3.6 The Servlet Life Cycle

In Section 1.4 (The Advantages of Servlets Over "Traditional" CGI) we referred to the fact that only a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. We'll now be more specific about how servlets are created and destroyed, and how and when the various methods are invoked. We summarize here, then elaborate in the following subsections.

When the servlet is first created, its init method is invoked, so init 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 (SingleThreadModel) that stipulates that only a single thread is permitted to run at any one time. The service method then calls doGet, doPost, or another do*Xxx* method, depending on the type of HTTP request it received. Finally, if the server decides to unload a servlet, it first calls the servlet's destroy method.

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. A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified. A POST request results from an HTML form that specifically lists POST as the METHOD. Other HTTP requests are generated only by custom clients. If you aren't familiar with HTML forms, see Chapter 19 (Creating and Processing HTML Forms).

Now, if you have a servlet that needs to handle both POST and GET requests identically, you may be tempted to override service directly rather than implementing both doGet and doPost. 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 several advantages over directly overriding service. First, you can later add support for other HTTP request methods by adding doPut, doTrace, etc., perhaps in a subclass. Overriding service directly precludes this possibility. Second, you can add support for modification dates by adding a getLastModified method, as illustrated in Listing 3.7. Since getLastModified is invoked by the default service method, overriding service eliminates this option. Finally, service gives you automatic support for HEAD, OPTION, and TRACE requests.

Core Approach

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

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 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.

Normally, you do not need to implement doHead in order to handle HEAD requests (HEAD requests stipulate that the server should return the normal HTTP headers, but no associated document). You don't normally need to implement doHead because the system automatically calls doGet and uses the resultant status line and header settings to answer HEAD requests. However, it is occasionally useful to implement doHead so that you can generate responses to HEAD requests (i.e., requests from custom clients that want just the HTTP headers, not the actual document) more quickly—without building the actual document output.

The init Method

Most of the time, your servlets deal only with per-request data, and doGet or doPost are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The init method is designed for this case; it is called when the servlet is first created, and *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 is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started (see the chapter on the web.xml file in Volume 2 of this book).

The init method definition looks like this:

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

The init method performs two varieties of initializations: general initializations and initializations controlled by initialization parameters.

