protocol.

The `request` and `response` objects are always implicitly available to you as you author JSP source files. The `request` object is discussed in more detail later in this tutorial.

How To Create a Form

You typically define an HTML form in a JSP source file, using JSP tags to pass data between the form and some type of server-side object (usually a bean). In general, you do the following things in your JSP application:

1. Start writing a JSP source file, creating an HTML form and giving each form element a name.
2. Write the bean in a `.java` file, defining properties, get, and set methods that correspond to the form element names (unless you want to set one property value at a time explicitly).
3. Return to the JSP source file. Add a `<jsp:useBean>` tag to create or locate an instance of the bean.
4. Add a `<jsp:setProperty>` tag to set properties in the bean from the HTML form (the bean needs a matching set method).
5. Add a `<jsp:getProperty>` tag to retrieve the data from the bean (the bean needs a matching get method).
6. If you need to do even more processing on the user data, use the `request` object from within a scriptlet.

The *Hello, User* example will make these steps more clear.

A Dynamic Hello Application

The *Hello, User* JSP application shown in FIGURE 1-3 and FIGURE 1-4 expands on the *Hello, World* application. The user has an opportunity to enter a name into a form and the JSP page generates a new page that displays the name.

FIGURE 0-3 The User Enters a Name

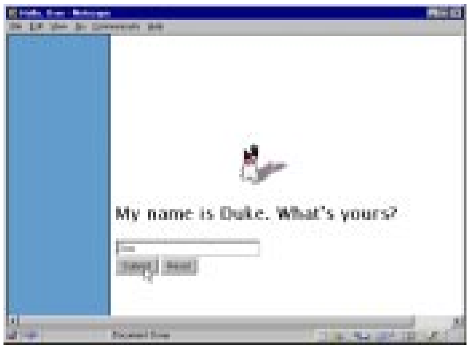
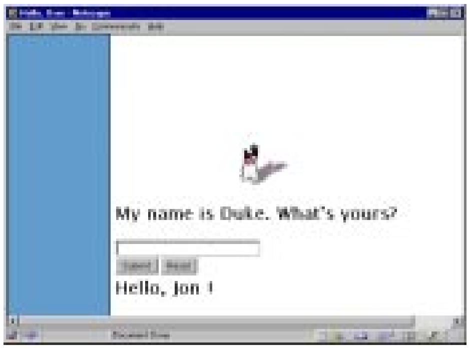


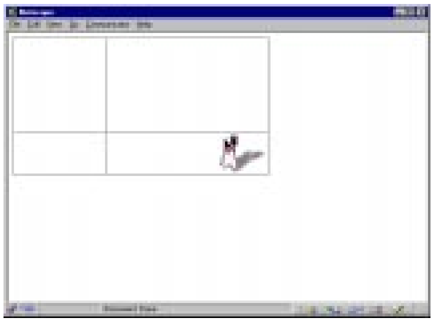
FIGURE 0-4 Then Duke Says Hello



Example Code

CODE EXAMPLE 1-3, CODE EXAMPLE 1-4, CODE EXAMPLE 1-5, and CODE EXAMPLE 1-6 contain the code for the Duke banner, main JSP page, response JSP page, and JavaBeans component that handles the name input.

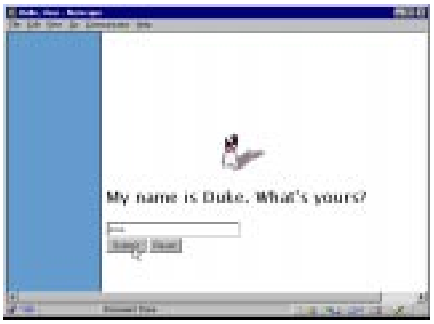
CODE EXAMPLE 0-3 The Duke Banner *(dukebanner.html)*



```
<table border="0" width="400" cellspacing="0" cellpadding="0">
<tr>
<td height="150" width="150"> &nbsp; </td>
<td width="250"> &nbsp; </td>
</tr>
<tr>
<td width="150"> &nbsp; </td>
<td align="right" width="250"> <img src="duke.waving.gif">
</td>
</tr>
</table>
<br>
```

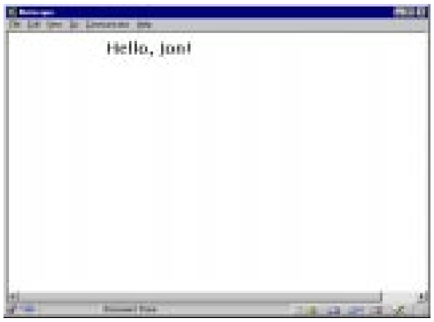
CODE EXAMPLE 0-4 The Main JSP File (*hellouser.jsp*)

```
<%@ page import="hello.NameHandler" %>

<jsp:useBean id="abean" scope="page" class="hello.NameHandler" />
<jsp:setProperty name="abean" property="*" />

<html>

<head><title>Hello, User</title></head>
<body bgcolor="#ffffff" background="background.gif">

<%@ include file="dukebanner.html" %>
```



```
<table border="0" width="700">
<tr>
<td width="150"> &nbsp; </td>
<td width="550">
<h1>My name is Duke. What’s yours?</h1>
</td>
</tr>

<tr>
<td width="150" &nbsp; </td>
<td width="550">
<form method="get">
<input type="text" name="username" size="25">
<br>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</td>
</tr>
</form>
</table>

<%
  if ( request.getParameter("username") != null ) {
%>
<%@ include file="response.jsp" %>

<%
  }
%>

</body>
</html>
```

CODE EXAMPLE 0-5 The Response File (*response.jsp*)



```
<table border="0" width="700">

<tr>
<td width="150">
&nbsp;
</td>

<td width="550">
<h1>Hello, <jsp:getProperty name="abean" property="username" />!
</h1>

</td>
</tr>
</table>
```

CODE EXAMPLE 0-6 The Bean That Handles the Form Data (*namehandler.java*)

```
package hello;

public class NameHandler {

   private String username;

   public NameHandler() {
      username = null;
   }

   public void setUsername( String name ) {
      username = name;
   }

   public String getUsername() {
      return username;
   }
}
```

Constructing the HTML Form

An HTML form has three main parts: the opening and closing `<form>` tags, the input elements, and the Submit button that sends the data to the server. In an ordinary HTML page, the opening `<form>` tag usually looks something like this:

```
<form method=get action=someURL>
```

In a JSP application, the `action` attribute specifies the component or JSP file that will receive the data the user enters in the form. You can omit the `action` attribute if you want the data processed by the object specified in the `<jsp:useBean>` tag. (This is similar to using `action` in other Web applications, where it specifies a CGI script or other program that will process the form data.)

