


FIRST SERVLETS

Topics in This Chapter

- 
- The basic structure of servlets
 - A simple servlet that generates plain text
 - The process of compiling, installing, and invoking servlets
 - A servlet that generates HTML
 - Some utilities to help build HTML
 - The life cycle of servlets
 - An example of reading initialization parameters
 - An example that uses initialization and page modification dates
 - Servlet debugging strategies
 - A tool for interactively talking to servlets

Home page for this book: <http://www.coreservlets.com>.

Home page for sequel: <http://www.moreservlets.com>.

Servlet and JSP training courses: <http://courses.coreservlets.com>.

Chapter 2

The previous chapter showed you how to install the software you need and how to set up your development environment. Now you want to really write a few servlets. Good. This chapter shows you how, outlining the structure that almost all servlets follow, walking you through the steps required to compile and execute a servlet, and giving details on how servlets are initialized and when the various methods are called. It also introduces a few general tools that you will find helpful in your servlet development.

2.1 Basic Servlet Structure

Listing 2.1 outlines a basic servlet that handles GET requests. GET requests, for those unfamiliar with HTTP, are the usual type of browser requests for Web pages. A browser generates this request when the user types a URL on the address line, follows a link from a Web page, or submits an HTML form that does not specify a METHOD. Servlets can also very easily handle POST requests, which are generated when someone submits an HTML form that specifies METHOD="POST". For details on using HTML forms, see Chapter 16.

To be a servlet, a class should extend `HttpServlet` and override `doGet` or `doPost`, depending on whether the data is being sent by GET or by POST. If you want the same servlet to handle both GET and POST and to take the same action for each, you can simply have `doGet` call `doPost`, or vice versa.

Chapter 2 First Servlets**Listing 2.1 ServletTemplate.java**

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // Use "request" to read incoming HTTP headers
        // (e.g. cookies) and HTML form data (e.g. data the user
        // entered and submitted).

        // Use "response" to specify the HTTP response status
        // code and headers (e.g. the content type, cookies).

        PrintWriter out = response.getWriter();
        // Use "out" to send content to browser
    }
}

```

Both of these methods take two arguments: an `HttpServletRequest` and an `HttpServletResponse`. The `HttpServletRequest` has methods by which you can find out about incoming information such as form data, HTTP request headers, and the client's hostname. The `HttpServletResponse` lets you specify outgoing information such as HTTP status codes (200, 404, etc.), response headers (`Content-Type`, `Set-Cookie`, etc.), and, most importantly, lets you obtain a `PrintWriter` used to send the document content back to the client. For simple servlets, most of the effort is spent in `println` statements that generate the desired page. Form data, HTTP request headers, HTTP responses, and cookies will all be discussed in detail in the following chapters.

Since `doGet` and `doPost` throw two exceptions, you are required to include them in the declaration. Finally, you have to import classes in `java.io` (for `PrintWriter`, etc.), `javax.servlet` (for `HttpServlet`, etc.), and `javax.servlet.http` (for `HttpServletRequest` and `HttpServletResponse`).

Strictly speaking, `HttpServlet` is not the only starting point for servlets, since servlets could, in principle, extend mail, FTP, or other types of servers. Servlets for these environments would extend a custom class derived from `GenericServlet`, the parent class of `HttpServlet`. In practice, however, servlets are used almost exclusively for servers that communicate via HTTP (i.e., Web and application servers), and the discussion in this book will be limited to this usage.

2.2 A Simple Servlet Generating Plain Text

Listing 2.2 shows a simple servlet that just generates plain text, with the output shown in Figure 2–1. Section 2.3 (A Servlet That Generates HTML) shows the more usual case where HTML is generated. However, before moving on, it is worth spending some time going through the process of installing, compiling, and running this simple servlet. You'll find this a bit tedious the first time you try it. Be patient; since the process is the same each time, you'll quickly get used to it, especially if you partially automate the process by means of a script file such as that presented in the following section.

Listing 2.2 HelloWorld.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        out.println("Hello World");
    }
}
```

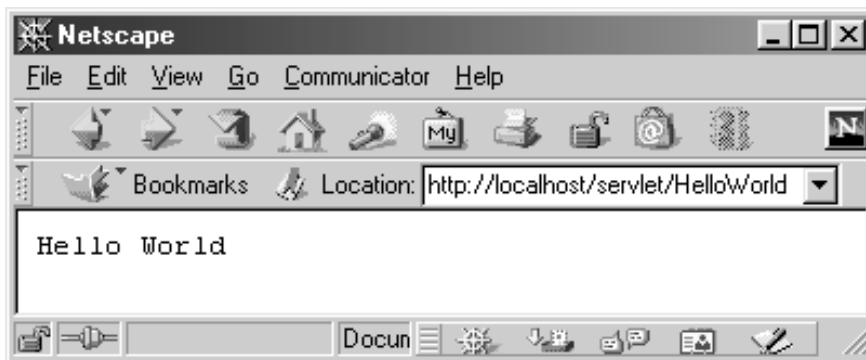


Figure 2–1 Result of Listing 2.2 (HelloWorld.java).

Chapter 2 First Servlets**Compiling and Installing the Servlet**

The first thing you need to do is to make sure that your server is configured properly and that your `CLASSPATH` refers to the JAR files containing the standard servlet classes. Please refer to Section 1.5 (Installation and Setup) for an explanation of this process.

The next step is to decide where to put the servlet classes. This location varies from server to server, so refer to your Web server documentation for definitive directions. However, there are some moderately common conventions. Most servers have three distinct locations for servlet classes, as detailed below.

1. **A directory for frequently changing servlet classes.**

Servlets in this directory are automatically reloaded when their class file changes, so you should use this directory during development. For example, this is normally `install_dir/servlets` with Sun's Java Web Server and IBM's WebSphere and `install_dir/myserver/servlet-classes` for BEA WebLogic, although most servers let the server administrator specify a different location. Neither Tomcat nor the JSWDK support automatic servlet reloading. Nevertheless, they still have a similar directory in which to place servlets; you just have to stop and restart the mini-server each time you change an existing servlet. With Tomcat 3.0, place servlets in `install_dir/webpages/WEB-INF/classes`. With the JSWDK 1.0.1, use `install_dir/webpages/WEB-INF/servlets`.

2. **A directory for infrequently changing servlet classes.**

Servlets placed in this location are slightly more efficient since the server doesn't have to keep checking their modification dates. However, changes to class files in this directory require you to restart the server. This option (or Option 3 below) is the one to use for "production" servlets deployed to a high-volume site. This directory is usually something like `install_dir/classes`, which is the default name with Tomcat, the JSWDK, and the Java Web Server. Since Tomcat and the JSWDK do not support automatic servlet reloading, this directory works the same as the one described in Option 1, so most developers stick with that previous option.

3. **A directory for infrequently changing servlets in JAR files.**

With the second option above, the class files are placed directly in the `classes` directory or in subdirectories corresponding to their package name. Here, the class files are packaged in a JAR file, and that file is then placed in the designated directory. With Tomcat, the JSWDK, the Java Web Server, and most other servers, this directory is `install_dir/lib`. You must restart the server whenever you change files in this directory.

Once you've configured your server, set your `CLASSPATH`, and placed the servlet in the proper directory, simply do "`javac HelloWorld.java`" to compile the servlet. In production environments, however, servlets are frequently placed into packages to avoid name conflicts with servlets written by other developers. Using packages involves a couple of extra steps that are covered in Section 2.4 (Packaging Servlets). Also, it is common to use HTML forms as front ends to servlets (see Chapter 16). To use them, you'll need to know where to place regular HTML files to make them accessible to the server. This location varies from server to server, but with the JSWDK and Tomcat, you place an HTML file in `install_dir/webpages/path/file.html` and then access it via `http://localhost/path/file.html` (replace `localhost` with the real hostname if running remotely). A JSP page can be installed anywhere that a normal HTML page can be.

Invoking the Servlet

With different servers, servlet classes can be placed in a variety of different locations, and there is little standardization among servers. To invoke servlets, however, there is a common convention: use a URL of the form `http://host/servlet/ServletName`. Note that the URL refers to `servlet`, singular, even if the real directory containing the servlet code is called `servlets`, plural, or has an unrelated name like `classes` or `lib`.