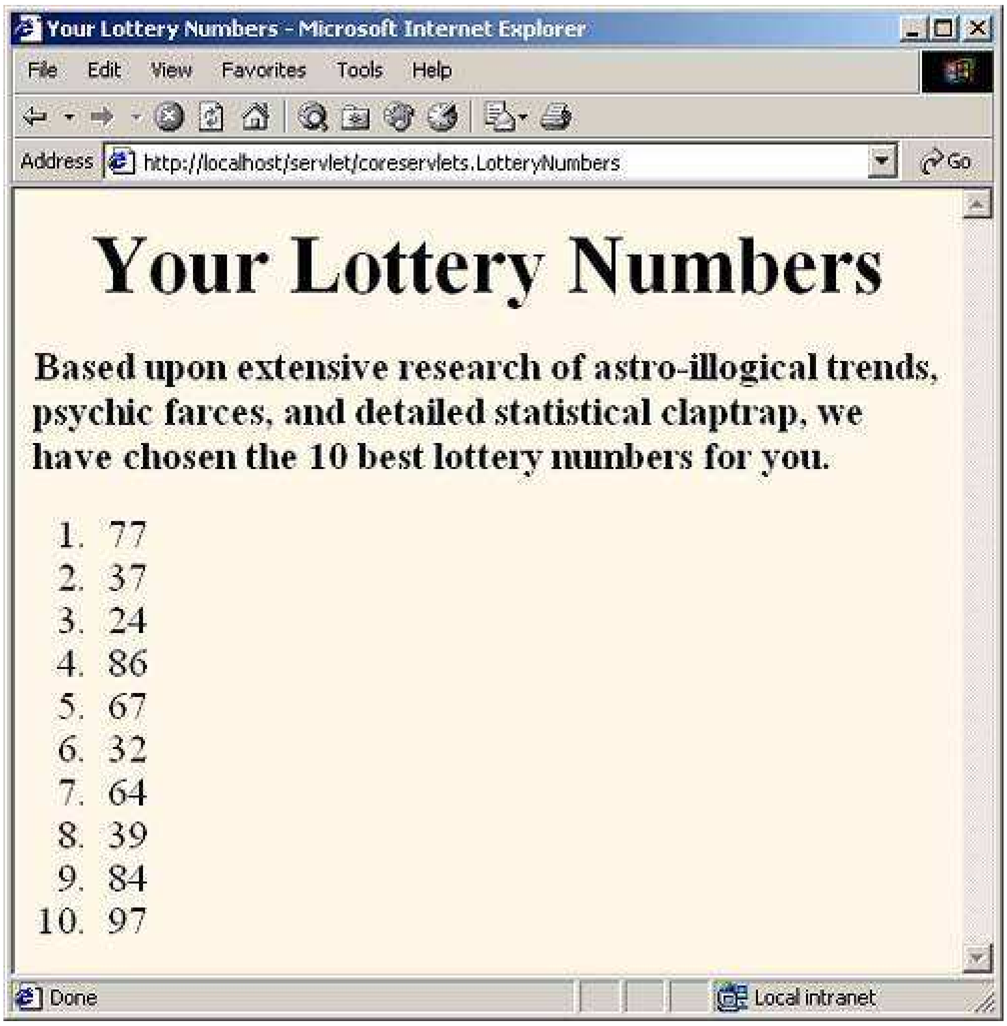
General Initializations

With the first type of initialization, init simply creates or loads some data that will be used throughout the life of the servlet, or it performs some one-time computation. If you are familiar with applets, this task is analogous to an applet calling getImage to load image files over the network: the operation only needs to be performed once, so it is triggered by init. Servlet examples include setting up a database connection pool for requests that the servlet will handle or loading a data file into a HashMap.

Listing 3.7 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, we don't want to repeat the computation for each request. So, doGet looks up the values that init computed, instead of generating them each time. The results of this technique are shown in Figure 3-6.

Figure 3-6. Result of the `LotteryNumbers` servlet.


Your Lottery Numbers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost/servlet/coreservlets.LotteryNumbers
Go
Done
Local intranet

Your Lottery Numbers

Based upon extensive research of astro-illogical trends, psychic farces, and detailed statistical claptrap, we have chosen the 10 best lottery numbers for you.

1. 77
2. 37
3. 24
4. 86
5. 67
6. 32
7. 64
8. 39
9. 84
10. 97

Second, since the output of the servlet does not change except when the server is rebooted, `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 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`, returning the page only 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 (they get faster results) and reduces server load (you send fewer complete documents). Since the `Last-Modified` and `If-Modified-Since` headers use only whole seconds, the `getLastModified` method should round times down to the nearest second.

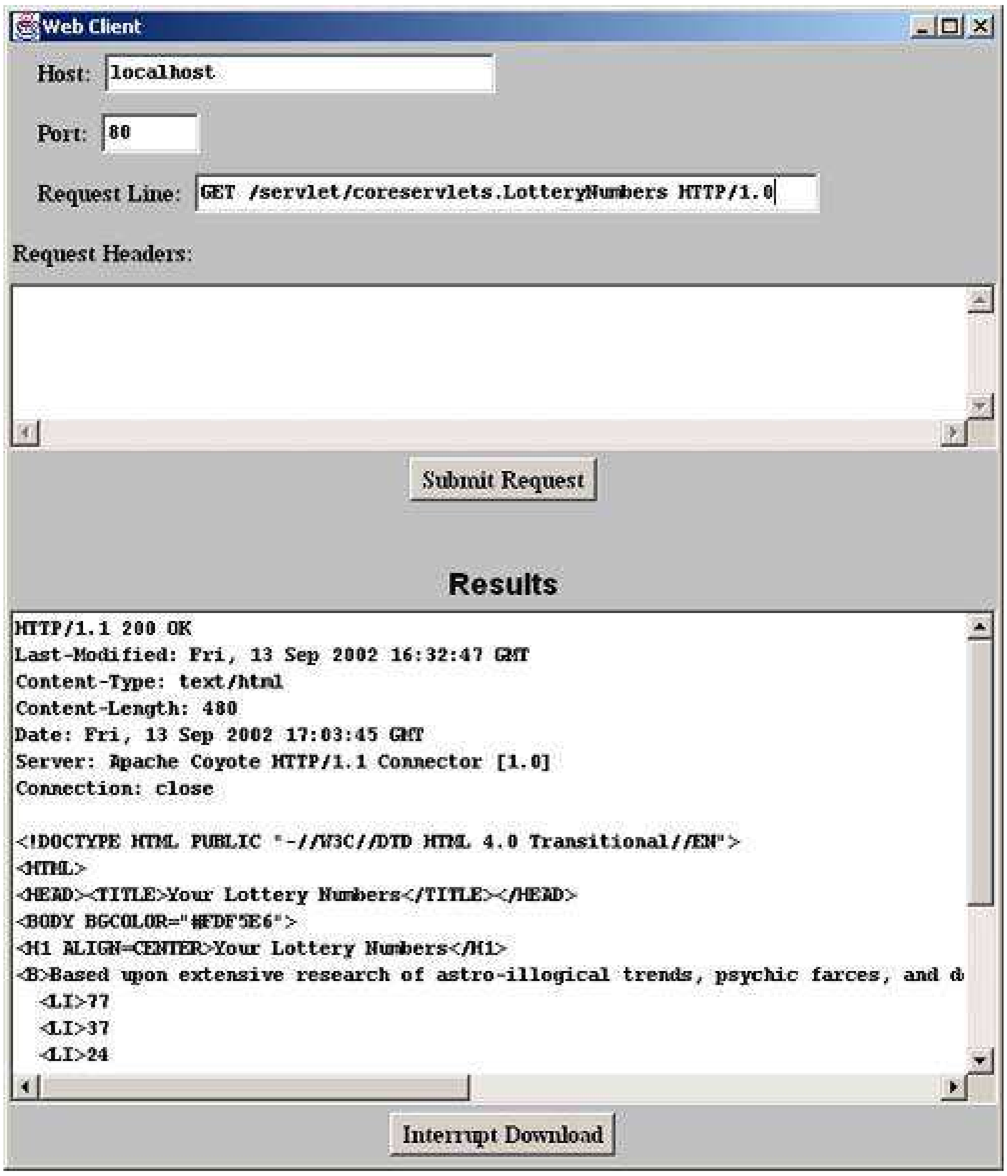
Listing 3.7 coreservlets/LotteryNumbers.java

```
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 (i.e., 1000 milliseconds)
    modTime = System.currentTimeMillis()/1000*1000;
    for(int i=0; i<numbers.length; i++) {
      numbers[i] = randomNum();
    }
  }

 /** Return the list of numbers that init computed. */

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Your Lottery Numbers";
    String docType =
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
      "Transitional//EN\">\n";
    out.println(docType +
                "<HTML>\n" +
                "<HEAD><TITLE>" + title + "</TITLE></HEAD>\n" +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                "<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));
  }
}
```