The rest of the form is constructed just like a standard HTML form, with input elements, a Submit button, and perhaps a Reset button. Be sure to give each input element a name, like this:

```
<input type="text" name="username">
```

Using the GET and POST Methods

The HTTP GET and POST methods send data to the server. In a JSP application, GET and POST send data to the server. (The data, along with the rest of the JSP application is compiled into a Java servlet that returns a response to the client Web browser; for more information, see “How the JSP Page Is Compiled” on page 16.)

In theory, GET is for getting data from the server and POST is for sending data there. However, GET appends the form data (called a *query string*) to an URL, in the form of key/value pairs from the HTML form, for example, `name=John`. In the query string, key/value pairs are separated by & characters, spaces are converted to + characters, and special characters are converted to their hexadecimal equivalents. Because the query string is in the URL, the page can be bookmarked or sent as email with its query string. The query string is usually limited to a relatively small number of characters.

The POST method, however, passes data of unlimited length as an HTTP request body to the server. The user working in the client Web browser cannot see the data that is being sent, so POST requests are ideal for sending confidential data (such as a credit card number) or large amounts of data to the server.

Writing the Bean

If your JSP application uses a bean, you can write the bean according to the design patterns outlined in the *JavaBeans component architecture*, remembering these general points:

- If you use a `<jsp:getProperty>` tag in your JSP source file, you need a corresponding `get` method in the bean.
- If you use a `<jsp:setProperty>` tag in your JSP source file, you need one or more corresponding `set` methods in the bean.

Setting properties in and getting properties from a bean is explained a bit more in the next section.

Getting Data From the Form to the Bean

Setting properties in a bean from an HTML form is a two-part task:

- Creating or locating the bean instance with `<jsp:useBean>`
- Setting property values in the bean with `<jsp:setProperty>`

The first step is to instantiate or locate a bean with a `<jsp:useBean>` tag *before* you set property values in the bean. In a JSP source file, the `<jsp:useBean>` tag must appear above the `<jsp:setProperty>` tag. The `<jsp:useBean>` tag first looks for a bean instance with the name you specify, but if it doesn't find the bean, it instantiates one. This allows you to create a bean in one JSP file and use it in another, as long as the bean has a large enough scope.

The second step is to set property values in the bean with a `<jsp:setProperty>` tag. The easiest way to use `<jsp:setProperty>` is to define properties in the bean with names that match the names of the form elements. You would also define corresponding `set` methods for each property. For example, if the form element is named `username`, you would define a property `username` property and methods `getUsername` and `setUsername` in the bean.

If you use different names for the form element and the bean property, you can still set the property value with `<jsp:setProperty>`, but you can only set one value at a time. For more information on the syntax variations of `<jsp:setProperty>`, see the *JavaServer Pages Syntax Card*.

Checking the Request Object

The data the user enters is stored in the `request` object, which usually implements javax.servlet.`HttpServletRequest` (or if your implementation uses a different protocol, another interface that is subclassed from `javax.servlet.ServletRequest`).

You can access the `request` object directly within a scriptlet. Scriptlets are discussed in more detail in the next section, but for now it's enough to know that they are fragments of code written in a scripting language and placed within `<%` and `%>` characters. In JSP version 1.1, you must use the Java programming language as your scripting language.

You may find some of these methods useful with the `request` object:

Method	Defined In	Job Performed
`getRequest`	`javax.servlet.jsp.PageContext`	Returns the current `request` object
`getParameterNames`	`javax.servlet.ServletRequest`	Returns the names of the parameters `request` currently contains
`getParameterValues`	`javax.servlet.ServletRequest`	Returns the values of the parameters `request` currently contains
`getParameter`	`javax.servlet.ServletRequest`	Returns the value of a parameter if you provide the name

You'll find other methods as well, those defined in `ServletRequest`, `HttpServletRequest`, or any subclass of `ServletRequest` that your implementation uses.

The JSP container always uses the `request` object behind the scenes, even if you do not call it explicitly from a JSP file.

Returning Data to the JSP Page

Once the user's data has been sent to the server, you may want to retrieve the data and display it in the JSP page. To do this, use the `<jsp:getProperty>` tag, giving it the bean name and property name:

```
<h1>Hello, <jsp:getProperty name="abean" property="username"/>!
```

The bean names you use on the `<jsp:useBean>`, `<jsp:setProperty>`, and `<jsp:getProperty>` tags must match, for example:

hellouser.jsp:

```
<jsp:useBean id="abean" scope="session" class="hello.NameHandler" />
<jsp:setProperty name="abean" property="*" />
```

response.jsp:
```
<h1>Hello, <jsp:getProperty name="abean" property="username"/>!
```

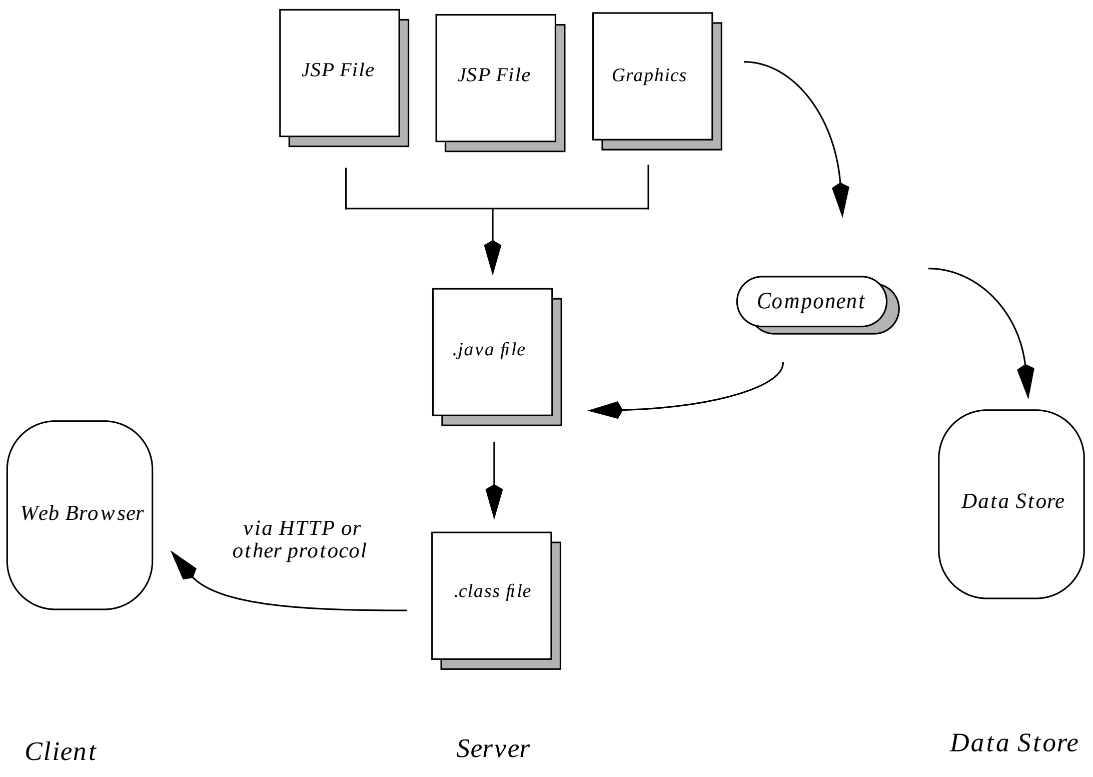
In this example, the tags are in two files, but the bean names still must match. If they don't, Tomcat throws an error, possibly a fatal one.