Figure 2-1, shown earlier in this section, gives an example with the Web server running directly on my PC ("`localhost`" means "the current machine").

Most servers also let you register names for servlets, so that a servlet can be invoked via `http://host/any-path/any-file`. The process for doing this is server-specific; check your server's documentation for details.

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

Chapter 2 First Servlets

2.3 A Servlet That Generates HTML

Most servlets generate HTML, not plain text as in the previous example. To build HTML, you need two additional steps:

1. Tell the browser that you're sending back HTML, and
2. Modify the `println` statements to build a legal Web page.

You accomplish the first step by setting the HTTP Content-Type response header. In general, headers are set by the `setHeader` method of `HttpServletResponse`, but setting the content type is such a common task that there is also a special `setContentType` method just for this purpose. The way to designate HTML is with a type of `text/html`, so the code would look like this:

```
response.setContentType("text/html");
```

Although HTML is the most common type of document servlets create, it is not unusual to create other document types. For example, Section 7.5 (Using Servlets to Generate GIF Images) shows how servlets can build and return custom images, specifying a content type of `image/gif`. As a second example, Section 11.2 (The `contentType` Attribute) shows how to generate and return Excel spreadsheets, using a content type of `application/vnd.ms-excel`.

Don't be concerned if you are not yet familiar with HTTP response headers; they are discussed in detail in Chapter 7. Note that you need to set response headers *before* actually returning any of the content via the `PrintWriter`. That's because an HTTP response consists of the status line, one or more headers, a blank line, and the actual document, *in that order*. The headers can appear in any order, and servlets buffer the headers and send them all at once, so it is legal to set the status code (part of the first line returned) even after setting headers. But servlets do not necessarily buffer the document itself, since users might want to see partial results for long pages. In version 2.1 of the servlet specification, the `PrintWriter` output is not buffered at all, so the first time you use the `PrintWriter`, it is too late to go back and set headers. In version 2.2, servlet engines are permitted to partially buffer the output, but the size of the buffer is left unspecified. You can use the `getBufferSize` method of `HttpServletResponse` to determine the size, or use `setBufferSize` to specify it. In version 2.2 with buffering enabled, you can set headers until the buffer fills up and is actually sent to the client. If you aren't sure if the buffer has been sent, you can use the `isCommitted` method to check.

Core Approach

*Always set the content type **before** transmitting the actual document.*



The second step in writing a servlet that builds an HTML document is to have your `println` statements output HTML, not plain text. The structure of an HTML document is discussed more in Section 2.5 (Simple HTML-Building Utilities), but it should be familiar to most readers. Listing 2.3 gives an example servlet, with the result shown in Figure 2–2.

Listing 2.3 HelloWWW.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
            \"Transitional//EN\">\n";
        out.println(docType +
            "<HTML>\n" +
            "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
            "<BODY>\n" +
            "<H1>Hello WWW</H1>\n" +
            "</BODY></HTML>");
    }
}
```

2.4 Packaging Servlets

In a production environment, multiple programmers may be developing servlets for the same server. So, placing all the servlets in the top-level servlet directory results in a massive hard-to-manage directory and risks name conflicts when two developers accidentally choose the same servlet name. Packages are the natural solution to this problem. Using packages results in changes in the way the servlets are created, the way that they are compiled,

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

Chapter 2 First Servlets**Figure 2-2** Result of Listing 2.3 (HelloWWW.java).

and the way they're invoked. Let's take these areas one at a time in the following three subsections. The first two changes are exactly the same as with any other Java class that uses packages; there is nothing specific to servlets.

Creating Servlets in Packages

Two steps are needed to place servlets in packages:

1. **Move the files to a subdirectory that matches the intended package name.**

For example, I'll use the `coreservlets` package for most of the rest of the servlets in this book. So, the class files need to go in a subdirectory called `coreservlets`.

2. **Insert a package statement in the class file.**

For example, to place a class file in a package called `somePackage`, the *first* line of the file should read

```
package somePackage;
```

For example, Listing 2.4 presents a variation of the `HelloWWW` servlet that is in the `coreservlets` package. The class file goes in `install_dir/webpages/WEB-INF/classes/coreservlets` for Tomcat 3.0, in `install_dir/webpages/WEB-INF/servlets/coreservlets` for the JSDK 1.0.1, and in `install_dir/servlets/coreservlets` for the Java Web Server 2.0.

Listing 2.4 HelloWWW2.java

```

package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW2 extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String docType =
            "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 \" +
            \"Transitional//EN\">\n";
        out.println(docType +
            "<HTML>\n" +
            "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
            "<BODY>\n" +
            "<H1>Hello WWW</H1>\n" +
            "</BODY></HTML>");
    }
}

```

Compiling Servlets in Packages

There are two main ways to compile classes that are in packages. The first option is to place your package subdirectory right in the directory where the Web server expects servlets to go. Then, you would set the CLASSPATH variable to point to the directory *above* the one actually containing your servlets, that is, to the main servlet directory used by the Web server. You can then compile normally from within the package-specific subdirectory. For example, if your base servlet directory is C:\JavaWebServer2.0\servlets and your package name (and thus subdirectory name) is coreservlets, and you are running Windows, you would do:

```

DOS> set CLASSPATH=C:\JavaWebServer2.0\servlets;%CLASSPATH%
DOS> cd C:\JavaWebServer2.0\servlets\coreservlets
DOS> javac HelloWorld.java

```

The first part, setting the CLASSPATH, you probably want to do permanently, rather than each time you start a new DOS window. On Windows 95/98 you typically put the set CLASSPATH=... statement in your autoexec.bat file somewhere *after* the line that sets the CLASSPATH to point to servlet.jar

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

Chapter 2 First Servlets

and the JSP JAR file. On Windows NT or Windows 2000, you go to the Start menu, select Settings, select Control Panel, select System, select Environment, then enter the variable and value. On Unix (C shell), you set the `CLASSPATH` variable by

```
setenv CLASSPATH /install_dir/servlets:$CLASSPATH
```

Put this in your `.cshrc` file to make it permanent.

If your package were of the form `name1.name2.name3` rather than simply `name1` as here, the `CLASSPATH` should still point to the top-level servlet directory, that is, the directory containing `name1`.

A second way to compile classes that are in packages is to keep the source code in a location distinct from the class files. First, you put your package directories in any location you find convenient. The `CLASSPATH` refers to this location. Second, you use the `-d` option of `javac` to install the class files in the directory the Web server expects. An example follows. Again, you will probably want to set the `CLASSPATH` permanently rather than set it each time.

```
DOS> cd C:\MyServlets\coreervlets
DOS> set CLASSPATH=C:\MyServlets;%CLASSPATH%
DOS> javac -d C:\tomcat\webpages\WEB-INF\classes HelloWWW2.java
```

Keeping the source code separate from the class files is the approach I use for my own development. To complicate my life further, I have a number of different `CLASSPATH` settings that I use for different projects, and typically use JDK 1.2, not JDK 1.1 as the Java Web Server expects. So, on Windows I find it convenient to automate the servlet compilation process with a batch file `servletc.bat`, as shown in Listing 2.5 (line breaks in the `set CLASSPATH` line inserted only for readability). I put this batch file in `C:\Windows\Command` or somewhere else in the Windows `PATH`. After this, to compile the `HelloWWW2` servlet and install it with the Java Web Server, I merely go to `C:\MyServlets\coreervlets` and do “`servletc HelloWWW2.java`”. The source code archive at <http://www.coreservlets.com/> contains variations of `servletc.bat` for the JSWDK and Tomcat. You can do something similar on Unix with a shell script.

Invoking Servlets in Packages

To invoke a servlet that is in a package, use the URL

```
http://host/servlet/packageName.ServletName
```

instead of

```
http://host/servlet/ServletName
```

Thus, if the Web server is running on the local system,

Listing 2.5 servletc.bat

```
@echo off

rem This is the version for the Java Web Server.
rem See http://www.coreservlets.com/ for other versions.

