# 7.4 Persistent Servlet State and Auto-Reloading Pages

Suppose your servlet or JSP page performs a calculation that takes a long time to complete: say, 20 seconds or more. In such a case, it is not reasonable to complete the computation and then send the results to the client—by that time the client may have given up and left the page or, worse, have hit the Reload button and restarted the process. To deal with requests that take a long time to process (or whose results periodically change), you need the following capabilities:

- **A way to store data between requests.** For data that is not specific to any one client, store it in a field (instance variable) of the servlet. For data that is specific to a user, store it in the `HttpSession` object (see Chapter 9, "Session Tracking"). For data that needs to be available to other servlets or JSP pages, store it in the `ServletContext` (see the section on sharing data in Chapter 14, "Using JavaBeans Components in JSP Documents").

- **A way to keep computations running after the response is sent to the user.** This task is simple: just start a `Thread`. The thread started by the system to answer requests automatically finishes when the response is finished, but other threads can keep running. The only subtlety: set the thread priority to a low value so that you do not slow down the server.

- **A way to get the updated results to the browser when they are ready.** Unfortunately, because browsers do not maintain an open connection to the server, there is no easy way for the server to proactively send the new results to the browser. Instead, the browser needs to be told to ask for updates. That is the purpose of the `Refresh` response header.

## Finding Prime Numbers for Use with Public Key Cryptography

Here is an example that lets you ask for a list of some large, randomly chosen prime numbers. As you are probably aware, access to large prime numbers is the key to most public-key cryptography systems, the kind of encryption systems used on the Web (e.g., for SSL and X509 certificates). Finding prime numbers may take some time for very large numbers (e.g., 100 digits), so the servlet immediately returns initial results but then keeps calculating, using a low-priority thread so that it won't degrade Web server performance. If the calculations are not complete, the servlet instructs the browser to ask for a new page in a few seconds by sending it a `Refresh` header.

In addition to illustrating the value of HTTP response headers (`Refresh` in this case), this example shows two other valuable servlet capabilities. First, it shows that the same servlet can handle multiple simultaneous connections, each with its own thread. So, while one thread is finishing a calculation for one client, another client can connect and still see partial results.

Second, this example shows how easy it is for servlets to maintain state between requests, something that is cumbersome to implement in most competing technologies (even .NET, which is perhaps the best of the alternatives). Only a single instance of the servlet is created, and each request simply results in a new thread calling the servlet's `service` method (which calls `doGet` or `doPost`). So, shared data simply has to be placed in a regular instance variable (field) of the servlet. Thus, the servlet can access the appropriate ongoing calculation when the browser reloads the page and can keep a list of the *N* most recently requested results, returning them immediately if a new request specifies the same parameters as a recent one. Of course, the normal rules that require authors to synchronize multithreaded access to shared data still apply to servlets. Servlets can also store persistent data in the `ServletContext` object that is

available through the `getServletContext` method. `ServletContext` has `setAttribute` and `getAttribute` methods that let you store arbitrary data associated with specified keys. The difference between storing data in instance variables and storing it in the `ServletContext` is that the `ServletContext` is shared by all servlets and JSP pages in the Web application.

Listing 7.2 shows the main servlet class. First, it receives a request that specifies two parameters: `numPrimes` and `numDigits`. These values are normally collected from the user and sent to the servlet by means of a simple HTML form. Listing 7.3 shows the source code and Figure 7-2 shows the result. Next, these parameters are converted to integers by means of a simple utility that uses `Integer.parseInt` (see Listing 7.6). These values are then matched by the `findPrimeList` method to an `ArrayList` of recent or ongoing calculations to see if a previous computation corresponds to the same two values. If so, that previous value (of type `PrimeList`) is used; otherwise, a new `PrimeList` is created and stored in the ongoing-calculations `Vector`, potentially displacing the oldest previous list. Next, that `PrimeList` is checked to determine whether it has finished finding all of its primes. If not, the client is sent a `Refresh` header to tell it to come back in five seconds for updated results. Either way, a bulleted list of the current values is returned to the client. See Figures 7-3 through 7-5 for representative results.