The response the JSP container returns to the client is within the implicit `response` object, which the JSP container creates.

How the JSP Page Is Compiled

It is very important to understand how a JSP page is compiled when your user loads it into a Web browser. The compilation process is illustrated in FIGURE 1-5.

FIGURE 0-5 How a JSP Page is Compiled



First of all, a JSP application is usually a collection of JSP files, HTML files, graphics and other resources. When the user loads the page for the first time, the files that make up the application are all translated together, without any dynamic data, into one Java source file (a `.java` file) with a name that your JSP implementation defines. Then, the `.java` file is compiled to a `.class` file. In most implementations, the `.java` file is a Java servlet that complies with the Java Servlet API. This entire stage is known as **translation time**.

When the user makes a request of the JSP application (in this case, when the user enters something in the form and clicks Submit), one or more of the application's components (a bean, enterprise bean, or servlet) handles data the user submits or retrieves data dynamically from a data store and returns the data to the `.java` file where it is recompiled in the `.class` file. The `.class` file, being a Java servlet, returns the data to the client Web browser by its `service` method. When the user makes a new request, the component obtains or handles the data again and returns it to the `.java` file, which is compiled again into the `.class` file. This stage is known as **request time**.

How to Run the Example

Install the example as described in "Installing and Running the Example Applications" on page 2. Then, open a Web browser and go to:

```
http://localhost:8080/hellouser/hellouser.jsp
```

Using Scripting Elements

At some point, you will probably want to add some good, old-fashioned programming to your JSP files. The JSP tags are powerful and encapsulate tasks that would be difficult or time-consuming to program. But even so, you will probably still want to use scripting language fragments to supplement the JSP tags.

The scripting languages that are available to you depend on the JSP container you are using. With Tomcat, you must use the Java^{TM} programming language for scripting, but other vendors' JSP containers may include support for other scripting languages).

How To Add Scripting

First, you'll need to know a few general rules about adding scripting elements to a JSP source file:

1. Use a `page` directive to define the scripting language used in the JSP page (unless you are using the Java language, which is a default value).
2. The declaration syntax `<%! ... %>` declares variables or methods.
3. The expression syntax `<%= ... %>` defines a scripting language expression and casts the result as a `String`.
4. The scriptlet syntax `<% ... %>` can handle declarations, expressions, or any other type of code fragment valid in the page scripting language.
5. When you write a scriptlet, end the scriptlet with `%>` before you switch to HTML, text, or another JSP tag.

The Difference Between <%, <%=, and <%!

Declarations, expressions, and scriptlets have similar syntax and usage, but also some important differences. Let's explore the similarities and differences here, with some examples.

Declarations (between `<%!` and `%>` tags) contain one or more variable or method declarations that end or are separated by semicolons:

```
<%! int i = 0; %>
<%! int a, b; double c; %>
<%! Circle a = new Circle(2.0); %>
```

You must declare a variable or method in a JSP page *before* you use it in the page. The scope of a declaration is usually a JSP file, but if the JSP file includes other files with the `include` directive, the scope expands to cover the included files as well.

Expressions (between `<%=` and `%>` tags) can contain any language expression that is valid in the page scripting language, but without a semicolon:

```
<%= Math.sqrt(2) %>
<%= items[i] %>
<%= a + b + c %>
<%= new java.util.Date() %>
```

The definition of a valid expression is up to the scripting language. When you use the Java language for scripting, what's between the expression tags can be any expression defined in the *Java Language Specification*. The parts of the expression are evaluated in left-to-right order. One key difference between expressions and scriptlets (which are described next and appear between `<%` and `%>` tags) is that a semicolon is not allowed within expression tags, even if the same expression requires a semicolon when you use it within scriptlet tags.

Scriptlets (between `<%` and `%>` tags) allow you to write any number of valid scripting language statements, like this:

```
<%
     String name = null;
     if (request.getParameter("name") == null) {
%>
```

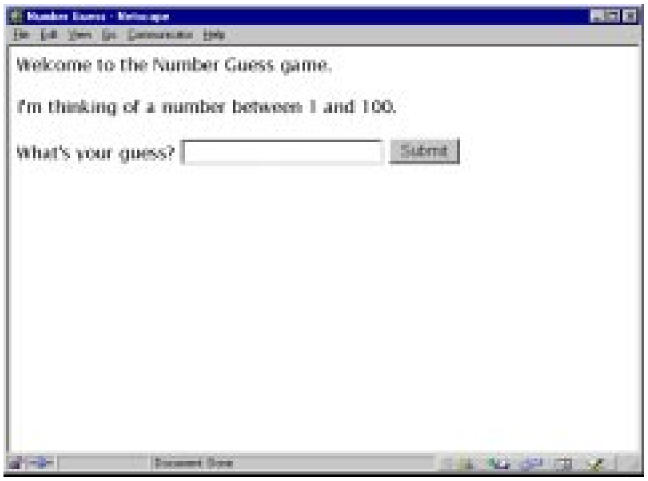
Remember that in a scriptlet you must end a language statement with a semicolon if the language requires it.