set CLASSPATH=C:\JavaWebServer2.0\lib\servlet.jar;
             C:\JavaWebServer2.0\lib\jsp.jar;
             C:\MyServlets
C:\JDK1.1.8\bin\javac -d C:\JavaWebServer2.0\servlets %1%
```

`http://localhost/servlet/coreservlets.HelloWWW2`

would invoke the `HelloWWW2` servlet, as illustrated in Figure 2-3.

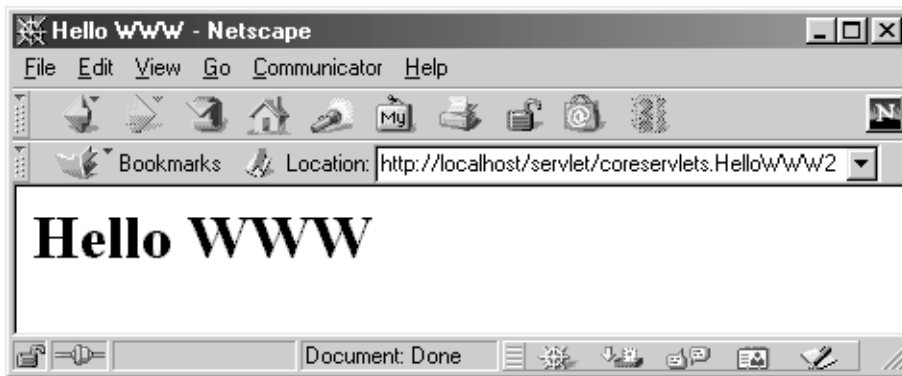


Figure 2-3 Invoking a servlet in a package via `http://hostname/servlet/packageName.servletName`.

2.5 Simple HTML-Building Utilities

An HTML document is structured as follows:

```
<!DOCTYPE ...>
<HTML>
<HEAD><TITLE>...</TITLE>...</HEAD>
<BODY ...>
...
</BODY>
</HTML>
```

You might be tempted to omit part of this structure, especially the `DOCTYPE` line, noting that virtually all major browsers ignore it, even though the

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

HTML 3.2 and 4.0 specifications require it. I strongly discourage this practice. The advantage of the `DOCTYPE` line is that it tells HTML validators which version of HTML you are using, so they know which specification to check your document against. These validators are very valuable debugging services, helping you catch HTML syntax errors that your browser guesses well on, but that other browsers will have trouble displaying. The two most popular on-line validators are the ones from the World Wide Web Consortium (<http://validator.w3.org/>) and from the Web Design Group (<http://www.htmlhelp.com/tools/validator/>). They let you submit a URL, then they retrieve the page, check the syntax against the formal HTML specification, and report any errors to you. Since a servlet that generates HTML looks like a regular Web page to visitors, it can be validated in the normal manner unless it requires `POST` data to return its result. Remember that `GET` data is attached to the URL, so you can submit a URL that includes `GET` data to the validators.



Core Approach

Use an HTML validator to check the syntax of pages that your servlets generate.

Admittedly it is a bit cumbersome to generate HTML with `println` statements, especially long tedious lines like the `DOCTYPE` declaration. Some people address this problem by writing detailed HTML generation utilities in Java, then use them throughout their servlets. I'm skeptical of the utility of an extensive library for this. First and foremost, the inconvenience of generating HTML programmatically is one of the main problems addressed by JavaServer Pages (discussed in the second part of this book). JSP is a better solution, so don't waste effort building a complex HTML generation package. Second, HTML generation routines can be cumbersome and tend not to support the full range of HTML attributes (`CLASS` and `ID` for style sheets, JavaScript event handlers, table cell background colors, and so forth). Despite the questionable value of a full-blown HTML generation library, if you find you're repeating the same constructs many times, you might as well create a simple utility file that simplifies those constructs. For standard servlets, there are two parts of the Web page (`DOCTYPE` and `HEAD`) that are unlikely to change and thus could benefit from being incorporated into a simple utility file. These are shown in List-

ing 2.6, with Listing 2.7 showing a variation of `HelloWWW2` that makes use of this utility. I'll add a few more utilities throughout the book.

Listing 2.6 `ServletUtilities.java`

```
package coreservlets;

import javax.servlet.*;
import javax.servlet.http.*;

public class ServletUtilities {
    public static final String DOCTYPE =
        "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
        "Transitional//EN">";

    public static String headWithTitle(String title) {
        return(DOCTYPE + "\n" +
            "<HTML>\n" +
            "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n");
    }
    ...
}
```

Listing 2.7 `HelloWWW3.java`

```
package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWWW3 extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println(ServletUtilities.headWithTitle("Hello WWW") +
            "<BODY>\n" +
            "<H1>Hello WWW</H1>\n" +
            "</BODY></HTML>");
    }
}
```

Chapter 2 First Servlets**Figure 2-4** Result of the HelloWWW3 servlet.

2.6 The Servlet Life Cycle

Earlier in this book, I vaguely referred to the fact that only a single instance of a servlet gets created, with each user request resulting in a new thread that is handed off to `doGet` or `doPost` as appropriate. I'll now be more specific about how servlets are created and destroyed, and how and when the various methods are invoked. I'll give a quick summary here, then elaborate in the following subsections.

When the servlet is first created, its `init` method is invoked, so that is where you put one-time setup code. After this, each user request results in a thread that calls the `service` method of the previously created instance. Multiple concurrent requests normally result in multiple threads calling `service` simultaneously, although your servlet can implement a special interface that stipulates that only a single thread is permitted to run at any one time. The `service` method then calls `doGet`, `doPost`, or another `doXxx` method, depending on the type of HTTP request it received. Finally, when the server decides to unload a servlet, it first calls the servlet's `destroy` method.

The init Method

The `init` method is called when the servlet is first created and is *not* called again for each user request. So, it is used for one-time initializations, just as with the `init` method of applets. The servlet can be created when a user first invokes a URL corresponding to the servlet or when the server is first started,

depending on how you have registered the servlet with the Web server. It will be created for the first user request if it is not explicitly registered but is instead just placed in one of the standard server directories. See the discussion of Section 2.2 (A Simple Servlet Generating Plain Text) for details on these directories.

There are two versions of `init`: one that takes no arguments and one that takes a `ServletConfig` object as an argument. The first version is used when the servlet does not need to read any settings that vary from server to server. The method definition looks like this:

```
public void init() throws ServletException {
    // Initialization code...
}
```

For examples of this type of initialization, see Section 2.8 (An Example Using Servlet Initialization and Page Modification Dates) later in this chapter. Section 18.8 (Connection Pooling: A Case Study) in the chapter on JDBC gives a more advanced application where `init` is used to preallocate multiple database connections.

The second version of `init` is used when the servlet needs to read server-specific settings before it can complete the initialization. For example, the servlet might need to know about database settings, password files, server-specific performance parameters, hit count files, or serialized cookie data from previous requests. The second version of `init` looks like this:

```
public void init(ServletConfig config)
    throws ServletException {
    super.init(config);
    // Initialization code...
}
```

Notice two things about this code. First, the `init` method takes a `ServletConfig` as an argument. `ServletConfig` has a `getInitParameter` method with which you can look up initialization parameters associated with the servlet. Just as with the `getParameter` method used in the `init` method of applets, both the input (the parameter name) and the output (the parameter value) are strings. For a simple example of the use of initialization parameters, see Section 2.7 (An Example Using Initialization Parameters); for a more complex example, see Section 4.5 (Restricting Access to Web Pages) where the location of a password file is given through the use of `getInitParameter`. Note that although you *look up* parameters in a portable manner, you *set* them in a server-specific way. For example, with Tomcat, you embed servlet properties in a file called `web.xml`, with the JSWDK you use `servlets.properties`, with the WebLogic application server you use `weblogic.properties`, and with the Java Web Server you set the properties interactively via the administration

Chapter 2 First Servlets

console. For examples of these settings, see Section 2.7 (An Example Using Initialization Parameters).

The second thing to note about the second version of `init` is that the first line of the method body is a call to `super.init`. This call is critical! The `ServletConfig` object is used elsewhere in the servlet, and the `init` method of the superclass registers it where the servlet can find it later. So, you can cause yourself huge headaches later if you omit the `super.init` call.



Core Approach

*If you write an `init` method that takes a `ServletConfig` as an argument, **always** call `super.init` on the first line.*

The service Method

Each time the server receives a request for a servlet, the server spawns a new thread and calls `service`. The `service` method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls `doGet`, `doPost`, `doPut`, `doDelete`, etc., as appropriate. Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override `service` directly as below, rather than implementing both `doGet` and `doPost`.

