3.7 The SingleThreadModel Interface

Normally, the system makes a single instance of your servlet and then creates a new thread for each user request. This means that if a new request comes in while a previous request is still executing, multiple threads can concurrently be accessing the same servlet object. Consequently, your `doGet` and `doPost` methods must be careful to synchronize access to fields and other shared data (if any) since multiple threads may access the data simultaneously. Note that local variables are not shared by multiple threads, and thus need no special protection.

In principle, you can prevent multithreaded access by having 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. In most cases, it does so by queuing all the requests and passing them one at a time to a single servlet instance. However, the server is permitted to create a pool of multiple instances, each of which handles one request at a time. Either way, 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.

Although `SingleThreadModel` prevents concurrent access in principle, in practice there are two reasons why it is usually a poor choice.

First, synchronous access to your servlets can significantly hurt performance (latency) if your servlet is accessed frequently. When a servlet waits for I/O, the server cannot handle pending requests for the same servlet. So, think twice before using the `SingleThreadModel` approach. Instead, consider synchronizing only the part of the code that manipulates the shared data.

The second problem with `SingleThreadModel` stems from the fact that the specification permits servers to use pools of instances instead of queueing up the requests to a single instance. As long as each instance handles only one request at a time, the pool-of-instances approach satisfies the requirements of the specification. But, it is a bad idea.

Suppose, on one hand, that you are using regular non-static instance variables (fields) to refer to shared data. Sure, `SingleThreadModel` prevents concurrent access, but it does so by throwing out the baby with the bath water: each servlet instance has a separate copy of the instance variables, so the data is no longer shared properly.

On the other hand, suppose that you are using static instance variables to refer to the shared data. In that case, the pool-of-instances approach to `SingleThreadModel` provides no advantage whatsoever; multiple requests (using different instances) can still concurrently access the static data.

Now, `SingleThreadModel` is still occasionally useful. For example, it can be used when the instance variables are reinitialized for each request (e.g., when they are used merely to simplify communication among methods). But, the problems with `SingleThreadModel` are so severe that it is deprecated in the servlet 2.4 (JSP 2.0) specification. You are much better off using explicit `synchronized` blocks.

Core Warning

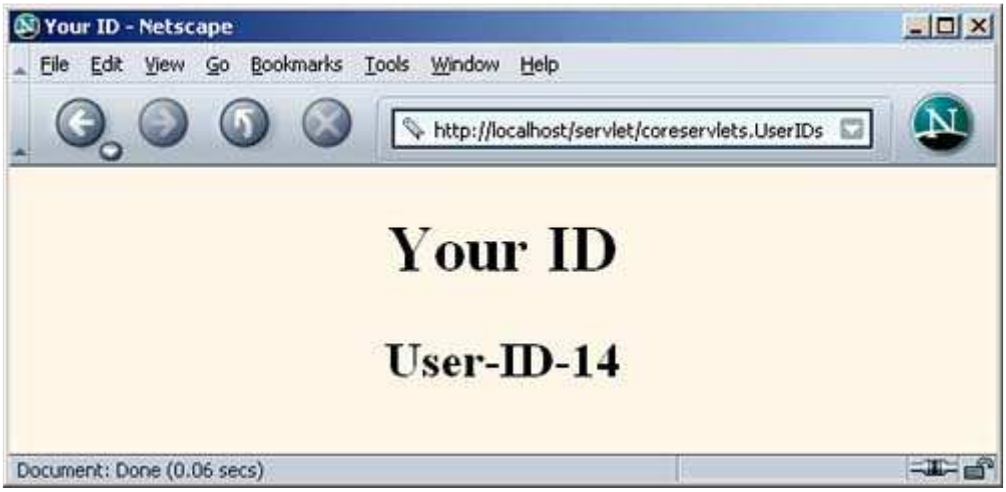
Avoid implementing `SingleThreadModel` for high-traffic servlets. Use it with great caution at other times. For production-level code, explicit code synchronization is almost always better. `SingleThreadModel` is deprecated in version 2.4 of the servlet specification.

For example, consider the servlet of Listing 3.8 that attempts to assign unique user IDs to each client (unique until the server restarts, that is). It uses an instance variable (field) called `nextID` to keep track of which ID should be assigned next, and uses the following code to output the ID.

```
String id = "User-ID-" + nextID;
out.println("<H2>" + id + "</H2>");
nextID = nextID + 1;
```

Now, suppose you were very careful in testing this servlet. You put it in a subdirectory called `coreservlets`, compiled it, and copied the `coreservlets` directory to the `WEB-INF/classes` directory of the default Web application (see Section 2.10, "Deployment Directories for Default Web Application: Summary"). You started the server. You repeatedly accessed the servlet with `http://localhost/servlet/coreservlets.UserIDs`. Every time you accessed it, you got a different value (Figure 3-9). So the code is correct, right? Wrong! The problem occurs only when there are multiple simultaneous accesses to the servlet. Even then, it occurs only once in a while. But, in a few cases, the first client could read the `nextID` field and have its thread preempted before it incremented the field. Then, a second client could read the field and get the same value as the first client. Big trouble! For example, there have been real-world e-commerce applications where customer purchases were occasionally charged to the wrong client's credit card, precisely because of such a race condition in the generation of user IDs.

Figure 3-9. Result of the `UserIDs` servlet.



Now, if you are familiar with multithreaded programming, the problem was very obvious to you. The question is, what is the proper solution? Here are three possibilities.

1. **Shorten the race**. Remove the third line of the code snippet and change the first line to the following.

   ```
   String id = "User-ID-" + nextID++;
   ```

   Boo! This approach decreases the likelihood of an incorrect answer, but does not eliminate the possibility. In many scenarios, lowering the probability of a wrong answer is a bad thing, not a good thing: it merely means that the problem is less likely to be

detected in testing, and more likely to occur after being fielded.

2. **Use `SingleThreadModel`**. Change the servlet class definition to the following.

```
public class UserIDs extends HttpServlet
                     implements SingleThreadModel {
```

Will this work? If the server implements `SingleThreadModel` by queueing up all the requests, then, yes, this will work. But at a performance cost if there is a lot of concurrent access. Even worse, if the server implements `SingleThreadModel` by making a pool of servlet instances, this approach will totally fail because each instance will have its own `nextID` field. Either server implementation approach is legal, so this "solution" is no solution at all.

3. **Synchronize the code explicitly**. Use the standard synchronization construct of the Java programming language. Start a `synchronized` block just before the first access to the shared data, and end the block just after the last update to the data, as follows.

```
synchronized(this) {
  String id = "User-ID-" + nextID;
  out.println("<H2>" + id + "</H2>");
  nextID = nextID + 1;
}
```

This technique tells the system that, once a thread has entered the above block of code (or any other `synchronized` section labelled with the same object reference), no other thread is allowed in until the first thread exits. This is the solution you have always used in the Java programming language. It is the right one here, too. Forget error-prone and low-performance `SingleThreadModel` shortcuts; fix race conditions the right way.

Listing 3.8 coreservlets/UserIDs.java

```
package coreservlets;

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

/** Servlet that attempts to give each user a unique
 *  user ID. However, because it fails to synchronize
 *  access to the nextID field, it suffers from race
 *  conditions: two users could get the same ID.
 */

public class UserIDs extends HttpServlet {
  private int nextID = 0;

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

```
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H1>" + title + "</H1>\n");
    String id = "User-ID-" + nextID;
    out.println("<H2>" + id + "</H2>");
    nextID = nextID + 1;
    out.println("</BODY></HTML>");
  }
}
```