When you write a scriptlet, you can use any of the JSP implicit objects or classes imported by the `page` directive, declared in a declaration, or named in a `<jsp:useBean>` tag.

The Number Guess Game

The Number Guess game makes good use of scriptlets and expressions, as well as using the knowledge of HTML forms you gained in the last example.

FIGURE 0-6 About to Guess a Number

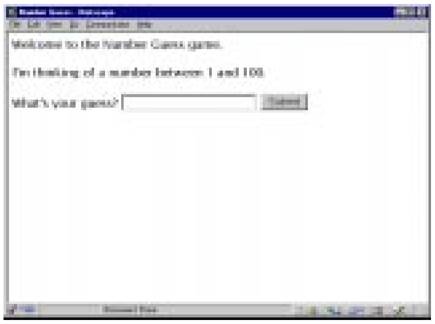


Welcome to the Number Guess game.

I'm thinking of a number between 1 and 100.

What's your guess?

Example Code



Welcome to the Number Guess game.

I'm thinking of a number between 1 and 100.

What's your guess?

CODE EXAMPLE 0-7 Displaying the Number Guess Screen (*numguess.jsp*)

```
<!--
Number Guess Game
Written by Jason Hunter, CTO, K&A Software
jasonh@kasoftware.com, http://www.servlets.com
Copyright 1999, K&A Software
Distributed by Sun Microsystems with permission
-->

<%@ page import = "num.NumberGuessBean" %>

<jsp:useBean id="numguess" class="num.NumberGuessBean"
   scope="session" />
<jsp:setProperty name="numguess" property="*" />

<html>
<head><title>Number Guess</title></head>
<body bgcolor="white">
<font size=4>

<% if (numguess.getSuccess() ) { %>

   Congratulations!  You got it.
   And after just <%= numguess.getNumGuesses() %> tries.<p>

   <% numguess.reset(); %>
   Care to <a href="numguess.jsp">try again</a>?

<% } else if (numguess.getNumGuesses() == 0) { %>

   Welcome to the Number Guess game.<p>
   I’m thinking of a number between 1 and 100.<p>
   <form method=get>
   What’s your guess? <input type=text name=guess>
   <input type=submit value="Submit">
   </form>

<% } else { %>

   Good guess, but nope.  Try <b><%= numguess.getHint() %></b>.
   You have made <%= numguess.getNumGuesses() %> guesses.<p>

   I’m thinking of a number between 1 and 100.<p>

   <form method=get>
   What’s your guess? <input type=text name=guess>
   <input type=submit value="Submit">
   </form>

<% } %>

</font>
</body>
</html>
```

CODE EXAMPLE 0-8 Handling the Guess (*NumberGuessBean.java*)

```
// Number Guess Game
// Written by Jason Hunter, CTO, K&A Software
// jasonh@kasoftware.com, http://www.servlets.com
// Copyright 1999, K&A Software
// Distributed by Sun Microsystems with permission

package num;

import java.util.*;
public class NumberGuessBean {

  int answer;
  boolean success;
  String hint;
  int numGuesses;

public NumberGuessBean() {
  reset();
}

public void setGuess(String guess) {
  numGuesses++;

  int g;
```

```
    try {
    g = Integer.parseInt(guess);
    }
    catch (NumberFormatException e) {
      g = -1;
    }
    if (g == answer) {
      success = true;
    }
    else if (g == -1) {
      hint = "a number next time";
    }
    else if (g < answer) {
      hint = "higher";
    }
    else if (g > answer) {
      hint = "lower";
    }
  }
    public boolean getSuccess() {
      return success;
    }
    public String getHint() {
      return "" + hint;
    }

    public int getNumGuesses() {
      return numGuesses;
    }

    public void reset() {
      answer = Math.abs(new Random().nextInt() % 100) + 1;
      success = false;
      numGuesses = 0;
    }
  }
```

Using Scripting Elements in a JSP File

The file numguess.jsp is an interesting example of the use of scripting elements, because it is structured as you might structure a Java programming language source file, with a large if ... else statement within scriptlet tags. The difference is that the body of each statement clause is written in HTML and JSP tags, rather than in a programming language.

You are not required to write scriptlets mingled with HTML and JSP tags, as shown in `numguess.jsp`. Between the `<%` and `%>` tags, you can write as many lines of scripting language code as you want. In general, doing less processing in scriptlets and more in components like servlets or Beans makes your application code more reusable and portable. Nonetheless, how you write your JSP application is your choice, and Tomcat specifies no limit on the length of a scriptlet.

Mingling Scripting Elements with Tags

When you mingle scripting elements with HTML and JSP tags, you must always end a scripting element before you start using tags and then reopen the scripting element afterwards, like this:

```
<% } else { %> <!-- closing the scriptlet before the tags start -->
```

... tags follow ...

```
<% } %> <!-- reopening the scriptlet to close the language block -->
```

At first, this may look a bit strange, but it ensures that the scripting elements are transformed correctly when the JSP source file is compiled.

When Are the Scripting Elements Executed?

A JSP source file is processed in two stages—*HTTP translation time* and *request processing time*.

At HTTP translation time, which occurs when a user first loads a JSP page, the JSP source file is compiled to a Java class, usually a Java servlet. The HTML tags and as many JSP tags as possible are processed at this stage, before the user makes a request.

Request processing time occurs when your user clicks in the JSP page to make a request. The request is sent from the client to the server by way of the `request` object. The JSP container then executes the compiled JSP file, or servlet, using the request values the user submitted.

When you use scripting elements in a JSP file, you should know when they are evaluated. Declarations are processed at request processing time and are available to other declarations, expressions, and scriptlets in the compiled JSP file. Expressions are evaluated at request processing time. The value of each expression is converted to a `String` and inserted in place in the compiled JSP file. Scriptlets also are evaluated at request processing time, using the values of any declarations that are made available to them.

How To Run the Example

Install the example as described in “Installing and Running the Example Applications” on page 2. Then, open a Web browser and go to:

```
http://localhost:8080/examples/jsp/num/numguess.jsp
```

Handling Exceptions

What was happening the last time you used a JSP application and you entered something incorrectly? If the application was well written, it probably threw an exception and displayed an error page. Exceptions that occur while a JSP application is running are called **runtime exceptions**.