```
public void service(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet Code
}
```

This is not a good idea. Instead, just have `doPost` call `doGet` (or vice versa), as below.

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    // Servlet Code
}

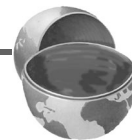
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
```

Although this approach takes a couple of extra lines of code, it has five advantages over directly overriding `service`:

1. You can add support for other services later by adding `doPut`, `doTrace`, etc., perhaps in a subclass. Overriding `service` directly precludes this possibility.
2. You can add support for modification dates by adding a `getLastModified` method. If you use `doGet`, the standard `service` method uses the `getLastModified` method to set `Last-Modified` headers and to respond properly to conditional GET requests (those containing an `If-Modified-Since` header). See Section 2.8 (An Example Using Servlet Initialization and Page Modification Dates) for an example.
3. You get automatic support for HEAD requests. The system just returns whatever headers and status codes `doGet` sets, but omits the page body. HEAD is a useful request method for custom HTTP clients. For example, link validators that check a page for dead hypertext links often use HEAD instead of GET in order to reduce server load.
4. You get automatic support for OPTIONS requests. If a `doGet` method exists, the standard `service` method answers OPTIONS requests by returning an `Allow` header indicating that GET, HEAD, OPTIONS, and TRACE are supported.
5. You get automatic support for TRACE requests. TRACE is a request method used for client debugging; it just returns the HTTP request headers back to the client.

Core Tip

If your servlet needs to handle both GET and POST identically, have your `doPost` method call `doGet`, or vice versa. Don't override `service` directly.



The `doGet`, `doPost`, and `doXxx` Methods

These methods contain the real meat of your servlet. Ninety-nine percent of the time, you only care about GET and/or POST requests, so you override `doGet` and/or `doPost`. However, if you want to, you can also override `doDelete` for DELETE requests, `doPut` for PUT, `doOptions` for OPTIONS, and `doTrace` for TRACE. Recall, however, that you have automatic support for OPTIONS and TRACE, as described in the previous section on the `service` method. Note that there is no `doHead` method. That's because the system

automatically uses the status line and header settings of `doGet` to answer HEAD requests.

The SingleThreadModel Interface

Normally, the system makes a single instance of your servlet and then creates a new thread for each user request, with multiple simultaneous threads running if a new request comes in while a previous request is still executing. This means that your `doGet` and `doPost` methods must be careful to synchronize access to fields and other shared data, since multiple threads may be trying to access the data simultaneously. See Section 7.3 (Persistent Servlet State and Auto-Reloading Pages) for more discussion of this. If you want to prevent this multithreaded access, you can have your servlet implement the `SingleThreadModel` interface, as below.

```
public class YourServlet extends HttpServlet
    implements SingleThreadModel {
    ...
}
```

If you implement this interface, the system guarantees that there is never more than one request thread accessing a single instance of your servlet. It does so either by queuing up all the requests and passing them one at a time to a single servlet instance, or by creating a pool of multiple instances, each of which handles one request at a time. This means that you don't have to worry about simultaneous access to regular fields (instance variables) of the servlet. You *do*, however, still have to synchronize access to class variables (static fields) or shared data stored outside the servlet.

Synchronous access to your servlets can significantly hurt performance (latency) if your servlet is accessed extremely frequently. So think twice before using the `SingleThreadModel` approach.

The destroy Method

The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator, or perhaps because the servlet is idle for a long time. Before it does, however, it calls the servlet's `destroy` method. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities. Be aware, however, that it is possible for the Web server to crash. After all, not all Web servers are written in reliable programming languages like Java; some are written

in languages (such as ones named after letters of the alphabet) where it is easy to read or write off the ends of arrays, make illegal typecasts, or have dangling pointers due to memory reclamation errors. Besides, even Java technology won't prevent someone from tripping over the power cable running to the computer. So, don't count on `destroy` as the only mechanism for saving state to disk. Activities like hit counting or accumulating lists of cookie values that indicate special access should also proactively write their state to disk periodically.

2.7 An Example Using Initialization Parameters

Listing 2.8 shows a servlet that reads the message and repeats initialization parameters when initialized. Figure 2-5 shows the result when message is Shibboleth, repeats is 5, and the servlet is registered under the name ShowMsg. Remember that, although servlets *read* init parameters in a standard way, developers *set* init parameters in a server-specific manner. Please refer to your server documentation for authoritative details. Listing 2.9 shows the configuration file used with Tomcat to obtain the result of Figure 2-5, Listing 2.10 shows the configuration file used with the JSDK, and Figures 2-6 and 2-7 show how to set the parameters interactively with the Java Web Server. The result is identical to Figure 2-5 in all three cases.

Because the process of setting init parameters is server-specific, it is a good idea to minimize the number of separate initialization entries that have to be specified. This will limit the work you need to do when moving servlets that use init parameters from one server to another. If you need to read a large amount of data, I recommend that the init parameter itself merely give the location of a parameter file, and that the real data go in that file. An example of this approach is given in Section 4.5 (Restricting Access to Web Pages), where the initialization parameter specifies nothing more than the location of the password file.

Core Approach

For complex initializations, store the data in a separate file and use the init parameters to give the location of that file.



Chapter 2 First Servlets**Listing 2.8 ShowMessage.java**

```

package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Example using servlet initialization. Here, the message
 *  to print and the number of times the message should be
 *  repeated is taken from the init parameters.
 */

public class ShowMessage extends HttpServlet {
    private String message;
    private String defaultMessage = "No message.";
    private int repeats = 1;

    public void init(ServletConfig config)
        throws ServletException {
        // Always call super.init
        super.init(config);
        message = config.getInitParameter("message");
        if (message == null) {
            message = defaultMessage;
        }
        try {
            String repeatString = config.getInitParameter("repeats");
            repeats = Integer.parseInt(repeatString);
        } catch (NumberFormatException nfe) {
            // NumberFormatException handles case where repeatString
            // is null *and* case where it is something in an
            // illegal format. Either way, do nothing in catch,
            // as the previous value (1) for the repeats field will
            // remain valid because the Integer.parseInt throws
            // the exception *before* the value gets assigned
            // to repeats.
        }
    }

    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "The ShowMessage Servlet";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>");
    }
}

```

2.7 An Example Using Initialization Parameters

41

Listing 2.8 ShowMessage.java (continued)

```
        for(int i=0; i<repeats; i++) {
            out.println(message + "<BR>");
        }
        out.println("</BODY></HTML>");
    }
}
```



Figure 2-5 The ShowMessage servlet with server-specific initialization parameters.

Listing 2.9 shows the setup file used to supply initialization parameters to servlets used with Tomcat 3.0. The idea is that you first associate a name with the servlet class file, then associate initialization parameters with that name (not with the actual class file). The setup file is located in *install_dir/webpages/WEB-INF*. Rather than recreating a similar version by hand, you might want to download this file from <http://www.coreservlets.com/>, modify it, and copy it to *install_dir/webpages/WEB-INF*.

Listing 2.10 shows the properties file used to supply initialization parameters to servlets in the JSWDK. As with Tomcat, you first associate a name with the servlet class, then associate the initialization parameters with the name. The properties file is located in *install_dir/webpages/WEB-INF*.