## Listing 7.2 PrimeNumberServlet.java

```
package coreservlets;

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

/** Servlet that processes a request to generate n
 *  prime numbers, each with at least m digits.
 *  It performs the calculations in a low-priority background
 *  thread, returning only the results it has found so far.
 *  If these results are not complete, it sends a Refresh
 *  header instructing the browser to ask for new results a
 *  little while later. It also maintains a list of a
 *  small number of previously calculated prime lists
 *  to return immediately to anyone who supplies the
 *  same n and m as a recently completed computation.
 */

public class PrimeNumberServlet extends HttpServlet {
  private ArrayList primeListCollection = new ArrayList();
  private int maxPrimeLists = 30;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    int numPrimes =
      ServletUtilities.getIntParameter(request,
                                       "numPrimes", 50);
    int numDigits =
      ServletUtilities.getIntParameter(request,
                                       "numDigits", 120);
    PrimeList primeList =
      findPrimeList(primeListCollection, numPrimes, numDigits);
    if (primeList == null) {
      primeList = new PrimeList(numPrimes, numDigits, true);
      // Multiple servlet request threads share the instance
      // variables (fields) of PrimeNumbers. So
      // synchronize all access to servlet fields.
      synchronized(primeListCollection) {
```

```
        if (primeListCollection.size() >= maxPrimeLists)
          primeListCollection.remove(0);
        primeListCollection.add(primeList);
      }
    }
    ArrayList currentPrimes = primeList.getPrimes();
    int numCurrentPrimes = currentPrimes.size();
    int numPrimesRemaining = (numPrimes - numCurrentPrimes);
    boolean isLastResult = (numPrimesRemaining == 0);
    if (!isLastResult) {
      response.setIntHeader("Refresh", 5);
    }
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String title = "Some " + numDigits + "-Digit Prime Numbers";
    out.println(ServletUtilities.headWithTitle(title) +
                "<BODY BGCOLOR=\"#FDF5E6\">\n" +
                "<H2 ALIGN=CENTER>" + title + "</H2>\n" +
                "<H3>Primes found with " + numDigits +
                " or more digits: " + numCurrentPrimes +
                ".</H3>");
    if (isLastResult)
      out.println("<B>Done searching.</B>");
    else
      out.println("<B>Still looking for " + numPrimesRemaining +
                  " more<BLINK>...</BLINK></B>");
    out.println("<OL>");
    for(int i=0; i<numCurrentPrimes; i++) {
      out.println("  <LI>" + currentPrimes.get(i));
    }
    out.println("</OL>");
    out.println("</BODY></HTML>");
  }

  // See if there is an existing ongoing or completed
  // calculation with the same number of primes and number
  // of digits per prime. If so, return those results instead
  // of starting a new background thread. Keep this list
  // small so that the Web server doesn't use too much memory.
  // Synchronize access to the list since there may be
  // multiple simultaneous requests.

  private PrimeList findPrimeList(ArrayList primeListCollection,
                                  int numPrimes,
                                  int numDigits) {
      for(int i=0; i<primeListCollection.size(); i++) {
        PrimeList primes =
          (PrimeList)primeListCollection.get(i);
    synchronized(primeListCollection) {
        if ((numPrimes == primes.numPrimes()) &&
            (numDigits == primes.numDigits()))
          return(primes);
      }
      return(null);
    }
  }
}
```

## Listing 7.3 PrimeNumbers.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
  <TITLE>Finding Large Prime Numbers</TITLE>
```