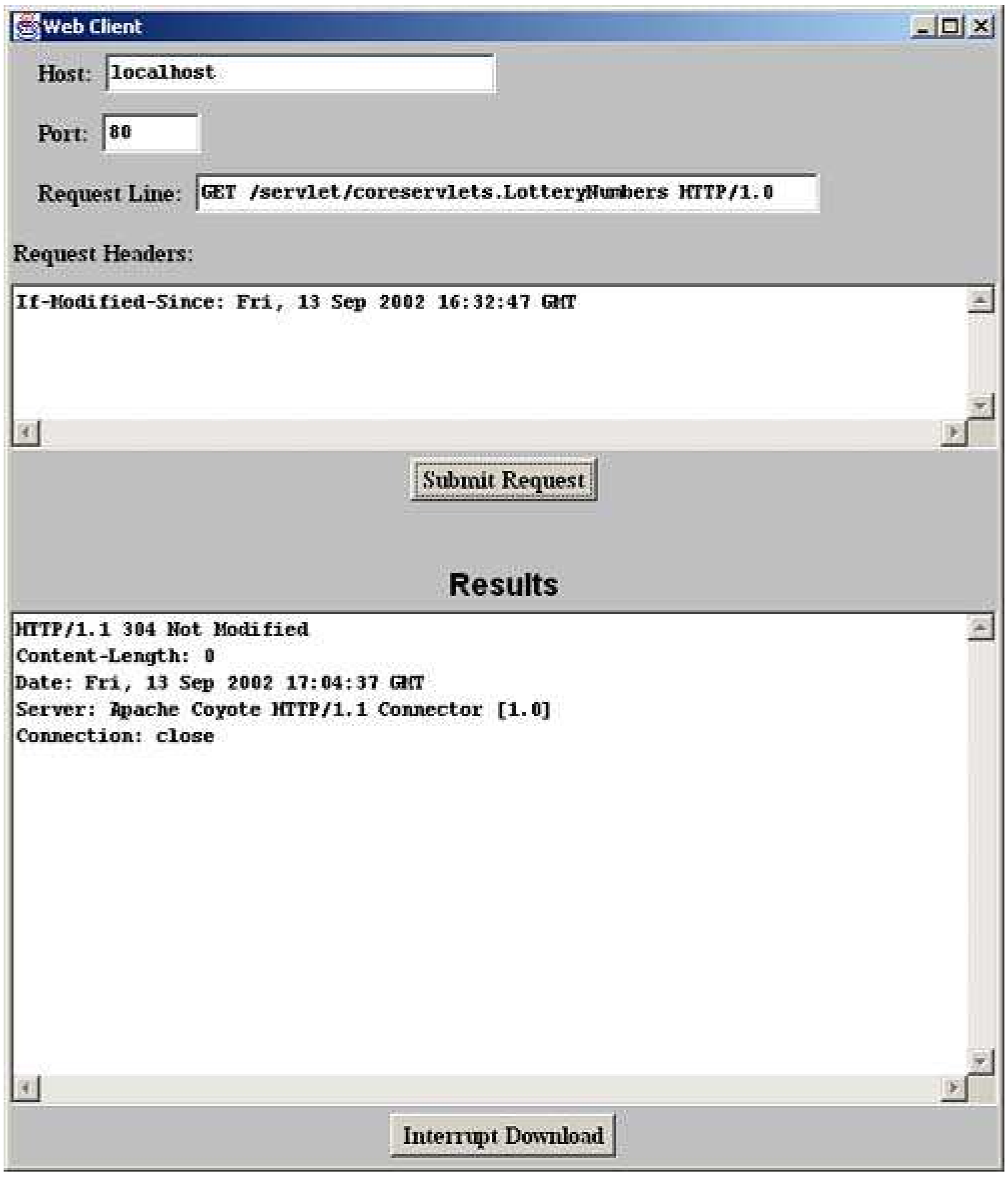
Figures 3-7 and 3-8 show the result of requests for the same servlet with two slightly different `If-Modified-Since` dates. To set the request headers and see the response headers, we used `WebClient`, a Java application that lets you interactively set up HTTP requests, submit them, and see the "raw" results. The code for `WebClient` is available at the source code archive on the book's home page (http://www.coreservlets.com/).

