Chapter 17. Network Programming

Topics in This Chapter

- Implementing a generic network client
- Processing strings with StringTokenizer
- Validating e-mail addresses with a network client
- Retrieving files from an HTTP server
- Retrieving Web documents by using the URL class
- Implementing a generic network server
- Creating a simple HTTP server
- Converting a single-threaded server to a multithreaded server
- Invoking distributed objects with RMI

Network programming involves two distinct pieces: a *client* and a *server.* A *client* is the program that connects to a system to request services. A *server* is a program that runs on a machine listening on a designated part of the network (a "port"), waiting for other programs to connect. When the client requests the connection, the server fulfills the request and provides some service to the connecting program. Servers often provide services to more than one connecting program, either one after the other or to several concurrently. If the distinction between who is requesting the service and who is providing the server seems blurry, just remember that the server starts first and doesn't need to know who will be connecting, whereas the client starts second and has to specify a particular host to talk to.

Core Note

Remember that the security manager of most browsers prohibits applets from making network connections to machines other than the one from which they are loaded.

In this chapter, you'll see how to directly implement clients and servers by using network "sockets." Although this is the lowest-level type of network programming in the Java platform, if you've used sockets in other languages, you may be surprised at how simple they are to use in Java technology. The `URL` class helps hide the details of network programming by providing methods for opening connections and binding input/output streams with sockets. In addition to sockets, which can communicate with general-purpose programs in arbitrary languages, Java provides two higher-level packages for communicating with specific types of systems: Remote Method Invocation (RMI) and database connectivity (JDBC). The RMI package lets you easily access methods in remote Java objects and transfer serializable objects across network connections. RMI is covered in Section 17.9. JDBC lets you easily send SQL statements to remote databases. Java Database Connectivity is covered in Chapter 22.

17.1 Implementing a Client

The client is the program that initiates a network connection. Implementing a client consists of five basic steps:

1. Create a `Socket` object.
2. Create an output stream that can be used to send information to the `Socket`.
3. Create an input stream to read the response from the server.
4. Do I/O with input and output streams.
5. Close the `Socket` when done.

Each of these steps is described in the sections that follow. Note that most of the methods described in these sections throw an `IOException` and need to be wrapped in a `try`/`catch` block.

Create a `Socket` object. A `Socket` is the Java object corresponding to a network connection. A client connects to an existing server that is listening on a numbered network port for a connection. The standard way of making a socket is to supply a hostname or IP address and port as follows:

```
Socket client = new Socket("hostname", portNumber);
```

or

```
Socket client = new Socket("IP address", portNumber);
```

If you are already familiar with network programming, note that this approach creates a connection-oriented socket. The Java programming language also supports connectionless (UDP) sockets through the `DatagramSocket` class.

Create an output stream that can be used to send information to the `Socket`. The Java programming language does not have separate methods to send data to files, sockets, and standard output. Instead, Java starts with different underlying objects, then layers standard output streams on top of them. So, any variety of `OutputStream` available for files is also available for sockets. A common one is `PrintWriter`. This stream lets you use `print` and `println` on the socket in exactly the same way as you would print to the screen. The `PrintWriter` constructor takes a generic `OutputStream` as an argument, which you can obtain from the `Socket` by means of `getOutputStream`. In addition, you should specify `true` in the constructor to force autoflush. Normally, the contents written to the stream will remain in a buffer until the buffer becomes completely full. Once the buffer is full, the contents are flushed out the stream. Autoflush guarantees that the buffer is flushed after every `println`, instead of waiting for the buffer to fill. Here's an example:

```
PrintWriter out =
  new PrintWriter(client.getOutputStream(), true);
```

You can also use an `ObjectOutputStream` to send complex Java objects over the network to be reassembled at the other end. An `ObjectOutputStream` connected to the network is used in exactly the same way as one connected to a file; simply use `writeObject` to send a serializable object and all referenced serializable objects. The server on the other end would use an `ObjectInputStream`'s `readObject` method to reassemble the sent object. Note that all AWT components are automatically serializable, and making other objects serializable is a simple matter of declaring that they implement the `Serializable` interface. See Section 13.9 (Serializing Windows) for more details and an example. Also see Section 17.9 (RMI: Remote Method Invocation) for a high-level interface that uses serialization to let you distribute Java objects across networks.

Create an input stream to read the response from the server. Once you send data to the server, you will want to read the server's response. Again, there is no socket-specific way of doing this; you use a standard input stream layered on top of the socket. The most common one is an `InputStreamReader`, for handling character-based data. Here is a sample:

```
InputStreamReader in =
  new InputStreamReader(client.getInputStream());
```

Although this approach is the simplest, in most cases a better approach is to wrap the socket's generic `InputStream` inside a `BufferedReader`. This approach causes the system to read the data in blocks behind the scenes, rather than reading the underlying stream every time the user performs a read. This approach usually results in significantly improved performance at the cost of a small increase in memory usage (the buffer size, which defaults to 512 bytes). Here's the idea:

```
BufferedReader in =
  new BufferedReader
   (new InputStreamReader(client.getInputStream()));
```

Core Performance Tip

If you are going to read from a socket multiple times, a buffered input stream can speed things up considerably.

In a few cases, you might want to send data to a server but not read anything back. You could imagine a simple e-mail client working this way. In that case, you can skip this step. In other cases, you might want to read data without sending anything first. For instance, you might connect to a network "clock" to read the time. In such a case, you would skip the output stream and just follow this step. In most cases, however, you will want to both send and receive data, so you will follow both steps. Also, if the server is sending complex objects and is written in the Java programming language, you will want to open an `ObjectInputStream` and use `readObject` to receive data.

Do I/O with input and output streams. A `PrintStream` has `print` and `println` methods that let you send a single primitive value, a `String`, or a string representation of an `Object` over the network. If you send an `Object`, the object is converted to a `String` by calling the `toString` method of the class. Most likely you are already familiar with these methods, since `System.out` is in fact an instance of `PrintStream`. `PrintStream` also inherits some simple `write` methods from `OutputStream`. These methods let you send binary data by sending an individual byte or an array of bytes.

`PrintWriter` is similar to `PrintStream` and has the same `print` and `println` methods. The main difference is that you can create print writers for different Unicode character sets, and you can't do that with `PrintStream`.

`BufferedReader` has two particularly useful methods: `read` and `readLine`. The `read` method returns a single `char` (as an `int`); `readLine` reads a whole line and returns a `String`. Both of these methods are *blocking;* they do not return until data is available. Because `readLine` will wait until receiving a carriage return or an `EOF` (the server closed the connection), `readLine` should be used only when you are sure the server will close the socket when done transmitting or when you know the number of lines that will be sent by the server. The `readLine` method returns `null` upon receiving an `EOF`.

Close the `Socket` when done. When you are done, close the socket with the `close` method:

```
client.close();
```

This method closes the associated input and output streams as well.

Example: A Generic Network Client

Listing 17.1 illustrates the approach outlined in the preceding section. Processing starts with the `connect` method, which initiates the connection, then passes the socket to `handleConnection` to do the actual communication. This version of `handleConnection` simply reports who made the connection, sends a single line to the server ("`Generic Network Client`"), reads and prints a single response line, and exits. Real clients would override `handleConnection` to implement their desired behavior but could leave `connect` unchanged.

Listing 17.1 `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.
 */

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

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

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

  /** Establishes the connection, then passes the socket
   *  to 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);
  }
}
```

The `SocketUtil` class is just a simple interface to the `BufferedReader` and `PrintWriter` constructors and is given in Listing 17.2.

Listing 17.2 `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 {
```

```
    // Second argument of true means autoflush.
    return(new PrintWriter(s.getOutputStream(), true));
  }
}
```

Finally, the `NetworkClientTest` class, shown in Listing 17.3, provides a way to use the `NetworkClient` class with any hostname and any port.