```
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<CENTER>
<H2>Finding Large Prime Numbers</H2>
<BR><BR>
<FORM ACTION="/servlet/coreservlets.PrimeNumberServlet">
  <B>Number of primes to calculate:</B>
  <INPUT TYPE="TEXT" NAME="numPrimes" VALUE=25 SIZE=4><BR>
  <B>Number of digits:</B>
  <INPUT TYPE="TEXT" NAME="numDigits" VALUE=150 SIZE=3><BR>
  <INPUT TYPE="SUBMIT" VALUE="Start Calculating">
</FORM>
</CENTER>
</BODY></HTML>
```

**Figure 7-2. Front end to the prime-number-generation servlet.**

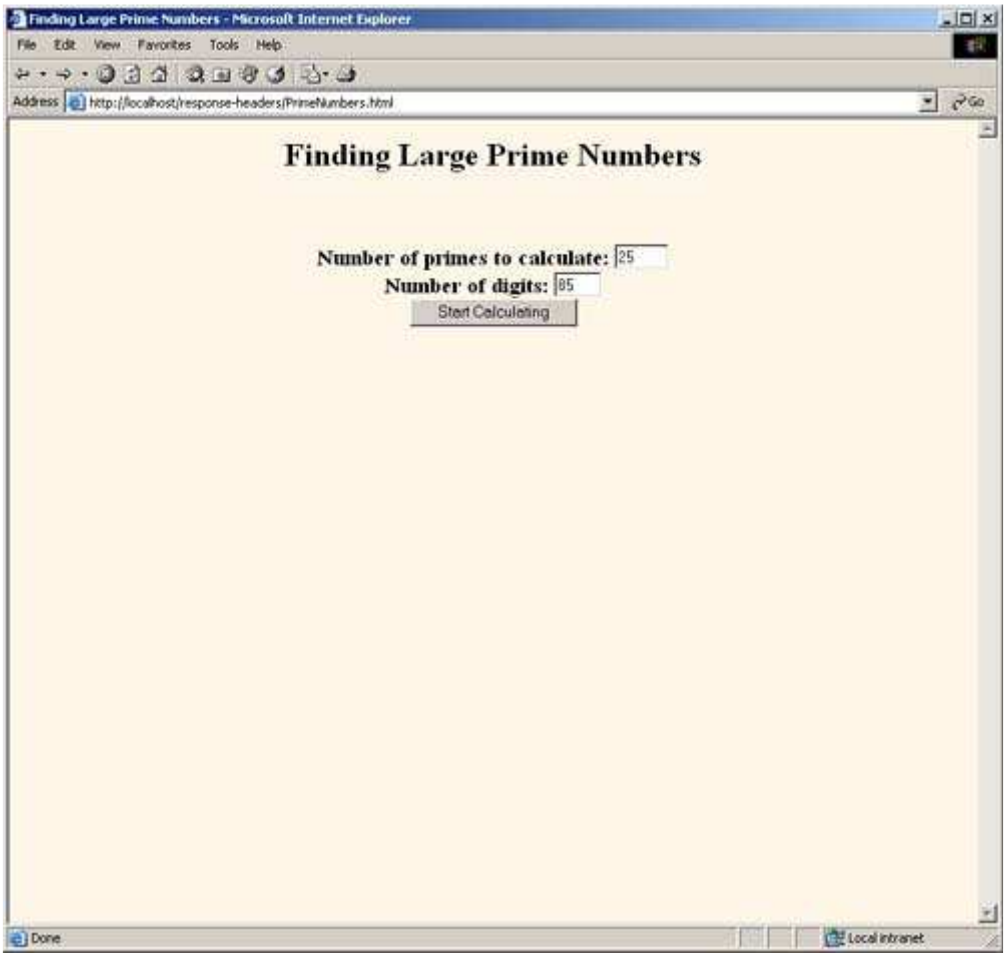



**Figure 7-3. Initial results of the prime-number-generation servlet. A quick result is sent to the browser, along with instructions (in the `Refresh` header) to reconnect for an update in five seconds.**

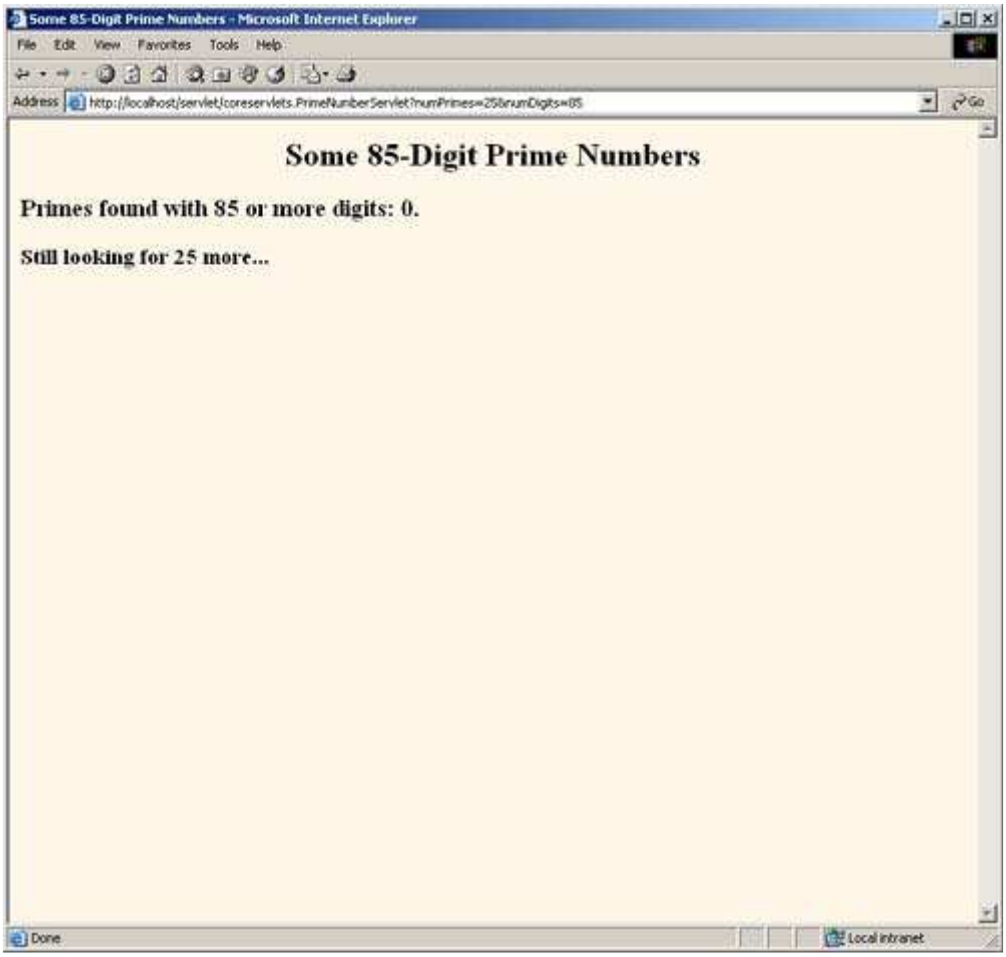



**Figure 7-5. Final results of the prime-number-generation servlet. Since the servlet has computed as many primes as the user requested, no `Refresh` header is sent to the browser and the page is no longer reloaded automatically.**

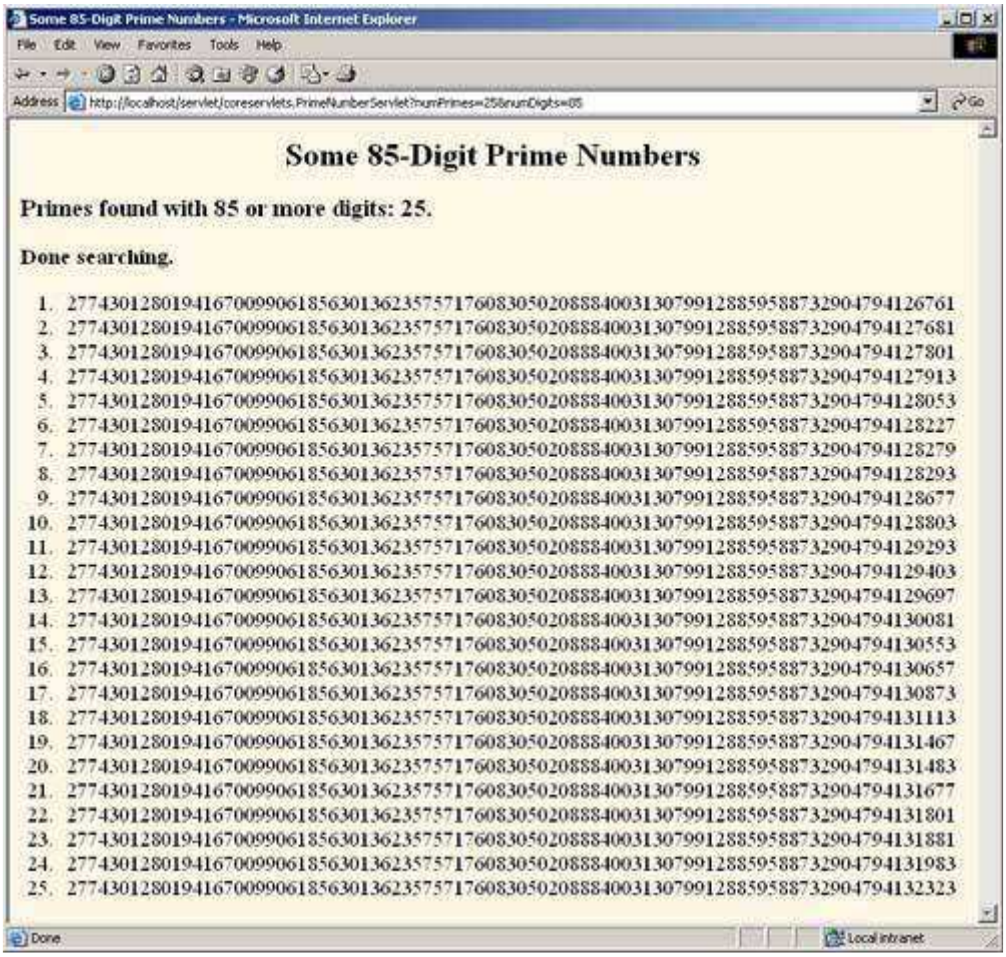