Figure 3-7. Accessing the `LotteryNumbers` servlet results in normal response (with the document sent to the client) in two situations: when there is an unconditional `GET` request or when there is a conditional request that specifies a date before servlet initialization. Code for the `WebClient` program (used here to interactively connect to the server) is available at the book's source code archive at http://www.coreservlets.com/.



```
HTTP/1.1 200 OK
Last-Modified: Fri, 13 Sep 2002 16:32:47 GMT
Content-Type: text/html
Content-Length: 480
Date: Fri, 13 Sep 2002 17:03:45 GMT
Server: Apache Coyote HTTP/1.1 Connector [1.0]
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD><TITLE>Your Lottery Numbers</TITLE></HEAD>
<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN=CENTER>Your Lottery Numbers</H1>
<B>Based upon extensive research of astro-illogical trends, psychic farces, and d
  <LI>77
  <LI>37
  <LI>24
```

Figure 3-8. Accessing the `LotteryNumbers` servlet results in a 304 (Not Modified) response with no actual document in one situation: when a conditional `GET` request is received that specifies a date at or after servlet initialization.



```
localhost
80
GET /servlet/coreservlets.LotteryNumbers HTTP/1.0
```

```
If-Modified-Since: Fri, 13 Sep 2002 16:32:47 GMT
```

```
HTTP/1.1 304 Not Modified
Content-Length: 0
Date: Fri, 13 Sep 2002 17:04:37 GMT
Server: Apache Coyote HTTP/1.1 Connector [1.0]
Connection: close
```

Initializations Controlled by Initialization Parameters

In the previous example, the `init` method computed some data that was used by the `doGet` and `getLastModified` methods. Although this type of general initialization is quite common, it is also common to control the initialization by the use of initialization parameters. To understand the motivation for init parameters, you need to understand the categories of people who might want to customize the way a servlet or JSP page behaves. There are three such groups:

1. Developers.
2. End users.
3. Deployers.

Developers change the behavior of a servlet by changing the code. End users change the behavior of a servlet by providing data to an HTML form (assuming that the developer has written the servlet to look for this data). But what about deployers? There needs to be a way to let administrators move servlets from machine to machine and change certain parameters (e.g., the address of a database, the size of a connection pool, or the location of a data file) without modifying the servlet source code. Providing this capability is the purpose of init parameters.

Because the use of servlet initialization parameters relies heavily on the deployment descriptor (web.xml), we postpone details and examples on init parameters until the deployment descriptor chapter in Volume 2 of this book. But, here is a brief preview:

1. Use the web.xml servlet element to give a name to your servlet.
2. Use the web.xml servlet-mapping element to assign a custom URL to your servlet. You never use default URLs of the form http://.../**servlet**/*ServletName* when using init parameters. In fact, these default URLs, although extremely convenient during initial development, are almost never used in deployment scenarios.
3. Add init-param subelements to the web.xml servlet element to assign names and values of initialization parameters.
4. From within your servlet's init method, call getServletConfig to obtain a reference to the ServletConfig object.
5. Call the getInitParameter method of ServletConfig with the name of the init parameter. The return value is the value of the init parameter or null if no such init parameter is found in the web.xml file.

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 (remember those California power outages?). So, don't count on destroy as the *only* mechanism for saving state to disk. If your servlet performs activities like counting hits or accumulating lists of cookie values that indicate special access, you should also proactively write the data to disk periodically.

[Team LiB] PREVIOUS NEXT