Listing 17.3 `NetworkClientTest.java`

```
/** Make simple connection to host and port specified. */

public class NetworkClientTest {
  public static void main(String[] args) {
    String host = "localhost";
    int port = 8088;
    if (args.length > 0) {
      host = args[0];
    }
    if (args.length > 1) {
      port = Integer.parseInt(args[1]);
    }
    NetworkClient nwClient = new NetworkClient(host, port);
    nwClient.connect();
  }
}
```

Output: Connecting to an FTP Server

Let's use the test program in Listing 17.3 to connect to Netscape's public FTP server, which listens on port 21. Assume > is the DOS or Unix prompt.

```
> java NetworkClientTest ftp.netscape.com 21
Generic Network Client:
Made connection to ftp.netscape.com and got '220 ftp26 FTP server
(UNIX(r) System V Release 4.0) ready.' in response
```

17.2 Parsing Strings by Using StringTokenizer

A common task when doing network programming is to break a large string down into various constituents. A developer could accomplish this task by using low-level `String` methods such as `indexOf` and `substring` to return substrings bounded by certain delimiters. However, the Java platform has a built-in class to simplify this process: the `StringTokenizer` class. This class isn't specific to network programming (the class is located in `java.util`, not `java.net`), but because string processing tends to be a large part of client-server programming, we discuss it here.

The StringTokenizer Class

The idea is that you build a tokenizer from an initial string, then retrieve tokens one at a time with `nextToken`, either based on a set of delimiters defined when the tokenizer was created or as an optional argument to `nextToken`. You can also see how many tokens are remaining (`countTokens`) or simply test whether the number of tokens remaining is nonzero (`hasMoreTokens`). The most common methods are summarized below.

Constructors

public StringTokenizer(String input)

This constructor builds a tokenizer from the input string, using white space (space, tab, newline, return) as the set of delimiters. The delimiters will not be included as part of the tokens returned.

public StringTokenizer(String input, String delimiters)

This constructor creates a tokenizer from the input string, using the specified delimiters. The delimiters will not be included as part of the tokens returned.

public StringTokenizer(String input, String delimiters, boolean includeDelimiters)

This constructor builds a tokenizer from the input string using the specified delimiters. The delimiters *will* be included as part of the tokens returned if the third argument is `true`.

Methods

public String nextToken()

This method returns the next token. The method throws a `NoSuchElementException` if no characters remain or only delimiter characters remain.

public String nextToken(String delimiters)

This method changes the set of delimiters, then returns the next token. The `nextToken` method throws a `NoSuchElementException` if no characters remain or only delimiter characters remain.

public int countTokens()

This method returns the number of tokens remaining, based on the current set of delimiters.

public boolean hasMoreTokens()

This method determines whether any tokens remain, based on the current set of delimiters. Most applications should either check for tokens before calling `nextToken` or catch a `NoSuchElementException` when calling `nextToken`. Note that `hasMoreTokens` has the side effect of advancing the internal counter, which yields unexpected results when doing the rare but possible sequence of checking `hasMoreTokens` with one delimiter set, then calling `nextToken` with another delimiter set.

Example: Interactive Tokenizer

A good way to get a feel for how `StringTokenizer` works is to try a bunch of test cases. Listing 17.4 gives a simple class that lets you enter an input string and a set of delimiters on the command line and prints the resultant tokens one to a line.

Listing 17.4 `TokTest.java`

```
import java.util.StringTokenizer;

/** Prints the tokens resulting from treating the first
 *  command-line argument as the string to be tokenized
 *  and the second as the delimiter set.
 */

public class TokTest {
  public static void main(String[] args) {
    if (args.length == 2) {
      String input = args[0], delimiters = args[1];
      StringTokenizer tok =
        new StringTokenizer(input, delimiters);
      while (tok.hasMoreTokens()) {
        System.out.println(tok.nextToken());
      }
    } else {
      System.out.println
        ("Usage: java TokTest string delimeters");
    }
  }
}
```

Here is `TokTest` in action:

```
> java TokTest http://www.microsoft.com/~gates/ :/.
http
www
microsoft
com
~gates
> java TokTest "if (tok.hasMoreTokens()) {" "(){. "
if
tok
hasMoreTokens
```

17.3 Example: A Client to Verify E-Mail Addresses

One of the best ways to get comfortable with a network protocol is to open a telnet connection to the port a server is on and try out commands interactively. For example, consider connecting directly to a mail server to verify the correctness of potential e-mail addresses. To connect to the mail server, we need to know that SMTP (Simple Mail Transfer Protocol) servers generally listen on port 25, send one or more lines of data after receiving a connection, accept commands of the form `expn` *`username`* to expand usernames, and accept `quit` as the command to terminate the connection. Commands are case insensitive. For instance, Listing 17.5 shows an interaction with the mail server at `apl.jhu.edu`.

Listing 17.5 Talking to a Mail Server

```
> telnet apl.jhu.edu 25
Trying 128.220.101.100...
Connected to aplcenMP.apl.jhu.edu.
Escape character is '^]'.
220 aplcenMP.apl.jhu.edu ESMTP Sendmail 8.9.3/8.9.1; Sat, 10 Feb 2001
12:05:42
500 (EST)
expn hall
250 Marty Hall <hall@aplcenMP.apl.jhu.edu>
expn root
250-Tom Vellani <vellani@aplcenMP.apl.jhu.edu>
250 Gary Gafke <gary@aplcenMP.apl.jhu.edu>
quit
221 aplcenMP.apl.jhu.edu closing connection
Connection closed by foreign host.
```

For a network client to verify an e-mail address, the program needs to break up the address into username and hostname sections, connect to port 25 of the host, read the initial connection message, send an `expn` on the username, read and print the result, then send a `quit`. The response from the SMTP server may not consist of only a single line. Because `readLine` blocks if the connection is still open but no line of data has been sent, we cannot call `readLine` more than once. So, rather than using `readLine` at all, we use `read` to populate a byte array *large* enough to hold the likely response, record how many bytes were read, then use `write` to print out that number of bytes. This process is shown in Listing 17.6, with Listing 17.7 showing the helper class that breaks an e-mail address into separate username and hostname components.

Listing 17.6 `AddressVerifier.java`

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

/** Given an e--mail address of the form user@host,
 *  connect to port 25 of the host and issue an
 *  'expn' request for the user. Print the results.
 */

public class AddressVerifier extends NetworkClient {
  private String username;

  public static void main(String[] args) {
    if (args.length != 1) {
      usage();
   }
    MailAddress address = new MailAddress(args[0]);
    AddressVerifier verifier
      = new AddressVerifier(address.getUsername(),
                            address.getHostname(), 25);
    verifier.connect();
  }

  public AddressVerifier(String username, String hostname,
                         int port) {
    super(hostname, port);
    this.username = username;
  }

  /** NetworkClient, the parent class, automatically establishes
   *  the connection and then passes the Socket to
   *  handleConnection. This method does all the real work
   *  of talking to the mail server.
   */

  // You can't use readLine, because it blocks. Blocking I/O
  // by readLine is only appropriate when you know how many
  // lines to read. Note that mail servers send a varying
  // number of lines when you first connect or send no line
  // closing the connection (as HTTP servers do), yielding
  // null for readLine. Also, we'll assume that 1000 bytes
  // is more than enough to handle any server welcome
  // message and the actual EXPN response.
  protected void handleConnection(Socket client) {
    try {
      PrintWriter out = SocketUtil.getWriter(client);
      InputStream in = client.getInputStream();
      byte[] response = new byte[1000];
      // Clear out mail server's welcome message.
      in.read(response);
      out.println("EXPN " + username);
      // Read the response to the EXPN command.
      int numBytes = in.read(response);
      // The 0 means to use normal ASCII encoding.
      System.out.write(response, 0, numBytes);
      out.println("QUIT");
      client.close();
    } catch(IOException ioe) {
      System.out.println("Couldn't make connection: " + ioe);
    }
  }

  /** If the wrong arguments, thn warn user. */

  public static void usage() {
    System.out.println ("You must supply an email address " +
       "of the form 'username@hostname'.");
    System.exit(--1);
  }
}
```

Listing 17.7 `MailAddress.java`

```
import java.util.*;

/** Takes a string of the form "user@host" and
 *  separates it into the "user" and "host" parts.
 */

public class MailAddress {
  private String username, hostname;

  public MailAddress(String emailAddress) {
    StringTokenizer tokenizer
      = new StringTokenizer(emailAddress, "@");
    this.username = getArg(tokenizer);
    this.hostname = getArg(tokenizer);
  }

  private static String getArg(StringTokenizer tok) {
    try { return(tok.nextToken()); }
    catch (NoSuchElementException nsee) {
      System.out.println("Illegal email address");
      System.exit(--1);
      return(null);
    }
  }

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

  public String getHostname() {
    return(hostname);
  }
}
```

Finally, here is an example of the address verifier in action, looking for the addresses of the main originators of the WWW and the Java platform.