Just as in a Java application, an exception is an object that is an instance of `java.lang.Throwable` or one of its subclasses. `Throwable` has two standard subclasses—`java.lang.Exception,` which describes exceptions, and `java.lang.Error`, which describes errors.

Errors are different from exceptions. Errors usually indicate linkage or virtual machine problems that your Web application probably won't recover from, such as running out of memory. Exceptions, however, are conditions that can be caught and recovered from. These exceptions might be, for example, a `NullPointerException` or a `ClassCastException`, which tell you that a null value or a value of the wrong data type has been passed to your application while it is running.

Runtime exceptions are easy to handle in a JSP application, because they are stored one at a time in the implicit object named `exception`. You can use the `exception` object in a special type of JSP page called an **error page**, where you display the exception's name and class, its stack trace, and an informative message for your user.

A runtime exception is thrown by the compiled JSP file, the Java class file that contains the translated version of your JSP page. This means that your application has already been compiled and translated correctly. (Exceptions that occur while a file is being compiled or translated are not stored in the `exception` object and have their messages displayed in the command window, rather than in error pages. These are not the type of exception described in this tutorial.)

This tutorial describes how to create a simple JSP application with several display pages, a JavaBeans component, and one error page that gives informative error messages to the user. In this example, the bean tracks which JSP page the user was working in when the exception was thrown, which gives you, the developer, valuable information so that you can display an informative message. This is a simple error tracking mechanism; we will describe more complex ones later in this book.

How To Add Error Pages

Even though we call them error pages, the specialized JSP pages we describe here actually display information about exceptions. To add error pages that display exception information to a Web application, follow these steps:

1. Write your component so that it throws certain exceptions under certain conditions.
2. In the JSP file, use a `page` directive with `errorPage` set to the name of a JSP file that will display a message to the user when an exception occurs.
3. Write an error page file, using a `page` directive with `isErrorPage="true"`. In the error page file, use the `exception` object to get information about the exception.
4. Use a simple tracking mechanism in your component to help you gather information about what your user was doing when the exception was thrown.
5. Use messages, either in your error page file or included from other files, to give your user information relevant to what he or she was doing when the exception was thrown.

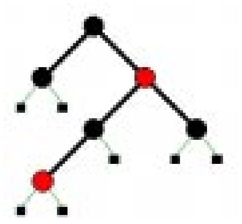
An Email Address Finder Example

This example, named `email`, stores names and email addresses in a map file based on the `java.util.TreeMap` class defined in the JDK 1.2. The `TreeMap` class creates a data structure called a **red-black tree**. In the tree, data is stored with a key and a value. In this example, the name is the key and the email address is the value.

When you add an entry to the map file, you enter both a name (the key) and an email address (the value). You can look up or delete an email address by entering just a name. The name cannot be null because it is a key. If a user tries to enter a null name, the application throws an exception and displays an error page.

So What's a Red-Black Tree?

For those of you who are curious about algorithms, a red-black tree is an **extended binary tree** that looks something like this (conceptually, at least):



If you are viewing this document online, you will see that some nodes are red and some are black. If you are viewing this document in print, the red nodes look a shade or two lighter than the black.

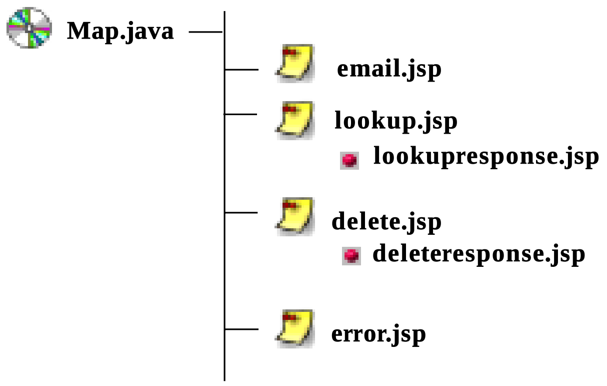
The red-black tree has nodes that are either **leaf nodes** or **branch nodes**. Leaf nodes are the small nodes at the end of a line, while branch nodes are the larger nodes that connect two or more lines. Data is stored in a balanced structure in the tree, using the following conditions:

- Every node has two children or is a leaf.
- Every node is colored red or black.
- Every leaf node is colored black.
- If a node is red, then both of its children are black.
- Every path from the root to a leaf contains the same number of black nodes.

If you want more detail on how a tree map works, you can find it in *Introduction to Algorithms* by Corman, Leiserson, and Rivest. The advantage of a tree map is that you can create a map file that stores data in ascending order (sorted by keys) and that has fast search times.

How the Example Is Structured

The `email` example has three pages with HTML forms, two response files, one error page, and one JavaBeans component. You can visualize the file structure as something like this:



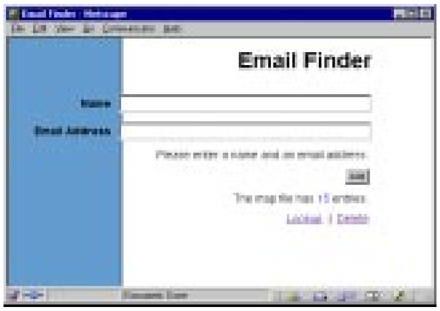
- `Map.java` is a JavaBeans component that creates the map file.
- `email.jsp` is a JSP page that displays a form where the user enters a name and email address.
- `lookup.jsp` is a JSP page that lets a user search for an email address that matches a name.
- `lookupresponse.jsp` is included in `lookup.jsp` and displays the entry the user wants to look up.
- `delete.jsp` is a JSP page that lets the user delete an email address that matches a name.
- `deleteresponse.jsp` is included in `delete.jsp` and displays the entry that was deleted from the map file.
- `error.jsp` is an error page that displays information about handling exceptions that occur while adding, looking up, or deleting entries in the map file.

The code for *email* is shown in CODE EXAMPLE 1-9 through CODE EXAMPLE 1-15, along with miniature versions of its screens. You may want to install and run the example while you look at the code. The instructions are in “How To Run the Example” on page 45.

CODE EXAMPLE 0-9 Adding a Name and Email Address (*email.jsp*)

```
<%@ include file="copyright.html" %>

<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>

<jsp:useBean id="mymap" scope="session" class="email.Map" />
<jsp:setProperty name="mymap" property="name" param="name" />
<jsp:setProperty name="mymap" property="email" param="email" />
```

```
<% mymap.setAction( "add" );  %>

<html>
<head><title>Email Finder</title></head>
<body bgcolor="#ffffff" background="background.gif" link="#000099">
```


Email Finder

Please enter a name and an email address.