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

Chapter 2 First Servlets**Listing 2.9 web.xml (for Tomcat)**

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!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>
  <servlet>
    <servlet-name>
      ShowMsg
    </servlet-name>

    <servlet-class>
      coreservlets.ShowMessage
    </servlet-class>

    <init-param>
      <param-name>
        message
      </param-name>
      <param-value>
        Shibboleth
      </param-value>
    </init-param>

    <init-param>
      <param-name>
        repeats
      </param-name>
      <param-value>
        5
      </param-value>
    </init-param>
  </servlet>
</web-app>
```

2.7 An Example Using Initialization Parameters

43

Listing 2.10 servlets.properties

```
# servlets.properties used with the JSWDK

# Register servlet via servletName.code=servletClassFile
# You access it via http://host/examples/servlet/servletName
ShowMsg.code=coreservlets.ShowMessage

# Set init params via
#   servletName.initparams=param1=val1,param2=val2,...
ShowMsg.initparams=message=Shibboleth,repeats=5

# Standard setting
jsp.code=com.sun.jsp.runtime.JspServlet

# Set this to keep servlet source code built from JSP
jsp.initparams=keepgenerated=true
```

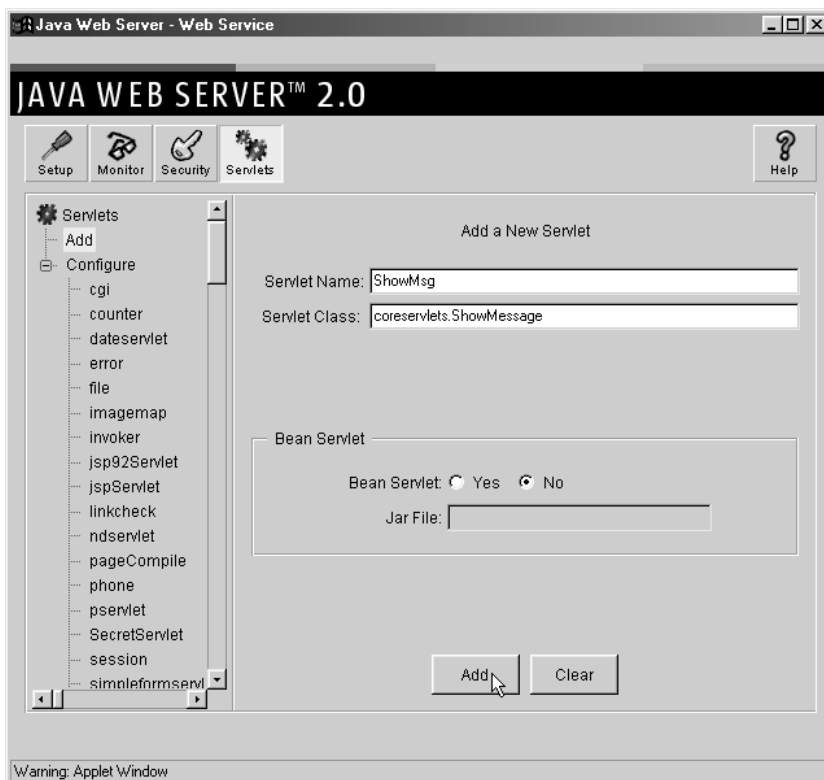


Figure 2-6 Registering a name for a servlet with the Java Web Server. Servlets that use initialization parameters must first be registered this way.

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

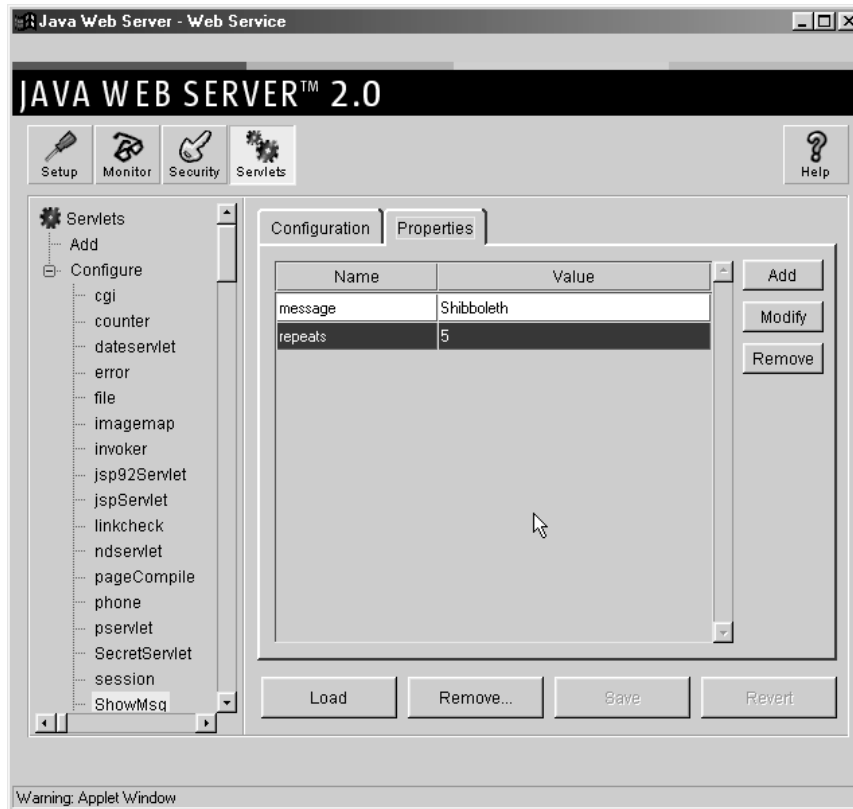
Chapter 2 First Servlets

Figure 2-7 Specifying initialization parameters for a named servlet with the Java Web Server.

2.8 An Example Using Servlet Initialization and Page Modification Dates

Listing 2.11 shows a servlet that uses `init` to do two things. First, it builds an array of 10 integers. Since these numbers are based upon complex calculations, I don't want to repeat the computation for each request. So I have `doGet` look up the values that `init` computed instead of generating them each time. The results of this technique are shown in Figure 2-8.

However, since all users get the same result, `init` also stores a page modification date that is used by the `getLastModified` method. This method should return a modification time expressed in milliseconds since 1970, as is standard

2.8 An Example Using Servlet Initialization and Page Modification Dates

45

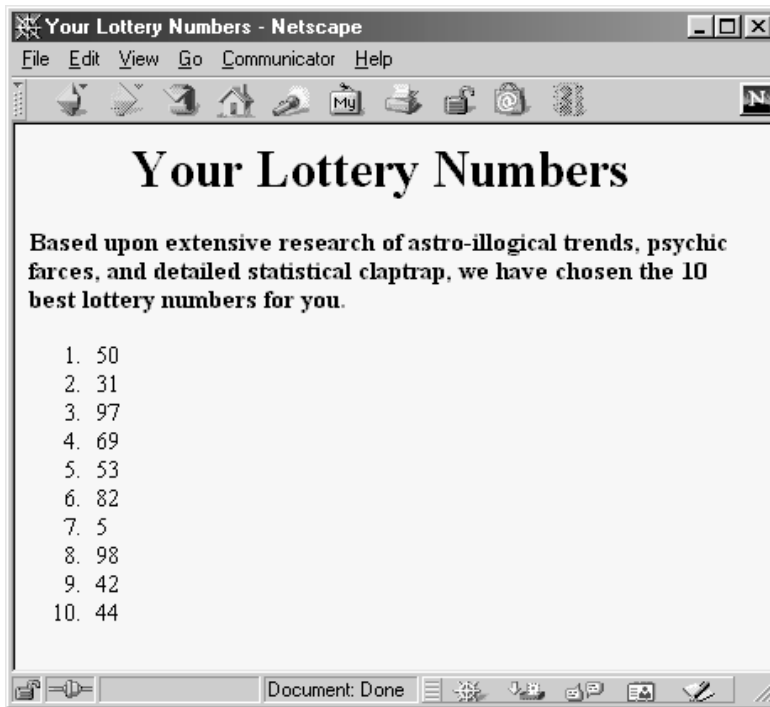


Figure 2-8 Output of LotteryNumbers servlet.

with Java dates. The time is automatically converted to a date in GMT appropriate for the `Last-Modified` header. More importantly, if the server receives a conditional GET request (one specifying that the client only wants pages marked `If-Modified-Since` a particular date), the system compares the specified date to that returned by `getLastModified`, only returning the page if it has been changed after the specified date. Browsers frequently make these conditional requests for pages stored in their caches, so supporting conditional requests helps your users as well as reducing server load. Since the `Last-Modified` and `If-Modified-Since` headers use only whole seconds, the `getLastModified` method should round times down to the nearest second.

Figures 2-9 and 2-10 show the result of requests for the same servlet with two slightly different `If-Modified-Since` dates. To set the request headers and see the response headers, I used `WebClient`, a Java application shown in Section 2.10 (`WebClient: Talking to Web Servers Interactively`) that lets you interactively set up HTTP requests, submit them, and see the results.

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

Chapter 2 First Servlets**Listing 2.11 LotteryNumbers.java**