```
> java AddressVerifier tbl@w3.org
250 <timbl@hq.lcs.mit.edu>
> java AddressVerifier timbl@hq.lcs.mit.edu
250 Tim Berners-Lee <timbl>
> java AddressVerifier gosling@mail.javasoft.com
550 gosling... User unknown
```

17.4 Example: A Network Client That Retrieves URLs

Retrieving a document through HTTP is remarkably simple. You open a connection to the HTTP port of the machine hosting the page, send the string `GET` followed by the address of the document, followed by the string `HTTP/1.0`, followed by a blank line (at least one blank space is required between `GET` and address, and between address and `HTTP`). You then read the result one line at a time. Reading a line at a time was not safe with the mail client of Section 17.3 because the server sent an indeterminate number of lines but kept the connection open. Here, however, a `readLine` is safe because the server closes the connection when done, yielding `null` as the return value of `readLine`.

Although quite simple, even this approach is slightly harder than necessary, because the Java programming language has built-in classes (`URL` and `URLConnection`) that simplify the process even further. These classes are demonstrated in Section 17.5, but connecting to a HTTP server "by hand" is a useful exercise to prepare yourself for dealing with protocols that don't have built-in helping methods as well as to gain familiarity with the HTTP protocol. Listing 17.8 shows a telnet connection to the www.corewebprogramming.com HTTP server running on port 80.

Listing 17.8 Retrieving an HTML document directly through telnet

```
Unix> telnet www.corewebprogramming.com 80
Trying 216.248.197.112...
Connected to www.corewebprogramming.com.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Sat, 10 Feb 2001 18:04:17 GMT
Server: Apache/1.3.3 (Unix) PHP/3.0.11 FrontPage/4.0.4.3
Connection: close
Content--Type: text/html

<!DOCTYPE HTML PUBLIC "--//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
...
</HTML>
Connection closed by foreign host.
```

In this telnet session, the document was retrieved through a `GET` request. In other cases, you may only want to receive the HTTP headers associated with the document. For instance, a link validator is an important class of network program that verifies that the links in a specified Web page point to "live" documents. Writing such a program in the Java programming language is relatively straightforward, but to limit load on your servers, you probably want the program to use `HEAD` instead of `GET` (see Section 19.7, "The Client Request: HTTP Request Headers"). Java has no helping class for simply sending a `HEAD` request, but only a trivial change in the following code is needed to perform this request.

A Class to Retrieve a Given URI from a Given Host

Listing 17.9 presents a class that retrieves a file given the host, port, and URI (the filename part of the URL) as separate arguments. The application uses the `NetworkClient` shown earlier in Listing 17.1 to send a single `GET` line to the specified host and port, then reads the result a line at a time, printing each line to the standard output.

Listing 17.9 `UriRetriever.java`

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

/** Retrieve a URL given the host, port, and file as three
 *  separate command--line arguments. A later class
 *  (UrlRetriever) supports a single URL instead.
 */

public class UriRetriever extends NetworkClient {
  private String uri;

  public static void main(String[] args) {
    UriRetriever uriClient
      = new UriRetriever(args[0], Integer.parseInt(args[1]),
                         args[2]);
    uriClient.connect();
  }

  public UriRetriever(String host, int port, String uri) {
    super(host, port);
    this.uri = uri;
  }

  /** Send one GET line, then read the results one line at a
   *  time, printing each to standard output.
   */

  // It is safe to use blocking IO (readLine), since
  // HTTP servers close connection when done, resulting
  // in a null value for readLine.

  protected void handleConnection(Socket uriSocket)
      throws IOException {
    PrintWriter out = SocketUtil.getWriter(uriSocket);
    BufferedReader in = SocketUtil.getReader(uriSocket);
    out.println("GET " + uri + " HTTP/1.0\n");
    String line;
    while ((line = in.readLine()) != null) {
      System.out.println("> " + line);
    }
  }
}
```

A Class to Retrieve a Given URL

The previous program requires the user to pass the hostname, port, and URI as three separate command-line arguments. Listing 17.10 improves on this program by building a front end that parses a whole URL, using `StringTokenizer` (Section 17.2), then passes the appropriate pieces to the `UriRetriever`.

Listing 17.10 `UrlRetriever.java`

```
import java.util.*;

/** This parses the input to get a host, port, and file, then
 *  passes these three values to the UriRetriever class to
 *  grab the URL from the Web.
 */

public class UrlRetriever {
  public static void main(String[] args) {
    checkUsage(args);
    StringTokenizer tok = new StringTokenizer(args[0]);
    String protocol = tok.nextToken(":");
    checkProtocol(protocol);
    String host = tok.nextToken(":/");
    String uri;
    int port = 80;
    try {
      uri = tok.nextToken("");
      if (uri.charAt(0) == ':') {
        tok = new StringTokenizer(uri);
        port = Integer.parseInt(tok.nextToken(":/"));
        uri = tok.nextToken("");
      }
    } catch(NoSuchElementException nsee) {
      uri = "/";
    }
    UriRetriever uriClient = new UriRetriever(host, port, uri);
    uriClient.connect();
  }

  /** Warn user if the URL was forgotten. */
  private static void checkUsage(String[] args) {
    if (args.length != 1) {
      System.out.println("Usage: UrlRetriever <URL>");
      System.exit(--1);
    }
  }

  /** Tell user that this can only handle HTTP. */

  private static void checkProtocol(String protocol) {
    if (!protocol.equals("http")) {
      System.out.println("Don't understand protocol " + protocol);
      System.exit(--1);
    }
  }
}
```

UrlRetriever Output

No explicit port number:

```
Prompt> java UrlRetriever
http://www.microsoft.com/netscape-beats-ie.html
> HTTP/1.1 404 Object Not Found
> Server: Microsoft-IIS/5.0
> Date: Fri, 31 Mar 2000 18:22:11 GMT
> Content-Length: 3243
> Content-Type: text/html
>
> <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
> <html dir=ltr>Explicit port number:
```

Explicit port number:

```
Prompt> java UrlRetriever
http://home.netscape.com:80/ie-beats-netscape.html
> HTTP/1.1 404 Not found
> Server: Netscape-Enterprise/3.6
> Date: Fri, 04 Feb 2000 21:52:29 GMT
> Content-type: text/html
> Connection: close
>
> <TITLE>Not Found</TITLE><H1>Not Found</H1> The requested
object does not exist on this server. The link you followed is
either outdated, inaccurate, or the server has been
instructed not to let you have it.
```

Hey! We just wrote a browser. OK, not quite, seeing as there is still the small matter of formatting the result. Still, not bad for about four pages of code. But we can do even better. In the next section, we'll reduce the code to two pages through the use of the built-in `URL` class. In Section 17.6 (WebClient: Talking to Web Servers Interactively) we'll add a simple user interface that lets you do HTTP requests interactively and view the raw HTML results. Also note that in Section 14.12 (The JEditorPane Component) we showed you how to use a `JEditorPane` to create a real browser that formats the HTML and lets the user follow the hypertext links.

THE COMPANY PAYS ME TEN DOLLARS FOR EVERY BUG I FIX IN MY CODE, RATBERT.
I WANT YOU TO DO YOUR LITTLE RAT DANCE ON MY KEYBOARD SO I'LL HAVE LOTS OF BUGS TO FIX.
HOW AM I DOING?
NOT SO GOOD. YOU JUST AUTHORED A WEB BROWSER.
S. Adams E-mail: SCOTTADAMS@AOL.COM
© 1996 United Feature Syndicate, Inc.(NYC)

DILBERT© UFS. Reprinted with permission.