The map file has 15 entries.

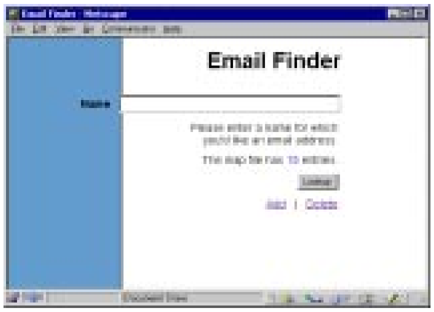
```
<!-- the form table -->

<form method="get">
<table border="0" cellspacing="0" cellpadding="5">

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<h1>Email Finder</h1> </td>
</tr>

<tr>
<td width="120" align="right"><b>Name</b></td>
<td align="left"><input type="text" name="name" size="35"></td>
</tr>

<tr>
<td width="120" align="right"><b>Email Address</b></td>
<td align="left"><input type="text" name="email" size="35"></td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
Please enter a name and an email address.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<input type="submit" value="Add">
```

```
</td>
</tr>

<!-- here we call the put method to add the
     name and email address to the map file -->

<%
    String rname = request.getParameter( "name" );
    String remail = request.getParameter( "email" );
    if ( rname != null) {
       mymap.put( rname, remail );
    }
%>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
The map file has <font color="blue"><%= mymap.size() %>
</font> entries.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<a href="lookup.jsp">Lookup</a>&nbsp; | &nbsp;
    <a href="delete.jsp">Delete</a>
</td>
</tr>

</table>
</form>

</body>
</html>
```

CODE EXAMPLE 0-10 Looking Up a Name in the Map File (*lookup.jsp*)

```
<%@ include file="copyright.html" %>

<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>

<jsp:useBean id="mymap" scope="session" class="email.Map" />
<jsp:setProperty name="mymap" property="name" param="name" />

<% mymap.setAction( "lookup" ); %>

<html>
```

```
<head><title> Email Finder </title></head>
<body bgcolor="#ffffff" background="background.gif" link="#000099">

<form method="get">
<table border="0" cellspacing="0" cellpadding="5">
```

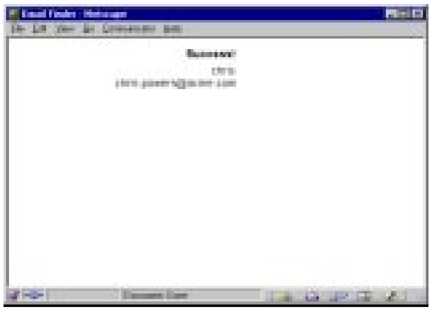


Please enter a name for which you'd like an email address.
The map file has 10 entries.

```
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<h1>Email Finder</h1> </td>
</tr>

<tr>
<td width="120" align="right"> <b>Name</b></td>
<td align="left">
<input type="text" name="name" size="35"></td> </tr>
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
Please enter a name for which
<br>
you’d like an email address.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
The map file has <font color="blue"> <%= mymap.size() %></font>
entries.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right"> <input type="submit" value="Lookup"> </td>
</tr>
```

```
<%  if ( request.getParameter( "name" ) != null ) {  %>
    <%@ include file="lookupresponse.jsp" %>
<%  }  %>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<a href="email.jsp">Add</a> &nbsp; | &nbsp;
    <a href="delete.jsp">Delete</a>
</td>
</tr>

</table>
</form>

</body>
</html>
```

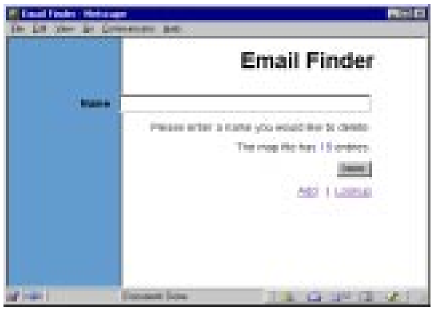
CODE EXAMPLE 0-11 Displaying the Lookup Response (*lookupresponse.jsp*)



```
<%@ page import="java.util.*, email.Map" %>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<b> Success! </b>
</td>
</tr>
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<jsp:getProperty name="mymap" property="name" />
<br>
<jsp:getProperty name="mymap" property="email" />
</td>
</tr>
```

CODE EXAMPLE 0-12 Deleting an Email Address (*delete.jsp*)

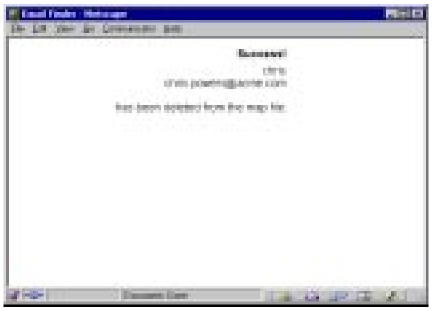
```
<%@ include file="copyright.html" %>

<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>

<jsp:useBean id="mymap" scope="session" class="email.Map" />
<jsp:setProperty name="mymap" property="name" param="name" />

<!-- tags the JSP page so that we can display
     the right exception message later -->

<% mymap.setAction( "delete" ); %>

<html>
<head><title> Email Finder </title></head>
<body bgcolor="#ffffff" background="background.gif" link="#000099">
```



```
<form method="get">
<table border="0" cellspacing="0" cellpadding="5">

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<h1>Email Finder</h1> </td>
</tr>
<tr>
<td width="120" align="right"><b>Name</b></td>
<td align="left"> <input type="text" name="name" size="35"> </td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
Please enter a name you would like to delete.
</td>
</tr>
```

```
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
The map file has <font color="blue"> <%= mymap.size() %></font>
entries.
</td>
</tr>

<tr>
<td width="120"> &nbsp; </td>
<td align="right"> <input type="submit" value="Delete"> </td>
</tr>

<!-- display the name and email address, then
     delete them from the map file -->

<%  if ( request.getParameter( "name" ) != null ) {  %>
        <%@ include file="deleteresponse.jsp" %>
<%
        mymap.remove( request.getParameter("name") ) ;
    }
%>

<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<a href="email.jsp">Add</a> &nbsp; | &nbsp;
    <a href="lookup.jsp">Lookup</a>
</td>
</tr>

</table>
</form>

</body>
</html>
```

CODE EXAMPLE 0-13 Displaying the Delete Response (*deleteresponse.jsp*)