```

package coreservlets;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

/** Example using servlet initialization and the
 *  getLastModified method.
 */

public class LotteryNumbers extends HttpServlet {
    private long modTime;
    private int[] numbers = new int[10];

    /** The init method is called only when the servlet
     *  is first loaded, before the first request
     *  is processed.
     */

    public void init() throws ServletException {
        // Round to nearest second (ie 1000 milliseconds)
        modTime = System.currentTimeMillis()/1000*1000;
        for(int i=0; i<numbers.length; i++) {
            numbers[i] = randomNum();
        }
    }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Your Lottery Numbers";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +

```

2.8 An Example Using Servlet Initialization and Page Modification Dates

47

Listing 2.11 LotteryNumbers.java (continued)

```
        "<B>Based upon extensive research of " +
        "astro-illogical trends, psychic farces, " +
        "and detailed statistical claptrap, " +
        "we have chosen the " + numbers.length +
        " best lottery numbers for you.</B>" +
        "<OL>");
    for(int i=0; i<numbers.length; i++) {
        out.println("  <LI>" + numbers[i]);
    }
    out.println("</OL>" +
        "</BODY></HTML>");
}

/** The standard service method compares this date
 *  against any date specified in the If-Modified-Since
 *  request header. If the getLastModified date is
 *  later, or if there is no If-Modified-Since header,
 *  the doGet method is called normally. But if the
 *  getLastModified date is the same or earlier,
 *  the service method sends back a 304 (Not Modified)
 *  response, and does <B>not</B> call doGet.
 *  The browser should use its cached version of
 *  the page in such a case.
 */

public long getLastModified(HttpServletRequest request) {
    return(modTime);
}

// A random int from 0 to 99.

private int randomNum() {
    return((int)(Math.random() * 100));
}
}
```

Chapter 2 First Servlets

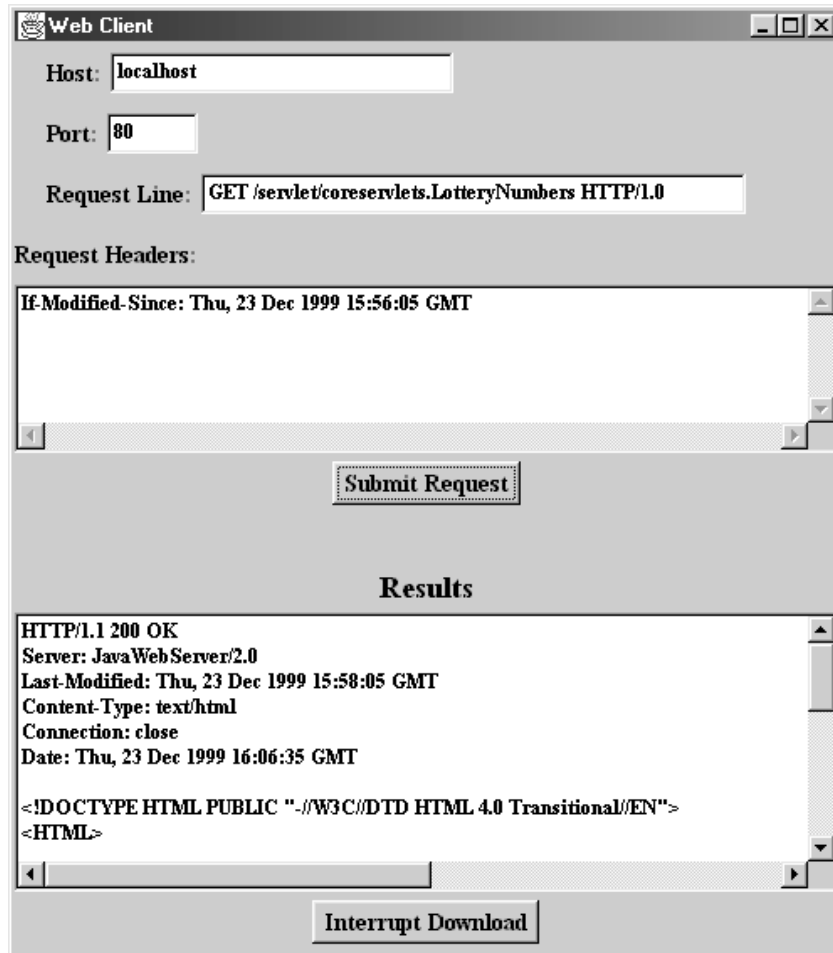


Figure 2-9 Accessing the `LotteryNumbers` servlet with an unconditional GET request or with a conditional request specifying a date before servlet initialization results in the normal Web page.

2.8 An Example Using Servlet Initialization and Page Modification Dates

49

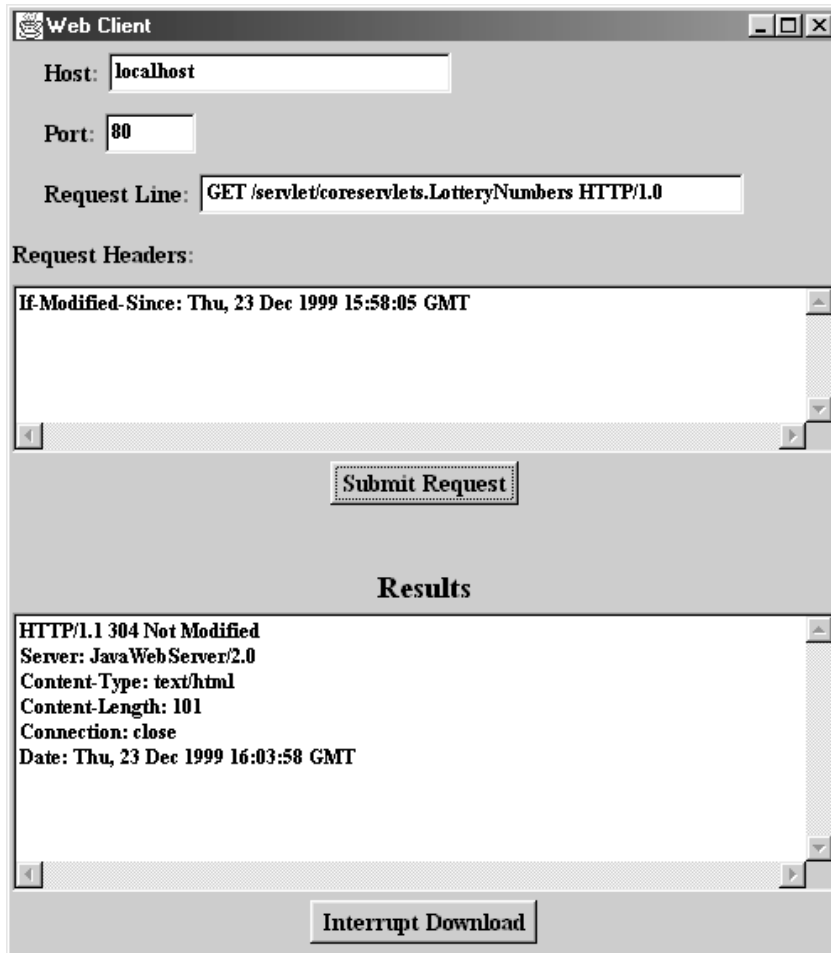


Figure 2-10 Accessing the `LotteryNumbers` servlet with a conditional GET request specifying a date at or after servlet initialization results in a 304 (Not Modified) response.

2.9 Debugging Servlets

Naturally, when *you* write servlets, you never make mistakes. However, some of your colleagues might make an occasional error, and you can pass this advice on to them. Seriously, though, debugging servlets can be tricky because you don't execute them directly. Instead, you trigger their execution by means of an HTTP request, and they are executed by the Web server. This remote execution makes it difficult to insert break points or to read debugging messages and stack traces. So, approaches to servlet debugging differ somewhat from those used in general development. Here are seven general strategies that can make your life easier.

1. **Look at the HTML source.**

If the result you see in the browser looks funny, choose "View Source" from the browser's menu. Sometimes a small HTML error like `<TABLE>` instead of `</TABLE>` can prevent much of the page from being viewed. Even better, use a formal HTML validator on the servlet's output. See Section 2.5 (Simple HTML-Building Utilities) for a discussion of this approach.

2. **Return error pages to the client.**

Sometimes certain classes of errors can be anticipated by the servlet. In these cases, the servlet should build descriptive information about the problem and return it to the client in a regular page or by means of the `sendError` method of `HttpServletResponse`. See Chapter 6 (Generating the Server Response: HTTP Status Codes) for details on `sendError`. For example, you should plan for cases when the client forgets some of the required form data and send an error page detailing what was missing. Error pages are not always possible, however. Sometimes something unexpected goes wrong with your servlet, and it simply crashes. The remaining approaches help you in those situations.

3. **Start the server from the command line.**

Most Web servers execute from a background process, and this process is often automatically started when the system is booted. If you are having trouble with your servlet, you should consider shutting down the server and restarting it from the command line. After this, `System.out.println` or `System.err.println` calls can be easily read from the window in which the server was started. When something goes wrong with

your servlet, your first task is to discover *exactly* how far the servlet got before it failed and to gather some information about the key data structures during the time period just before it failed. Simple `println` statements are surprisingly effective for this purpose. If you are running your servlets on a server that you cannot easily halt and restart, then do your debugging with the JSWDK, Tomcat, or the Java Web Server on your personal machine, and save deployment to the real server for later.

4. **Use the log file.**

The `HttpServlet` class has a method called `log` that lets you write information into a logging file on the server. Reading debugging messages from the log file is a bit less convenient than watching them directly from a window as with the previous approach, but using the log file does not require stopping and restarting the server. There are two variations of this method: one that takes a `String`, and the other that takes a `String` and a `Throwable` (an ancestor class of `Exception`). The exact location of the log file is server-specific, but it is generally clearly documented or can be found in subdirectories of the server installation directory.

5. **Look at the request data separately.**

Servlets read data from the HTTP request, construct a response, and send it back to the client. If something in the process goes wrong, you want to discover if it is because the client is sending the wrong data or because the servlet is processing it incorrectly. The `EchoServer` class, shown in Section 16.12 (A Debugging Web Server), lets you submit HTML forms and get a result that shows you *exactly* how the data arrived at the server.

6. **Look at the response data separately.**

Once you look at the request data separately, you'll want to do the same for the response data. The `WebClient` class, presented next in Section 2.10 (WebClient: Talking to Web Servers Interactively), permits you to connect to the server interactively, send custom HTTP request data, and see everything that comes back, HTTP response headers and all.

7. **Stop and restart the server.**

Most full-blown Web servers that support servlets have a designated location for servlets that are under development. Servlets in this location (e.g., the `servlets` directory for the Java Web Server) are supposed to be automatically reloaded when their associated class file changes. At times, however, some servers can

Chapter 2 First Servlets

get confused, especially when your only change is to a lower-level class, not to the top-level servlet class. So, if it appears that changes you make to your servlets are not reflected in the servlet's behavior, try restarting the server. With the JSWDK and Tomcat, you have to do this *every* time you make a change, since these mini-servers have no support for automatic servlet reloading.

2.10 WebClient: Talking to Web Servers Interactively

This section presents the source code for the `WebClient` program discussed in Section 2.9 (Debugging Servlets) and used in Section 2.8 (An Example Using Servlet Initialization and Page Modification Dates) and extensively throughout Chapter 16 (Using HTML Forms). As always, the source code can be downloaded from the on-line archive at <http://www.coreservlets.com/>, and there are no restrictions on its use.

WebClient

This class is the top-level program that you would use. Start it from the command line, then customize the HTTP request line and request headers, then press "Submit Request."

Listing 2.12 WebClient.java

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 * A graphical client that lets you interactively connect to
 * Web servers and send custom request lines and
 * request headers.
 */

public class WebClient extends CloseableFrame
    implements Runnable, Interruptible, ActionListener {
    public static void main(String[] args) {
        new WebClient("Web Client");
    }
}
```