17.5 The URL Class

The `URL` class provides simple access to URLs. The class automatically parses a string for you, letting you retrieve the protocol (e.g., http), host (e.g., `java.sun.com`), port (e.g., 80), and filename (e.g., `/reports/earnings.html`) separately. The `URL` class also provides an easy-to-use interface for reading remote files.

Reading from a URL

Although writing a client to explicitly connect to an HTTP server and retrieve a URL was quite simple, this task is so common that the Java programming language provides a helper class: `java.net.URL`. We saw this class when we looked at applets (see Section 9.5, "Other Applet Methods"): a `URL` object of this type that needed to be passed to `getAppletContext().showDocument`. However, the `URL` class can also be used to parse a string representing a URL and read the contents. An example of parsing a URL is shown in Listing 17.11.

Listing 17.11 `UrlRetriever2.java`

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

/** Read a remote file using the standard URL class
 *  instead of connecting explicitly to the HTTP server.
 */

public class UrlRetriever2 {
  public static void main(String[] args) {
    checkUsage(args);
    try {
      URL url = new URL(args[0]);
      BufferedReader in = new BufferedReader(
        new InputStreamReader(url.openStream()));
      String line;
      while ((line = in.readLine()) != null) {
        System.out.println("> " + line);
     }
      in.close();
    } catch(MalformedURLException mue) { // URL constructor
        System.out.println(args[0] + "is an invalid URL: " + mue);
    } catch(IOException ioe) { // Stream constructors
      System.out.println("IOException: " + ioe);
    }
  }

  private static void checkUsage(String[] args) {
    if (args.length != 1) {
      System.out.println("Usage: UrlRetriever2 <URL>");
      System.exit(--1);
    }
  }
}
```

Here is the `UrlRetriever2` in action:

```
Prompt> java UrlRetriever2 http://www.whitehouse.gov/
> <HTML>
> <HEAD>
> <TITLE>Welcome To The White House</TITLE>
> </HEAD>
> ... Remainder of HTML document omitted ...
> </HTML>
```

This implementation just prints out the resultant document, not the HTTP response lines included in the original "raw" `UrlRetriever` class. However, another Java class called `URLConnection` will supply this information. Create a `URLConnection` object by calling the `openConnection` method of an existing `URL`, then use methods such as `getContentType` and `getLastModified` to retrieve the response header information. See the on-line API for `java.net.URLConnection` for more details.

Other Useful Methods of the URL Class

The most valuable use of a `URL` object is to use the constructor to parse a string representation and then to use `openStream` to provide an `InputStream` for reading. However, the class is useful in a number of other ways, as outlined in the following sections.

public URL(String absoluteSpec)

public URL(URL base, String relativeSpec)

public URL(String protocol, String host, String file)

public URL(String protocol, String host, int port, String file)

These four constructors build a URL in different ways. All throw a `MalformedURLException`.

public String getFile()

This method returns the filename (URI) part of the URL. See the output following Listing 17.12.

public String getHost()

This method returns the hostname part of the URL. See the output following Listing 17.12.

public int getPort()

This method returns the port if one was explicitly specified. If not, it returns –1 (*not* 80). See the output following Listing 17.12.

public String getProtocol()

This method returns the protocol part of the URL (i.e., `http`). See the output following Listing 17.12.

public String getRef()

The `getRef` method returns the "reference" (i.e., section heading) part of the URL. See the output following Listing 17.12.

public final InputStream openStream()

This method returns the input stream that can be used for reading, as used in the `UrlRetriever2` class. The method can also throw an `IOException`.

public URLConnection openConnection()

This method yields a `URLConnection` that can be used to retrieve header lines and (for `POST` requests) to supply data to the HTTP server. The `POST` method is discussed in Chapter 19 (Server-Side Java: Servlets).

public String toExternalForm()

This method gives the string representation of the URL, useful for printouts. This method is identical to `toString`.

Listing 17.12 gives an example of some of these methods.

Listing 17.12 `UrlTest.java`

```
import java.net.*;

/** Read a URL from the command line, then print
 *  the various components.
 */

public class UrlTest {
  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        URL url = new URL(args[0]);
        System.out.println
          ("URL: " + url.toExternalForm() + "\n" +
           "  File:      " + url.getFile() + "\n" +
           "  Host:      " + url.getHost() + "\n" +
           "  Port:      " + url.getPort() + "\n" +
           "  Protocol:  " + url.getProtocol() + "\n" +
           "  Reference: " + url.getRef());
      } catch(MalformedURLException mue) {
        System.out.println("Bad URL.");
      }
    } else
      System.out.println("Usage: UrlTest <URL>");
  }
}
```

Here's `UrlTest` in action:

```
> java UrlTest http://www.irs.gov/mission/#squeeze-them-dry
URL: http://www.irs.gov/mission/#squeeze-them-dry
  File:      /mission/
  Host:      www.irs.gov
  Port:      -1
  Protocol:  http
  Reference: squeeze-them-dry
```

17.6 WebClient: Talking to Web Servers Interactively

In Section 17.4 (Example: A Network Client That Retrieves URLs) and Section 17.5 (The URL Class) we illustrated how easy it is to connect to a Web server and request a document. The programs presented accept a URL from the command line and display the retrieved page to the output stream. In this section, we present `WebClient`, a simple graphical interface to HTTP servers. This program, Listing 17.13, accepts an HTTP request line and request headers from the user. When the user presses Submit Request, the request and headers are sent to the server, followed by a blank line. The response is displayed in a scrolling text area. Downloading is performed in a separate thread so that the user can interrupt the download of long documents.

The `HttpClient` class, Listing 17.14, does the real network communication. It simply sends the designated request line and request headers to the Web server, then reads, one at a time, the lines that come back, placing them into a `JTextArea` until either the server closes or the `HttpClient` is interrupted by means of the `isInterrupted` flag.

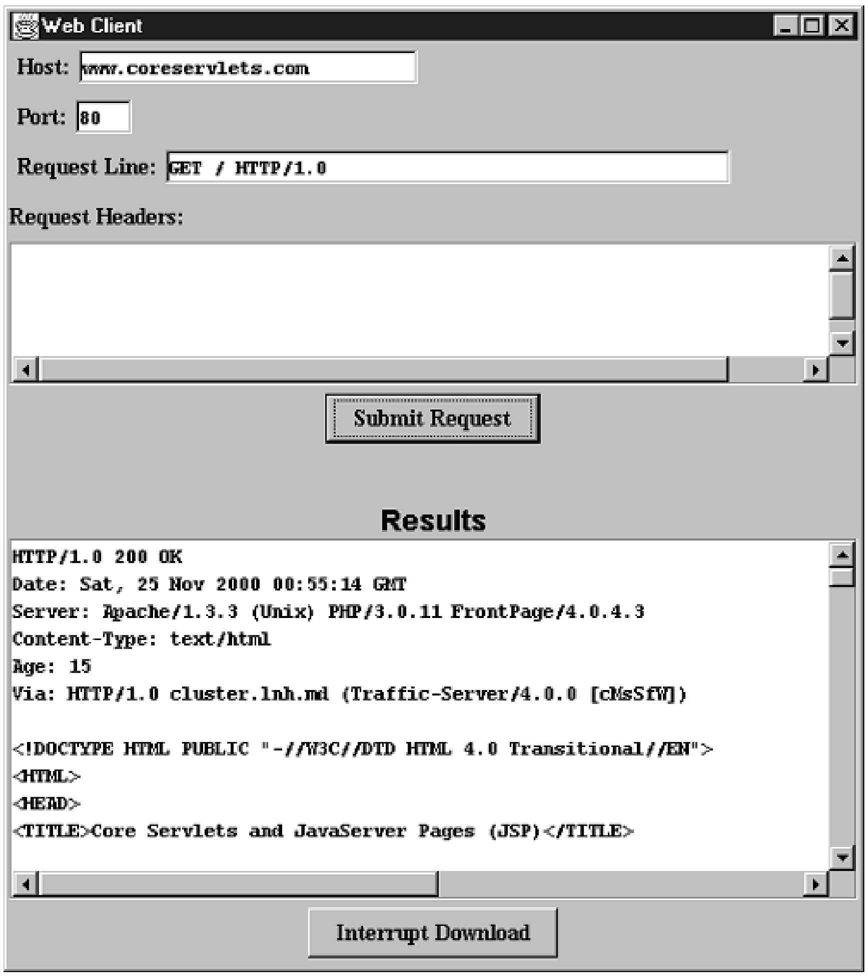
The `LabeledTextField` class, Listing 17.15, is a simple combination of a `JTextField` and a `JLabel` and is used in the upper panel of `WebClient` for user input. The `Interruptible` class, Listing 17.16, is a simple interface used to identify classes that have an `isInterrupted` method.

`Interruptible` is used by `HttpClient` to poll `WebClient` to see if the user has interrupted the program.

As always, the source code can be downloaded from the on-line archive at http://www.corewebprogramming.com/, and there are no restrictions on use of the code.

A representative conversation with www.coreservlets.com is shown in Figure 17-1. Here, only a `GET` request was sent to the server, without any additional HTTP request headers. For more information on HTTP request headers, see Section 19.7 (The Client Request: HTTP Request Headers), and for HTTP response headers, see Section 19.10 (The Server Response: HTTP Response Headers).

Figure 17-1. A conversation with www.coreservlets.com shows a typical request and response.



Host: www.coreservlets.com

Port: 80

Request Line: GET / HTTP/1.0

Request Headers:

```
HTTP/1.0 200 OK
Date: Sat, 25 Nov 2000 00:55:14 GMT
Server: Apache/1.3.3 (Unix) PHP/3.0.11 FrontPage/4.0.4.3
Content-Type: text/html
Age: 15
Via: HTTP/1.0 cluster.lnh.md (Traffic-Server/4.0.0 [cMsSfW])

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Core Servlets and JavaServer Pages (JSP)</TITLE>
```

Listing 17.13 `WebClient.java`

```
import java.awt.*; // For BorderLayout, GridLayout, Font, Color.
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

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

public class WebClient extends JPanel
    implements Runnable, Interruptible, ActionListener {
```

```
  public static void main(String[] args) {
    WindowUtilities.setNativeLookAndFeel();
    WindowUtilities.openInJFrame(new WebClient(), 600, 700,
                                 "Web Client",
                                 SystemColor.control);
  }

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

  public WebClient() {
    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);
    JPanel inputPanel = new JPanel();
    inputPanel.setLayout(new BorderLayout());
    JPanel labelPanel = new JPanel();
    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);
    JLabel requestHeadersLabel =
      new JLabel("Request Headers:");
    requestHeadersLabel.setFont(labelFont);
    labelPanel.add(requestHeadersLabel);
    inputPanel.add(labelPanel, BorderLayout.NORTH);
    requestHeadersArea = new JTextArea(5, 80);
    requestHeadersArea.setFont(textFont);
    JScrollPane headerScrollArea =
      new JScrollPane(requestHeadersArea);
    inputPanel.add(headerScrollArea, BorderLayout.CENTER);
    JPanel buttonPanel = new JPanel();
    submitButton = new JButton("Submit Request");
    submitButton.addActionListener(this);
    submitButton.setFont(labelFont);
    buttonPanel.add(submitButton);
    inputPanel.add(buttonPanel, BorderLayout.SOUTH);
    add(inputPanel, BorderLayout.NORTH);
    JPanel resultPanel = new JPanel();
    resultPanel.setLayout(new BorderLayout());
    JLabel resultLabel =
      new JLabel("Results", JLabel.CENTER);
    resultLabel.setFont(headingFont);
    resultPanel.add(resultLabel, BorderLayout.NORTH);
    resultArea = new JTextArea();
    resultArea.setFont(textFont);
    JScrollPane resultScrollArea =
      new JScrollPane(resultArea);
    resultPanel.add(resultScrollArea, BorderLayout.CENTER);
    JPanel interruptPanel = new JPanel();
    interruptButton = new JButton("Interrupt Download");
    interruptButton.addActionListener(this);
    interruptButton.setFont(labelFont);
    interruptPanel.add(interruptButton);
    resultPanel.add(interruptPanel, BorderLayout.SOUTH);
    add(resultPanel, BorderLayout.CENTER);
  }
  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);
    }
    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();
    }
  }
}
```

Listing 17.14 `HttpClient.java`

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

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

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

  public HttpClient(String host, int port,
                    String requestLine, String[] requestHeaders,
                    JTextArea 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);
    }
  }

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

Listing 17.15 `LabeledTextField.java`

```
import java.awt.*; // For FlowLayout, Font.
import javax.swing.*;

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

public class LabeledTextField extends JPanel {
  private JLabel label;
  private JTextField textField;
```

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

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

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

/** The TextField at the right side of the
 *  LabeledTextField.
 */
```

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

Listing 17.16 **`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();
}
```

17.7 Implementing a Server

The server is the program that starts first and waits for incoming connections. Implementing a server consists of six basic steps:

1. Create a `ServerSocket` object.
2. Create a `Socket` object from the `ServerSocket`.
3. Create an input stream to read input from the client.
4. Create an output stream that can be used to send information back to the client.
5. Do I/O with input and output streams.
6. Close the `Socket` when done.

Each of these steps is described in more detail in the following sections. As with the client, note that most of the methods described throw an `IOException`, so they need to be wrapped inside a `try/catch` block in an actual implementation.

Create a `ServerSocket` object. With a client socket, you actively go out and connect to a particular system. With a server, however, you passively sit and wait for someone to come to you. So, creation requires only a port number, not a host, as follows:

```
ServerSocket listenSocket =
  new ServerSocket(portNumber);
```

On Unix, if you are a nonprivileged user, this port number *must* be greater than 1023 (lower numbers are reserved) and *should* be greater than 5000 (numbers from 1024 to 5000 are more likely to already be in use). In addition, you should check `/etc/services` to make sure your selected port doesn't conflict with other services running on the same port number. If you try to listen on a socket that is already in use, an `IOException` will be thrown.

Create a `Socket` object from the `ServerSocket`. Many servers allow multiple connections, continuing to accept connections until some termination condition is reached. The `ServerSocket accept` method blocks until a connection is established, then returns a normal `Socket` object. Here is the basic idea:

```
while(someCondition) {
  Socket server = listenSocket.accept();
  doSomethingWith(server);
}
```

If you want to allow multiple simultaneous connections to the socket, you should pass this socket to a separate thread to create the input/output streams. In the next section, we give an example of creating a separate thread for each connection.

Create an input stream to read input from the client. Once you have a `Socket`, you can use the socket in the same way as with the client code shown in Section 17.1. The example here shows the creation of an input stream before an output stream, assuming that most servers will read data before transmitting a reply. You can switch the order of this step and the next if you send data before reading, and even omit this step if your server only transmits information.

As was also discussed in the client section, a `BufferedReader` is more efficient to use underneath the `InputStreamReader`, as follows:

```
BufferedReader in =
  new BufferedReader
   (new InputStreamReader(server.getInputStream()));
```

Java also lets you use `ObjectInputStream` to receive complex objects from another Java program. An `ObjectInputStream` connected to the network is used in exactly the same way as one connected to a file; simply use `readObject` and cast the result to the appropriate type. See Section 13.9 (Serializing Windows) for more details and an example. Also see Section 17.9 (RMI: Remote Method Invocation) for a high-level interface that uses serialization to let you distribute Java objects across networks.

Create an output stream that can be used to send information back to the client. You can use a generic `OutputStream` if you want to send binary data. If you want to use the familiar `print` and `println` commands, create a `PrintWriter`. Here is an example of creating a `PrintWriter`:

```
PrintWriter out =
  new PrintWriter(server.getOutputStream());
```

In Java, you can use an `ObjectOutputStream` if the client is written in the Java programming language and is expecting complex Java objects.

Do I/O with input and output streams. The `BufferedReader`, `DataInputStream`, and `PrintWriter` classes can be used in the same ways as discussed in the client section earlier in this chapter. `BufferedReader` provides `read` and `readLine` methods for reading characters or strings. `DataInputStream` has `readByte` and `readFully` methods for reading a single byte or a byte array. Use `print` and `println` for sending high-level data through a `PrintWriter`; use `write` to send a byte or byte array.

Close the `Socket` when done. When finished, close the socket:

```
server.close();
```

This method closes the associated input and output streams but does *not* terminate any loop that listens for additional incoming connections.

Example: A Generic Network Server

Listing 17.17 gives a sample implementation of the approach outlined at the beginning of Section 17.7. Processing starts with the `listen` method, which waits until receiving a connection, then passes the socket to `handleConnection` to do the actual communication. Real servers might have handleConnection operate in a separate thread to allow multiple simultaneous connections, but even if not, they would override the method to provide the server with the desired behavior. The generic version of handleConnection simply reports the hostname of the system that made the connection, shows the first line of input received from the client, sends a single line to the client ("Generic Network Server"), then closes the connection.

Listing 17.17 NetworkServer.java

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

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

public class NetworkServer {
  private int port, maxConnections;

  /** Build a server on specified port. It will continue to
   *  accept connections, passing each to handleConnection until
   *  an explicit exit command is sent (e.g., System.exit) or
   *  the maximum number of connections is reached. Specify
   *  0 for maxConnections if you want the server to run
   *  indefinitely.
   */

  public NetworkServer(int port, int maxConnections) {
    setPort(port);
    setMaxConnections(maxConnections);
  }

  /** Monitor a port for connections. Each time one is
   *  established, pass resulting Socket to handleConnection.
   */

  public void listen() {
    int i=0;
    try {
      ServerSocket listener = new ServerSocket(port);
      Socket server;
      while((i++ < maxConnections) || (maxConnections == 0)) {
        server = listener.accept();
        handleConnection(server);
      }
    } catch (IOException ioe) {
      System.out.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }

  /** This is the method that provides the behavior to the
   *  server, since it determines what is done with the
   *  resulting socket. <B>Override this method in servers
   *  you write.</B>
   *  <P>
   *  This generic version simply reports the host that made
   *  the connection, shows the first line the client sent,
   *  and sends a single line in response.
   */

  protected void handleConnection(Socket server)
      throws IOException{
    BufferedReader in = SocketUtil.getReader(server);
    PrintWriter out = SocketUtil.getWriter(server);
    System.out.println
      ("Generic Network Server: got connection from " +
       server.getInetAddress().getHostName() + "\n" +
       "with first line '" + in.readLine() + "'");
    out.println("Generic Network Server");
    server.close();
  }

  /** Gets the max connections server will handle before
   *  exiting. A value of 0 indicates that server should run
   *  until explicitly killed.
   */

  public int getMaxConnections() {
    return(maxConnections);
  }

  /** Sets max connections. A value of 0 indicates that server
   *  should run indefinitely (until explicitly killed).
   */

  public void setMaxConnections(int maxConnections) {
    this.maxConnections = maxConnections;
  }
  /** Gets port on which server is listening. */

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

  /** Sets port. <B>You can only do before "connect" is
   *  called.</B> That usually happens in the constructor.
   */

  protected void setPort(int port) {
    this.port = port;
  }
}
```

Finally, the `NetworkServerTest` class provides a way to invoke the `NetworkServer` class on a specified port, shown in Listing 17.18.

Listing 17.18 `NetworkServerTest.java`

```
public class NetworkServerTest {
public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      port = Integer.parseInt(args[0]);
    }
    NetworkServer nwServer = new NetworkServer(port, 1);
    nwServer.listen();
  }
}
```

Output: Accepting a Connection from a WWW Browser

Suppose the test program in Listing 17.18 is started on port 8088 of `system1.com`:

```
system1> java NetworkServerTest
```

Then, a standard Web browser (Netscape in this case) on `system2.com` requests http://system1.com:8088/foo/:, yielding the following back on `system1.com`:

```
Generic Network Server:
got connection from system2.com
with first line 'GET /foo/ HTTP/1.0'
```

Connecting NetworkClient and NetworkServer

OK, we showed the `NetworkClient` and `NetworkServer` classes tested separately, with the client talking to a standard FTP server and the server talking to a standard Web browser. However, we can also connect them to each other. No changes in the source code are required; simply specify the appropriate hostnames and port. The test server is started on port 6001 of `system1.com`, then the client is started on `system2.com`, yielding the following results:

Time t_0, system1:

```
system1> java NetworkServerTest 6001
```

Time t_1, system2:

```
system2> java NetworkClientTest system1.com 6001
```

Time t_2, system1:

```
Generic Network Server:
got connection from system2.com
with first line 'Generic Network Client'
```

Time t_3, system2:

```
Generic Network Client:
Made connection to system1.com and got 'Generic Network
Server'in response
```

17.8 Example: A Simple HTTP Server

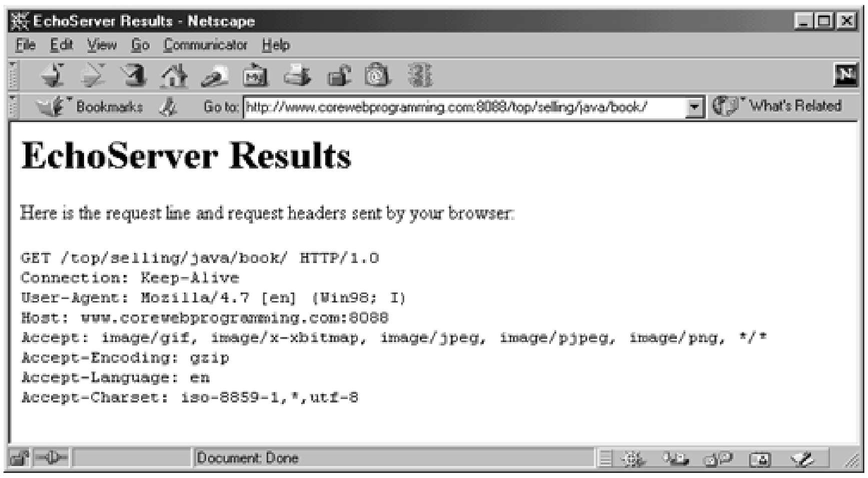
In Listing 17.19 we adapt the `NetworkServer` class to act as an HTTP server. Rather than returning files, however, we have the server simply echo back the received input by storing all of the input lines, then transmit back an HTML file that shows the sent line. Although writing programs that output HTML seems odd, in Chapter 19 (Server-Side Java: Servlets) and Chapter 24 (JavaScript: Adding Dynamic Content to Web Pages) you'll see that this is actually common practice. Furthermore, having a program that can act as an HTTP server but returns a Web page showing the received input is a useful debugging tool when you are working with HTTP clients and servlet or JSP programming. You'll see this class used many times in the HTTP and servlet chapters.

Listing 17.19 `EchoServer.java`

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

/** A simple HTTP server that generates a Web page showing all
 *  of the data that it received from the Web client (usually
 *  a browser). To use this server, start it on the system of
 *  your choice, supplying a port number if you want something
 *  other than port 8088. Call this system server.com. Next,
 *  start a Web browser on the same or a different system, and
 *  connect to http://server.com:8088/whatever. The resultant
 *  Web page will show the data that your browser sent. For
 *  debugging in servlet or CGI programming, specify
 *  http://server.com:8088/whatever as the ACTION of your HTML
 *  form. You can send GET or POST data; either way, the
 *  resultant page will show what your browser sent.
 */

public class EchoServer extends NetworkServer {
  protected int maxRequestLines = 50;
  protected String serverName = "EchoServer";

  /** Supply a port number as a command-line
   *  argument. Otherwise, use port 8088.
   */

  public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      try {
        port = Integer.parseInt(args[0]);
      } catch(NumberFormatException nfe) {}
    }
    new EchoServer(port, 0);
  }

  public EchoServer(int port, int maxConnections) {
    super(port, maxConnections);
    listen();
  }
  /** Overrides the NetworkServer handleConnection method to
   *  read each line of data received, save it into an array
   *  of strings, then send it back embedded inside a PRE
   *  element in an HTML page.
   */

  public void handleConnection(Socket server)
      throws IOException{
    System.out.println
        (serverName + ": got connection from " +
          server.getInetAddress().getHostName());
    BufferedReader in = SocketUtil.getReader(server);
    PrintWriter out = SocketUtil.getWriter(server);
    String[] inputLines = new String[maxRequestLines];
    int i;
    for (i=0; i<maxRequestLines; i++) {
      inputLines[i] = in.readLine();
      if (inputLines[i] == null) // Client closed connection.
        break;
      if (inputLines[i].length() == 0) { // Blank line.
        if (usingPost(inputLines)) {
          readPostData(inputLines, i, in);
          i = i + 2;
        }
        break;
      }
    }
    printHeader(out);
    for (int j=0; j<i; j++) {
      out.println(inputLines[j]);
    }
    printTrailer(out);
    server.close();
  }