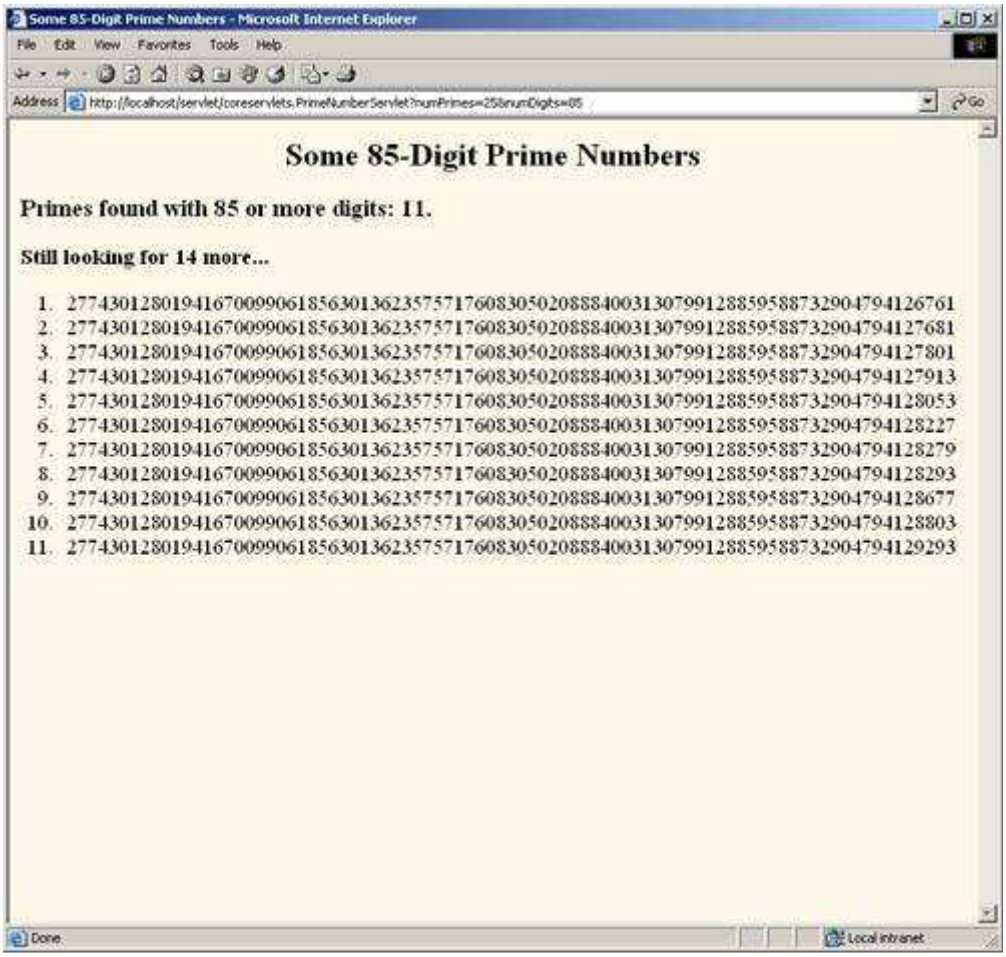



**Figure 7-4. Intermediate results of the prime-number-generation servlet. The servlet stores the previous computations and matches the current request with the stored values by comparing the request parameters (the size and number of primes to compute). Other clients that request the same parameters see the same already computed results.**




Listings 7.4 (`PrimeList.java`) and 7.5 (`Primes.java`) present auxiliary code used by the servlet. `PrimeList.java` handles the background thread for the creation of a list of primes for a specific set of values. The point of this example is twofold: that servlets can maintain data between requests by storing it in instance variables (or the `ServletContext`) and that the servlet can use the `Refresh` header to instruct the browser to return for updates. However, if you care about the gory details of prime-number generation, `Primes.java` contains the low-level algorithms for choosing a random number of a specified length and then finding a prime at or above that value. It uses built-in methods in the `BigInteger` class; the algorithm for determining if the number is prime is a probabilistic one and thus has a chance of being mistaken. However, the probability of being wrong can be specified, and we use an error value of 100. Assuming that the algorithm used in most Java implementations is the Miller-Rabin test, the likelihood of falsely reporting a composite (i.e., non-prime) number as prime is provably less than $2^{100}$. This is almost certainly smaller than the likelihood of a hardware error or random radiation causing an incorrect response in a deterministic algorithm, and thus the algorithm can be considered deterministic.

## Listing 7.4 PrimeList.java