Listing 2.12 WebClient.java (continued)

```
private LabeledTextField hostField, portField,
    requestLineField;
private TextArea requestHeadersArea, resultArea;
private String host, requestLine;
private int port;
private String[] requestHeaders = new String[30];
private Button submitButton, interruptButton;
private boolean isInterrupted = false;

public WebClient(String title) {
    super(title);
    setBackground(Color.lightGray);
    setLayout(new BorderLayout(5, 30));
    int fontSize = 14;
    Font labelFont =
        new Font("Serif", Font.BOLD, fontSize);
    Font headingFont =
        new Font("SansSerif", Font.BOLD, fontSize+4);
    Font textFont =
        new Font("Monospaced", Font.BOLD, fontSize-2);
    Panel inputPanel = new Panel();
    inputPanel.setLayout(new BorderLayout());
    Panel labelPanel = new Panel();
    labelPanel.setLayout(new GridLayout(4,1));
    hostField = new LabeledTextField("Host:", labelFont,
                                    30, textFont);
    portField = new LabeledTextField("Port:", labelFont,
                                    "80", 5, textFont);

    // Use HTTP 1.0 for compatibility with the most servers.
    // If you switch this to 1.1, you *must* supply a
    // Host: request header.
    requestLineField =
        new LabeledTextField("Request Line:", labelFont,
                            "GET / HTTP/1.0", 50, textFont);

    labelPanel.add(hostField);
    labelPanel.add(portField);
    labelPanel.add(requestLineField);
    Label requestHeadersLabel =
        new Label("Request Headers:");
    requestHeadersLabel.setFont(labelFont);
    labelPanel.add(requestHeadersLabel);
    inputPanel.add(labelPanel, BorderLayout.NORTH);
    requestHeadersArea = new TextArea(5, 80);
    requestHeadersArea.setFont(textFont);
    inputPanel.add(requestHeadersArea, BorderLayout.CENTER);
    Panel buttonPanel = new Panel();
    submitButton = new Button("Submit Request");
    submitButton.addActionListener(this);
    submitButton.setFont(labelFont);
    buttonPanel.add(submitButton);
```

Chapter 2 First Servlets**Listing 2.12 WebClient.java (continued)**

```

inputPanel.add(buttonPanel, BorderLayout.SOUTH);
add(inputPanel, BorderLayout.NORTH);
Panel resultPanel = new Panel();
resultPanel.setLayout(new BorderLayout());
Label resultLabel =
    new Label("Results", Label.CENTER);
resultLabel.setFont(headingFont);
resultPanel.add(resultLabel, BorderLayout.NORTH);
resultArea = new TextArea();
resultArea.setFont(textFont);
resultPanel.add(resultArea, BorderLayout.CENTER);
Panel interruptPanel = new Panel();
interruptButton = new Button("Interrupt Download");
interruptButton.addActionListener(this);
interruptButton.setFont(labelFont);
interruptPanel.add(interruptButton);
resultPanel.add(interruptPanel, BorderLayout.SOUTH);
add(resultPanel, BorderLayout.CENTER);
setSize(600, 700);
setVisible(true);
}

public void actionPerformed(ActionEvent event) {
    if (event.getSource() == submitButton) {
        Thread downloader = new Thread(this);
        downloader.start();
    } else if (event.getSource() == interruptButton) {
        isInterrupted = true;
    }
}

public void run() {
    isInterrupted = false;
    if (hasLegalArgs())
        new HttpClient(host, port, requestLine,
            requestHeaders, resultArea, this);
}

public boolean isInterrupted() {
    return(isInterrupted);
}

private boolean hasLegalArgs() {
    host = hostField.getTextField().getText();
    if (host.length() == 0) {
        report("Missing hostname");
        return(false);
    }
}

```

Listing 2.12 WebClient.java (continued)

```
String portString =
    portField.getTextField().getText();
if (portString.length() == 0) {
    report("Missing port number");
    return(false);
}
try {
    port = Integer.parseInt(portString);
} catch(NumberFormatException nfe) {
    report("Illegal port number: " + portString);
    return(false);
}
requestLine =
    requestLineField.getTextField().getText();
if (requestLine.length() == 0) {
    report("Missing request line");
    return(false);
}
getRequestHeaders();
return(true);
}

private void report(String s) {
    resultArea.setText(s);
}

private void getRequestHeaders() {
    for(int i=0; i<requestHeaders.length; i++)
        requestHeaders[i] = null;
    int headerNum = 0;
    String header =
        requestHeadersArea.getText();
    StringTokenizer tok =
        new StringTokenizer(header, "\r\n");
    while (tok.hasMoreTokens())
        requestHeaders[headerNum++] = tok.nextToken();
}
}
```

HttpClient

The `HttpClient` class does the real network communication. It simply sends the designated request line and request headers to the Web server, then reads the lines that come back one at a time, placing them into a `TextArea` until either the server closes the connection or the `HttpClient` is interrupted by means of the `isInterrupted` flag.