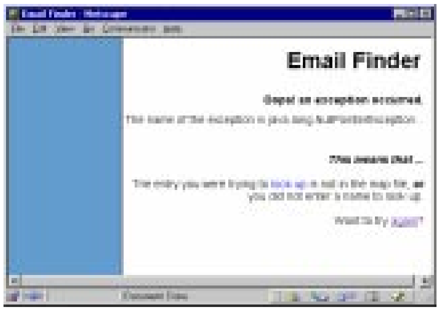
Success!
chris
chris.powers@acme.com
has been deleted from the map file.

```
<%@ page import="java.util.*, email.Map"  %>

<tr>
<td width="120"> &nbsp;
</td>
<td align="right"> <b>Success!</b>
</td>
</tr>
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<jsp:getProperty name="mymap" property="name" />
<br>
<jsp:getProperty name="mymap" property="email" />
<br><p>
has been deleted from the map file.
</td>
</tr>
```

CODE EXAMPLE 0-14 Displaying Exception Messages (*error.jsp*)

```
<%@ include file="copyright.html" %>

<%@ page isErrorPage="true" import="java.util.*, email.Map" %>
<jsp:useBean id="mymap" scope="session" class="email.Map" />

<html>
<head><title>Email Finder</title></head>
<body bgcolor="#ffffff" background="background.gif" link="#000099">

<table border="0" cellspacing="0" cellpadding="5">
<tr>
<td width="150" align="right"> &nbsp; </td>
<td align="right" valign="bottom"> <h1> Email Finder </h1> </td>
</tr>
```

```
<tr>
<td width="150" align="right"> &nbsp; </td>
<td align="right"> <b>Oops! an exception occurred.</b> </td>
</tr>
```



Email Finder

Oops! an exception occurred.

The name of the exception is java.lang.NullPointerException.

This means that ...

The entry you were trying to look up is not in the map file, **or** you did not enter a name to look up.

Want to try again?

```
<tr>
<td width="150" align="right"> &nbsp; </td>
<td align="right"> The name of the exception is
   <%= exception.toString() %>.
</td>
</tr>


<tr>
<td width="150" align="right"> &nbsp; </td>
<td align="right"> &nbsp; </td>
</tr>

<% if (mymap.getAction() == "delete" ) {  %>
   <tr>
   <td width=150 align=right> &nbsp; </td>
   <td align=right>
   <b>This means that ...</b>
   <p>The entry you were trying to
   <font color="blue">delete</font> is not in the map file <br>
   <b><i>or</i></b>
   <br>
   you did not enter a name to delete.
   <p>
   Want to try <a href="delete.jsp">again</a>?
   </td>
   </tr>

<% }

else if (mymap.getAction() == "lookup" ) { %>
   <tr>
   <td width="150" align="right"> &nbsp; </td>
   <td align="right">
   <b><i>This means that ...</b></i>
   <p>the entry you were trying to
   <font color="blue">look up</font>
   is not in the map file, <b><i>or</i></b>
   <br>
   you did not enter a name to look up.
   <p>
   Want to try <a href="lookup.jsp">again</a>?
   </td>
   </tr>

<% }

else if (mymap.getAction() == "add" ) { %>
   <tr>
   <td width="150" align="right"> &nbsp; </td>
   <td align="right">
   <b><i>This means that ...</b></i>
   <p>You were trying to <font color="blue">add</font>
   an entry with a name of null.
   <br>
   The map file doesn't allow this.
   <p>
   Want to try <a href="email.jsp">again</a>?
   </td>
   </tr>

<%  }  %>

</table>
```

CODE EXAMPLE 0-15 Creating the Map File (*Map.java*)

```
package email;
import java.util.*;

public class Map extends TreeMap {

// In this treemap, name is the key and email is the value

   private String name, email, action;
   private int count = 0;

   public Map() { }

   public void setName( String formName ) {
      if ( formName != "" ) {
         name = formName;
      }
   }

   public String getName()
      return name;
   }

   public void setEmail( String formEmail ) {
      if ( formEmail != "" ) {
         email = formEmail;
         System.out.println( name );// for debugging only
         System.out.println( email );// for debugging only
      }
   }

   public String getEmail() {
      email = get(name).toString();
      return email;
   }

   public void setAction( String pageAction ) {
      action = pageAction;
   }

   public String getAction() {
      return action;
   }

}
```

Handling Exceptions in the Bean

In this example, the code that throws exceptions is in the `TreeMap` class, which our `email.Map` bean extends, so we won't need to write code that throws exceptions in the bean.

The methods that we use from `TreeMap` are shown below, with their exceptions:

- `public Object get( Object key )`
  `throws ClassCastException, NullPointerException`
  - *retrieves an entry from the map file*
- `public Object put( Object key, Object value )`
  `throws ClassCastException, NullPointerException`
  - *adds an entry to the map file*

- `public Object remove( Object key )`
  `throws ClassCastException, NullPointerException`
  - removes an entry from the map file
- `int size()`
  - returns the number of entries in the map file

Of course, if you need more information about these methods, you can find it in the Javadoc API reference for `java.util.TreeMap`.

The `TreeMap` class throws a `ClassCastException` when the user tries to enter data of the wrong type in the map file, for example, an `int` where the map file is expecting a `String`. Keep in mind that the TreeMap class is also used with Java client applications. In our JSP application, this exception won't occur, because the user enters a name and an email address in an HTML form, which always passes data as strings to the bean. Even if the user typed `6` as a name, the value is still sent as a `String`.

However, the `get`, `put`, and `remove` methods throw a `NullPointerException` if the user enters nothing and a null value is passed to the bean. This is the most common exception that the *email* application needs to handle. This exception might occur while your user is trying to add, look up, or remove an entry from the map file. Remember that the key (in this case, the name) cannot be null.

When the User Tries to Add a Null Value

The first case, where the user attempts to add a null name or email address, is handled by some simple code in the bean and in `email.jsp`. (Here *null* means the user has entered nothing in the form text box. It does not handle the case where the user enters one or more blank spaces, then presses Return.)