  // Send standard HTTP response and top of a standard Web page.
  // Use HTTP 1.0 for compatibility with all clients.

  private void printHeader(PrintWriter out) {
    out.println
      ("HTTP/1.0 200 OK\r\n" +
       "Server: " + serverName + "\r\n" +
       "Content-Type: text/html\r\n" +
       "\r\n" +
       "<HTML>\n" +
       "<!DOCTYPE HTML PUBLIC " +
         "\"-//W3C//DTD HTML 4.0 Transitional//EN\">\n" +
       "<HEAD>\n" +
       "  <TITLE>" + serverName + " Results</TITLE>\n" +
       "</HEAD>\n" +
       "\n" +
       "<BODY BGCOLOR=\"#FDF5E6\">\n" +
       "<H1 ALIGN=\"CENTER\">" + serverName +
         " Results</H1>\n" +
       "Here is the request line and request headers\n" +
       "sent by your browser:\n" +
       "<PRE>");
  }

  // Print bottom of a standard Web page.

  private void printTrailer(PrintWriter out) {
    out.println
      ("</PRE>\n" +
       "</BODY>\n" +
       "</HTML>\n");
  }

  // Normal Web page requests use GET, so this server can simply
  // read a line at a time. However, HTML forms can also use
  // POST, in which case we have to determine the number of POST
  // bytes that are sent so we know how much extra data to read
  // after the standard HTTP headers.

  private boolean usingPost(String[] inputs) {
    return(inputs[0].toUpperCase().startsWith("POST"));
  }

  private void readPostData(String[] inputs, int i,
                            BufferedReader in)
      throws IOException {
    int contentLength = contentLength(inputs);
    char[] postData = new char[contentLength];
    in.read(postData, 0, contentLength);
    inputs[++i] = new String(postData, 0, contentLength);
  }
  // Given a line that starts with Content-Length,
  // this returns the integer value specified.

  private int contentLength(String[] inputs) {
    String input;
    for (int i=0; i<inputs.length; i++) {
      if (inputs[i].length() == 0)
        break;
      input = inputs[i].toUpperCase();
      if (input.startsWith("CONTENT-LENGTH"))
        return(getLength(input));
    }
    return(0);
  }

  private int getLength(String length) {
    StringTokenizer tok = new StringTokenizer(length);
    tok.nextToken();
    return(Integer.parseInt(tok.nextToken()));
  }
}
```

Figure 17-2 shows the `EchoServer` in action, displaying the header lines sent by Netscape 4.7 on Windows 98.

Figure 17-2. The `EchoServer` shows data sent by the browser.


EchoServer Results - Netscape
File Edit View Go Communicator Help
Bookmarks Go to: http://www.corewebprogramming.com:8088/top/selling/java/book/
What's Related
Document: Done

EchoServer Results

Here is the request line and request headers sent by your browser:

```
GET /top/selling/java/book/ HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (Win98; I)
Host: www.corewebprogramming.com:8088
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

ThreadedEchoServer: Adding Multithreading

The problem with the `EchoServer` is that the service can only accept one connection at a time. If, for instance, it takes 0.001 seconds to establish a connection but 0.01 seconds for the client to transmit the request and 0.01 seconds for the server to return the results, then the entire process takes about 0.02 seconds, and the server can only handle about 50 connections per second. By doing the socket processing in a separate thread, establishing the connection becomes the rate-limiting step, and the server could handle about 1,000 connections per second with these example times.

Listing 17.20 shows how to convert the `EchoServer` into a multithreaded version. Section 16.4 (Creating a Multithreaded Method) discusses in detail the process for converting a single-threaded method to a multithreaded method. The basic idea is that the new version's `handleConnection` starts up a thread, which calls back to the original `handleConnection`. The problem is how to get the `Socket` object from `handleConnection` to `run`, because placing the `Socket` object in an instance variable would subject it to race conditions. So, a `Connection` class, which is simply a `Thread` with a place to store the `Socket` object, is used.

Listing 17.20 `ThreadedEchoServer.java`

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

/** A multithreaded variation of EchoServer. */

public class ThreadedEchoServer extends EchoServer
                                implements Runnable {
  public static void main(String[] args) {
    int port = 8088;
    if (args.length > 0) {
      try {
        port = Integer.parseInt(args[0]);
      } catch(NumberFormatException nfe) {}
    }
    ThreadedEchoServer echoServer =
      new ThreadedEchoServer(port, 0);
    echoServer.serverName = "Threaded EchoServer";
  }

  public ThreadedEchoServer(int port, int connections) {
    super(port, connections);
  }
  /** The new version of handleConnection starts a thread. This
   *  new thread will call back to the <I>old</I> version of
   *  handleConnection, resulting in the same server behavior
   *  in a multithreaded version. The thread stores the Socket
   *  instance since run doesn't take any arguments, and since
   *  storing the socket in an instance variable risks having
   *  it overwritten if the next thread starts before the run
   *  method gets a chance to copy the socket reference.
   */

  public void handleConnection(Socket server) {
    Connection connectionThread = new Connection(this, server);
    connectionThread.start();
  }

  public void run() {
    Connection currentThread =
      (Connection)Thread.currentThread();
    try {
      super.handleConnection(currentThread.getSocket());
    } catch(IOException ioe) {
      System.out.println("IOException: " + ioe);
      ioe.printStackTrace();
    }
  }
}

/** This is just a Thread with a field to store a Socket object.
 *  Used as a thread-safe means to pass the Socket from
 *  handleConnection to run.
 */


class Connection extends Thread {
  private Socket serverSocket;

  public Connection(Runnable serverObject,
                    Socket serverSocket) {
    super(serverObject);
    this.serverSocket = serverSocket;
  }