Home page for this book: www.coreservlets.com; Home page for sequel: www.moreservlets.com.
Servlet and JSP training courses by book's author: courses.coreservlets.com.

Chapter 2 First Servlets**Listing 2.13 HttpClient.java**

```

import java.awt.*;
import java.net.*;
import java.io.*;

/**
 * The underlying network client used by WebClient.
 */

public class HttpClient extends NetworkClient {
    private String requestLine;
    private String[] requestHeaders;
    private TextArea outputArea;
    private Interruptible app;

    public HttpClient(String host, int port,
                      String requestLine, String[] requestHeaders,
                      TextArea outputArea, Interruptible app) {
        super(host, port);
        this.requestLine = requestLine;
        this.requestHeaders = requestHeaders;
        this.outputArea = outputArea;
        this.app = app;
        if (checkHost(host))
            connect();
    }

    protected void handleConnection(Socket uriSocket)
        throws IOException {
        try {
            PrintWriter out = SocketUtil.getWriter(uriSocket);
            BufferedReader in = SocketUtil.getReader(uriSocket);
            outputArea.setText("");
            out.println(requestLine);
            for(int i=0; i<requestHeaders.length; i++) {
                if (requestHeaders[i] == null)
                    break;
                else
                    out.println(requestHeaders[i]);
            }
            out.println();
            String line;
            while ((line = in.readLine()) != null &&
                   !app.isInterrupted())
                outputArea.append(line + "\n");
            if (app.isInterrupted())
                outputArea.append("---- Download Interrupted ----");
        } catch (Exception e) {
            outputArea.setText("Error: " + e);
        }
    }
}

```

Listing 2.13 HttpClient.java (continued)

```
private boolean checkHost(String host) {
    try {
        InetAddress.getByNames(host);
        return(true);
    } catch(UnknownHostException uhe) {
        outputArea.setText("Bogus host: " + host);
        return(false);
    }
}
```

NetworkClient

The NetworkClient class is a generic starting point for network clients and is extended by HttpClient.

Listing 2.14 NetworkClient.java

```
import java.net.*;
import java.io.*;

/** A starting point for network clients. You'll need to
 *  * override handleConnection, but in many cases
 *  * connect can remain unchanged. It uses
 *  * SocketUtil to simplify the creation of the
 *  * PrintWriter and BufferedReader.
 *  *
 *  * @see SocketUtil
 */

public class NetworkClient {
    protected String host;
    protected int port;

    /** Register host and port. The connection won't
     *  * actually be established until you call
     *  * connect.
     *  *
     *  * @see #connect
     */

    public NetworkClient(String host, int port) {
        this.host = host;
        this.port = port;
    }
}
```

Chapter 2 First Servlets**Listing 2.14 NetworkClient.java (continued)**

```

/** Establishes the connection, then passes the socket
 * to handleConnection.
 *
 * @see #handleConnection
 */

public void connect() {
    try {
        Socket client = new Socket(host, port);
        handleConnection(client);
    } catch(UnknownHostException uhe) {
        System.out.println("Unknown host: " + host);
        uhe.printStackTrace();
    } catch(IOException ioe) {
        System.out.println("IOException: " + ioe);
        ioe.printStackTrace();
    }
}

/** This is the method you will override when
 * making a network client for your task.
 * The default version sends a single line
 * ("Generic Network Client") to the server,
 * reads one line of response, prints it, then exits.
 */

protected void handleConnection(Socket client)
    throws IOException {
    PrintWriter out =
        SocketUtil.getWriter(client);
    BufferedReader in =
        SocketUtil.getReader(client);
    out.println("Generic Network Client");
    System.out.println
        ("Generic Network Client:\n" +
         "Made connection to " + host +
         " and got '" + in.readLine() + "' in response");
    client.close();
}

/** The hostname of the server we're contacting. */

public String getHost() {
    return(host);
}

/** The port connection will be made on. */

public int getPort() {
    return(port);
}
}

```

SocketUtil

SocketUtil is a simple utility class that simplifies creating some of the streams used in network programming. It is used by `NetworkClient` and `HttpClient`.

Listing 2.15 SocketUtil.java

```
import java.net.*;
import java.io.*;

/** A shorthand way to create BufferedReaders and
 *  PrintWriters associated with a Socket.
 */

public class SocketUtil {
    /** Make a BufferedReader to get incoming data. */

    public static BufferedReader getReader(Socket s)
        throws IOException {
        return(new BufferedReader(
            new InputStreamReader(s.getInputStream())));
    }

    /** Make a PrintWriter to send outgoing data.
     *  This PrintWriter will automatically flush stream
     *  when println is called.
     */

    public static PrintWriter getWriter(Socket s)
        throws IOException {
        // 2nd argument of true means autoflush
        return(new PrintWriter(s.getOutputStream(), true));
    }
}
```


Chapter 2 First Servlets***CloseableFrame***

`CloseableFrame` is an extension of the standard `Frame` class, with the addition that user requests to quit the frame are honored. This is the top-level window on which `WebClient` is built.

Listing 2.16 `CloseableFrame.java`

```
import java.awt.*;
import java.awt.event.*;

/** A Frame that you can actually quit. Used as
 *  the starting point for most Java 1.1 graphical
 *  applications.
 */

public class CloseableFrame extends Frame {
    public CloseableFrame(String title) {
        super(title);
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    }

    /** Since we are doing something permanent, we need
     *  to call super.processWindowEvent <B>first</B>.
     */

    public void processWindowEvent(WindowEvent event) {
        super.processWindowEvent(event); // Handle listeners
        if (event.getID() == WindowEvent.WINDOW_CLOSING)
            System.exit(0);
    }
}
```

LabeledTextField

The LabeledTextField class is a simple combination of a TextField and a Label and is used in WebClient.

Listing 2.17 LabeledTextField.java

```
import java.awt.*;

/** A TextField with an associated Label.
 */

public class LabeledTextField extends Panel {
    private Label label;
    private TextField textField;

    public LabeledTextField(String labelString,
                           Font labelFont,
                           int textFieldSize,
                           Font textFont) {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        label = new Label(labelString, Label.RIGHT);
        if (labelFont != null)
            label.setFont(labelFont);
        add(label);
        textField = new TextField(textFieldSize);
        if (textFont != null)
            textField.setFont(textFont);
        add(textField);
    }

    public LabeledTextField(String labelString,
                           String textFieldString) {
        this(labelString, null, textFieldString,
             textFieldString.length(), null);
    }

    public LabeledTextField(String labelString,
                           int textFieldSize) {
        this(labelString, null, textFieldSize, null);
    }

    public LabeledTextField(String labelString,
                           Font labelFont,
                           String textFieldString,
                           int textFieldSize,
                           Font textFont) {
        this(labelString, labelFont,
             textFieldSize, textFont);
        textField.setText(textFieldString);
    }
}
```

Chapter 2 First Servlets**Listing 2.17 LabeledTextField.java (continued)**

```
/** The Label at the left side of the LabeledTextField.
 * To manipulate the Label, do:
 * <PRE>
 *   LabeledTextField ltf = new LabeledTextField(...);
 *   ltf.getLabel().someLabelMethod(...);
 * </PRE>
 *
 * @see #getTextField
 */

public Label getLabel() {
    return(label);
}

/** The TextField at the right side of the
 * LabeledTextField.
 *
 * @see #getLabel
 */

public TextField getTextField() {
    return(textField);
}
}
```

Interruptible

Interruptible is a simple interface used to identify classes that have an `isInterrupted` method. It is used by `HttpClient` to poll `WebClient` to see if the user has interrupted it.

Listing 2.18 `Interruptible.java`

```
/**
 * An interface for classes that can be polled to see
 * if they've been interrupted. Used by HttpClient
 * and WebClient to allow the user to interrupt a network
 * download.
 */

public interface Interruptible {
    public boolean isInterrupted();
}
```