The code that handles adding null values is in the `setName` and `setEmail` methods of `Map.java` and in a scriptlet in `email.jsp` (CODE EXAMPLE 1-16):

CODE EXAMPLE 0-16 Catching a Null Value on Add

Map.java:

```
public void setName( String formName ) {
  if ( formName != "" ) {
     name = formName;
  }
}

public void setEmail( String formEmail ) {
   if ( formEmail != "" ) {
     email = formEmail;
     System.out.println( name ); // for debugging only
```

```
      System.out.println( email ); // for debugging only
    }
}
```

email.jsp:

```
<%
    String rname = request.getParameter( "name" );
    String remail = request.getParameter( "email" );
    if ( rname != null) {
      mymap.put( rname, remail );
    }
%>
```

Both `setName` and `setEmail` check whether the user has entered a null value in the form before setting their respective properties. If the form value is null, the bean does not set a property, the `put` method does not add a value to the map file, and no exception is thrown.

When the User Tries to Look Up a Null Value

But if you go to the *Lookup* or *Delete* page of the example and try to look up or delete an entry that isn't in the map file at all, the *email* application throws a `NullPointerException` and displays the error page. The code that handles looking up null values is shown in CODE EXAMPLE 1-17.

CODE EXAMPLE 0-17 Catching a Null Value on Look Up

lookup.jsp:

```
<%  if ( request.getParameter( "name" ) != null ) {  %>
     <%@ include file="lookupresponse.jsp" %>
<%  }  %>
```

lookupresponse.jsp:

```
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<font face="helvetica" size="-2">
<jsp:getProperty name="mymap" property="name" />
<br>
<jsp:getProperty name="mymap" property="email" />
</font>
</td>
</tr>
```

This example has two pieces of code that work together. The page `lookup.jsp`, where you enter a name you want to look up in the map file, has a scriptlet that checks whether or not the user has entered a name in the form. If the user doesn't enter a name, or enters a name that doesn't exist in the map file, the bean throws a `NullPointerException` and the application displays the error page—which is the desired behavior! In this case, you can be happy that the error page is displayed.

You may have noticed that the lines from `lookupresponse.jsp` use the `<jsp:getProperty>` tag to retrieve the name and email address from the bean. You could also try to retrieve the email address using expressions, something like this:

```
<%= request.getParameter( "name" ) %>
<br>
<%= mymap.get( request.getParameter( "name" ) ) %>
```

If you use these lines, the application would behave a little differently. Rather than throwing a `NullPointerException` and displaying an error page, it would display the name the user entered, with the word *null* below it in the JSP page. In Tomcat, the `<jsp:getProperty>` tag intentionally handles null values differently than scriptlets or expressions. The way null values are handled will vary according to the JSP container you use.

When the User Tries to Delete a Null Value

Handling the case of a user trying to delete a null value is very similar to handling the lookup of a null value. The code that handles null values that occur while you are trying to delete an entry is shown in CODE EXAMPLE 1-18.

CODE EXAMPLE 0-18 Catching a Null Value on Delete

delete.jsp:

```
<% if ( request.getParameter( "name" ) != null ) {  %>
      <%@ include file="deleteresponse.jsp" %>
<%
     mymap.remove( request.getParameter("name") ) ;
   }
%>
```

deleteresponse.jsp:

```
<tr>
<td width="120"> &nbsp; </td>
<td align="right">
<font face="helvetica" size="-2">
<jsp:getProperty name="mymap" property="name" />
<br>
<jsp:getProperty name="mymap" property="email" />
```

```
<br><p>
has been deleted from the map file.
</font>
</td>
</tr>
```

Calling an Error Page From Another Page

To link the display pages to the error page, each display page in the *email* application uses a `page` directive with the `errorPage` attribute, like this:

```
<%@ page isThreadSafe="false" import="java.util.*, email.Map"
    errorPage="error.jsp" %>
```

In the code examples, the files that use this directive are `email.jsp`, `lookup.jsp`, and `delete.jsp`. You can only specify one error page for each JSP page.

This means that you can design a JSP application so that each JSP page calls a different error page, or so that several JSP pages call one error page. In the email application, several JSP pages call one error page, as it simplifies the number of files you need to maintain for one application. In designing your applications, the choice is up to you.

You should always use at least one error page in a JSP application. If you don't specify an error page, the exception message and stack trace are displayed in the command window from which the JSP container was started, while the Web browser displays a non-informative HTTP error message, for example, a 404 or 501 message. This is definitely not a graceful way to handle exceptions.

Writing an Error Page

An error page is different from an ordinary JSP page. In an error page, you must explicitly set the `isErrorPage` attribute of the `page` directive to `true`. You also have access to the `exception` object, which gives you information about the exception.

First, let's look at an example of the `page` directive for an error page:

```
<%@ page isErrorPage="true" import="java.util.*, email.Map" %>
```

Once you have set `isErrorPage` to `true`, you can use the `exception` object. `exception` is of type `java.lang.Throwable`, so you can use any of the methods defined in `Throwable` with `exception` in a scriptlet or expression, for example:

- `<%= exception.toString() %>`
- `<% exception.printStackTrace(); %>`

The expression `exception.toString()` displays the exception's class name, for example, `java.lang.NullPointerException`, while `exception.printStackTrace()` displays the exception's stack trace. The class name and stack trace are probably very helpful to you the developer, but probably not very helpful to your user. To get around this, you may want to write some type of tracking mechanism to provide information that helps you give an informative message to your user.

Writing a Simple Tracking Mechanism

The *email* example uses a property named `action` in `Map.java` to track which page the user was working in when the exception was thrown. That gives you valuable information to help you write an informative error message for your user. The bean has a variable named `action`, a `getAction` method, and a `setAction` method. The variable and method declarations in the bean look like this:

```
private String action;

public void setAction( String pageAction ) {
  action = pageAction;
}

public String getAction() {
  return action;
}
```

Each of the pages `email.jsp`, `lookup.jsp`, and `delete.jsp` sets the value of `action` with a line like this one (which comes from `email.jsp`):

```
<% mymap.setAction( "add" ); %>
```

If an exception occurs, `error.jsp` checks the value of `action` and includes an appropriate message for each value, using lines like these:

```
<% if (mymap.getAction() == "delete" ) {  %>
.. text message here ..
else if (mymap.getAction() == "lookup" ) { %>
.. text message here ..
else if (mymap.getAction() == "add" ) { %>
.. text message here ..
<%  }  %>
```

Of course, this is a very simple way to implement tracking. If you move into developing J2EE applications, you can write applications that save state.

How To Run the Example

Install the example as described in “Installing and Running the Example Applications” on page 2. Then, open a Web browser and go to:

- `http://localhost:8080/email/email.jsp`