```
package coreservlets;

import java.util.*;
import java.math.BigInteger;

/** Creates an ArrayList of large prime numbers, usually in
 *  a low-priority background thread. Provides a few small
 *  thread-safe access methods.
 */

public class PrimeList implements Runnable {
```

```
  private ArrayList primesFound;
  private int numPrimes, numDigits;

  /** Finds numPrimes prime numbers, each of which is
   *  numDigits long or longer. You can set it to return
   *  only when done, or have it return immediately,
   *  and you can later poll it to see how far it
   *  has gotten.
   */

  public PrimeList(int numPrimes, int numDigits,
                   boolean runInBackground) {
    primesFound = new ArrayList(numPrimes);
    this.numPrimes = numPrimes;
    this.numDigits = numDigits;
    if (runInBackground) {
      Thread t = new Thread(this);
      // Use low priority so you don't slow down server.
      t.setPriority(Thread.MIN_PRIORITY);
      t.start();
    } else {
      run();
    }
  }

  public void run() {
    BigInteger start = Primes.random(numDigits);
    for(int i=0; i<numPrimes; i++) {
      start = Primes.nextPrime(start);
      synchronized(this) {
        primesFound.add(start);
      }
    }
  }

  public synchronized boolean isDone() {
    return(primesFound.size() == numPrimes);
  }

  public synchronized ArrayList getPrimes() {
    if (isDone())
      return(primesFound);
    else
      return((ArrayList)primesFound.clone());
  }

  public int numDigits() {
    return(numDigits);
  }

  public int numPrimes() {
    return(numPrimes);
  }

  public synchronized int numCalculatedPrimes() {
    return(primesFound.size());
  }
}
```

### Listing 7.5 Primes.java

```
package coreservlets;

import java.math.BigInteger;
```

```
/** A few utilities to generate a large random BigInteger,
 *  and find the next prime number above a given BigInteger.
 */

public class Primes {
  // Note that BigInteger.ZERO and BigInteger.ONE are
  // unavailable in JDK 1.1.
  private static final BigInteger ZERO = BigInteger.ZERO;
  private static final BigInteger ONE = BigInteger.ONE;
  private static final BigInteger TWO = new BigInteger("2");

  // Likelihood of false prime is less than 1/2^ERR_VAL.
  // Presumably BigInteger uses the Miller-Rabin test or
  // equivalent, and thus is NOT fooled by Carmichael numbers.
  // See section 33.8 of Cormen et al.'s Introduction to
  // Algorithms for details.
  private static final int ERR_VAL = 100;

  public static BigInteger nextPrime(BigInteger start) {
    if (isEven(start))
      start = start.add(ONE);
    else
      start = start.add(TWO);
    if (start.isProbablePrime(ERR_VAL))
      return(start);
    else
      return(nextPrime(start));
  }

  private static boolean isEven(BigInteger n) {
    return(n.mod(TWO).equals(ZERO));
  }

  private static StringBuffer[] digits =
    { new StringBuffer("0"), new StringBuffer("1"),
      new StringBuffer("2"), new StringBuffer("3"),
      new StringBuffer("4"), new StringBuffer("5"),
      new StringBuffer("6"), new StringBuffer("7"),
      new StringBuffer("8"), new StringBuffer("9") };

  private static StringBuffer randomDigit(boolean isZeroOK) {
    int index;
    if (isZeroOK) {
      index = (int)Math.floor(Math.random() * 10);
    } else {
      index = 1 + (int)Math.floor(Math.random() * 9);
    }
    return(digits[index]);
  }

  /** Create a random big integer where every digit is
   *  selected randomly (except that the first digit
   *  cannot be a zero).
   */

  public static BigInteger random(int numDigits) {
    StringBuffer s = new StringBuffer("");
    for(int i=0; i<numDigits; i++) {
      if (i == 0) {
        // First digit must be non-zero.
        s.append(randomDigit(false));
      } else {
        s.append(randomDigit(true));
```

```
      }
    }
    return(new BigInteger(s.toString()));
  }

  /** Simple command-line program to test. Enter number
   *  of digits, and the program picks a random number of that
   *  length and then prints the first 50 prime numbers
   *  above that.
   */

  public static void main(String[] args) {
    int numDigits;
    try {
      numDigits = Integer.parseInt(args[0]);
    } catch (Exception e) { // No args or illegal arg.
      numDigits = 150;
    }
    BigInteger start = random(numDigits);
    for(int i=0; i<50; i++) {
      start = nextPrime(start);
      System.out.println("Prime " + i + " = " + start);
    }
  }
}
```

### Listing 7.6 ServletUtilities.java (Excerpt)

```
package coreservlets;

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

/** Some simple time savers. Note that most are static methods. */

public class ServletUtilities {
  // ...

  /** Read a parameter with the specified name, convert it
   *  to an int, and return it. Return the designated default
   *  value if the parameter doesn't exist or if it is an
   *  illegal integer format.
   */

  public static int getIntParameter(HttpServletRequest request,
                                    String paramName,
                                    int defaultValue) {
    String paramString = request.getParameter(paramName);
    int paramValue;
    try {
      paramValue = Integer.parseInt(paramString);
    } catch(NumberFormatException nfe) { // null or bad format
      paramValue = defaultValue;
    }
    return(paramValue);
  }
}
```