  public Socket getSocket() {
    return serverSocket;
  }
}
```

This server gives the same results as the `EchoServer` but allows multiple simultaneous connections.

17.9 RMI: Remote Method Invocation

Java directly supports distributing run-time objects across multiple computers through Remote Method Invocation (RMI). This distributed-objects package simplifies communication among Java applications on multiple machines. If you are already familiar with the Common Object Request Broker Architecture, think of RMI as a simpler but less powerful variation of CORBA that only works with Java systems. If you don't know anything about CORBA, think of RMI as an object-oriented version of remote procedure calls (RPC).

The idea is that the client requests an object from the server, using a simple high-level request. Once the client has the object, the client invokes the object's methods as though the object were a normal local object. Behind the scenes, however, the requests are routed to the server, where methods in the "real" object are invoked and the results returned. The beauty of the process is that neither the client nor the server has to do anything explicit with input streams, output streams, or sockets. The values that are sent back and forth can be complex Java objects (including windows and other graphical components), but no parsing is required at either end. The conversion of the object is handled by the Java serialization facility.

Now, RMI seems so convenient that you might wonder why anyone would implement sockets "by hand." First of all, RMI only works among Java systems, so RMI cannot be used for an HTTP client, an HTTP server, an e-mail client, or other applications where the other end will not necessarily be using Java. Second, even for Java-to-Java applications, RMI requires some common code to be installed on both the client and the server. This approach is in contrast to sockets, where random programs can talk to each other as long as they both understand the types of commands that should be sent. Finally, RMI is a bit more taxing on the server than regular sockets because RMI requires two versions of the Java Virtual Machine to be running (one to broker requests for objects, the other to provide the actual object implementation).

Following, we summarize the steps necessary to build an RMI application. Afterward, we present four RMI examples:

1. A simple RMI example that returns a message string from a remote object.
2. A realistic example that performs numerical integration through a remote object.
3. Extension of the numerical integration to an enterprise configuration showing how to set up a security policy file and HTTP server for downloading of RMI files.
4. An RMI applet that connects to a remote object.

Steps to Build an RMI Application

To use RMI, you need to do two things: build four classes and execute five compilation steps. We briefly describe the classes and the steps here. In the next subsections, we build the classes—simple and more advanced—and show the command-line commands that execute the steps, along with the output, if any, of the commands.

The Four Required Classes

To use RMI, you will need to build four main classes:

1. **An interface for the remote object.** This interface will be used by both the client and the server.
2. **The RMI client.** This client will look up the object on the remote server, cast the object to the type of the interface from Step 1, then use the object like a local object. Note that as long as a "live" reference to the remote object is present, the network connection will be maintained. The connection will be automatically closed when the remote object is garbage-collected on the client.
3. **The object implementation.** This object must implement the interface of Step 1 and will be used by the server.
4. **The RMI server.** This class will create an instance of the object from Step 3 and register the object with a particular URL.

Compiling and Running the System

Once you have the basic four classes, five further steps are required to actually use the application.

1. **Compile client and server.** This step automatically compiles the remote object interface and implementation.
2. **Generate the client stub and the server skeleton.** The client stub and server skeleton support the

method calls and provide parameter marshalling (device-independent coding and serialization for transmission across a byte stream). Use the `rmic` compiler on the remote object implementation for this step.

The client system will need the client class, the interface class, and the client stub class. If the client is an applet, these three classes must be available from the applet's home machine.

The server system will need the server class, the remote object interface and implementation, and the server skeleton class. Note that Java 2 no longer requires the skeleton class normally placed on the server. If both the server and client are running the Java 2 Platform, then use the `-v1.2` switch for the `rmic` compiler.

3. **Start the RMI registry.** This step needs to be done only once, not for each remote object. The current version of RMI requires this registry to be running on the same system as the server.
4. **Start the server.** This step must be done on the same machine as the registry of Step 3.
5. **Start the client.** This step can be on an arbitrary machine.

A Simple Example

Here's a simple example to illustrate the process. The remote object simply returns a message string. See the next subsection for a more realistic example.

A Simple Example of the Four Required Classes

1. **The interface for the remote object.** The interface should extend `java.rmi.Remote`, and all the methods should throw `java.rmi.RemoteException`. Listing 17.21 shows an example.

   Listing 17.21 `Rem.java`

```
import java.rmi.*;

/** The RMI client will use this interface directly. The RMI
 *  server will make a real remote object that implements this,
 *  then register an instance of it with some URL.
 */

public interface Rem extends Remote {
  public String getMessage() throws RemoteException;
}
```

2. **The RMI client.** This class should look up the object from the appropriate host, using `Naming.lookup`, cast the object to the appropriate type, then use the object like a local object. Unlike the case in CORBA, RMI clients must know the host that is providing the remote services. The URL can be specified by `rmi://host/path` or `rmi://host:port/path`. If the port is omitted, 1099 is used. This process can throw three possible exceptions: `RemoteException`, `NotBoundException`, and `MalformedURLException`. You are required to catch all three. You should import `java.rmi.*` for `RemoteException`, `Naming`, and `NotBoundException`. You should import `java.net.*` for `MalformedURLException`. In addition, many clients will pass `Serializable` objects to the remote object, so importing `java.io.*` is a good habit, even though it is not required in this particular case. Listing 17.22 shows an example.

   Listing 17.22 `RemClient.java`

```
import java.rmi.*; // For Naming, RemoteException, etc.
import java.net.*; // For MalformedURLException
import java.io.*;  // For Serializable interface

/** Get a Rem object from the specified remote host.
 *  Use its methods as though it were a local object.
 */

public class RemClient {
  public static void main(String[] args) {
    try {
      String host =
        (args.length > 0) ? args[0] : "localhost";
      // Get the remote object and store it in remObject:
      Rem remObject =
        (Rem)Naming.lookup("rmi://" + host + "/Rem");
      // Call methods in remObject:
      System.out.println(remObject.getMessage());
    } catch(RemoteException re) {
      System.out.println("RemoteException: " + re);
    } catch(NotBoundException nbe) {
      System.out.println("NotBoundException: " + nbe);
    } catch(MalformedURLException mfe) {
      System.out.println("MalformedURLException: " + mfe);
    }
  }
}
```

3. **The remote object implementation.** This class must extend `UnicastRemoteObject` and implement the remote object interface defined earlier. The constructor should throw `RemoteException`. Listing 17.23 shows an example.

Listing 17.23 `RemImpl.java`

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

/** This is the actual implementation of Rem that the RMI
 *  server uses. The server builds an instance of this, then
 *  registers it with a URL. The client accesses the URL and
 *  binds the result to a Rem (not a RemImpl; it doesn't
 *  have this).
 */

public class RemImpl extends UnicastRemoteObject
                     implements Rem {
  public RemImpl() throws RemoteException {}

  public String getMessage() throws RemoteException {
    return("Here is a remote message.");
  }
}
```

4. **The RMI server.** The server's job is to build an object and register the object with a particular URL. Use `Naming.rebind` (replace any previous bindings) or `Naming.bind` (throw `AlreadyBoundException` if a previous binding exists) to register the object. (The term "bind" is used differently than in the CORBA world; here, bind means "register" and is performed by the server, not the client.) You are required to catch `RemoteException` and

MalformedURLException. Listing 17.24 shows an example.

Listing 17.24 RemServer.java

```
import java.rmi.*;
import java.net.*;

/** The server creates a RemImpl (which implements the Rem
 *  interface), then registers it with the URL Rem, where
 *  clients can access it.
 */

public class RemServer {
  public static void main(String[] args) {
    try {
      RemImpl localObject = new RemImpl();
      Naming.rebind("rmi:///Rem", localObject);
    } catch(RemoteException re) {
      System.out.println("RemoteException: " + re);
    } catch(MalformedURLException mfe) {
      System.out.println("MalformedURLException: " + mfe);
    }
  }
}
```

Compiling and Running the System for the Simple Example

As outlined earlier in this section, compiling and running the system requires five steps.

For this example, you must start the RMI registry, the server (RemServer), and the client (RemClient) in the same host directory. If the client and server are started from different directories, then the RMI protocol requires a SecurityManager on the client to load the stub files. Configuration of RMI to run the client and server on different hosts (or different directories) is explained later.

Core Note

For the following example to execute properly, the RMI registry, server, and client must be started from the same directory.

1. **Compile the client and the server.** The following command automatically compiles the Rem interface.

   ```
   Prompt> javac RemClient.java
   ```

   The following command automatically compiles the RemImpl object implementation.

   ```
   Prompt> javac RemServer.java
   ```

2. **Generate the client Stub and server Skeleton.** The following command builds RemImpl_Stub.class and RemImpl_Skeleton.class.

   ```
   Prompt> rmic RemImpl
   ```

   or

   ```
   Prompt> rmic -v1.2 RemImpl
   ```
   (for Java 2 Platform)

The client requires `Rem.class`, `RemClient.class`, and `RemImpl_Stub.class`. The server requires `Rem.class`, `RemImpl.class`, `RemServer.class`, and `RemImpl_Skeleton.class`.

For the Java 2 platform, the `RemImpl_Skeleton.class` is no longer required. To generate only the stub file required for the Java 2 Platform, add the `-v1.2` switch in the RMI compiler command. This switch generates a stub file consistent with the RMI 1.2 stub protocol used by the Java 2 Platform. By default, `rmic` creates stubs and skeletons compatible with both the RMI 1.2 stub protocol and the earlier RMI 1.1 stub protocol used in JDK 1.1.

Core Note

If the client and server are both running the Java 2 Platform, use `rmic -v1.2` *to compile the interface. Using the* `-v1.2` *switch does not generate the unnecessary skeleton file.*

3. **Start the RMI registry.** Start the registry as follows.

```
Prompt> rmiregistry
```

On Unix systems, you would probably add `&` to put the registry process in the background. On Windows, you would probably precede the command with `start`, as in `start rmiregistry`. You can also specify a port number; if omitted, port 1099 is used.

4. **Start the server.** Start the server as follows.

```
Server> java RemServer
```

Again, on Unix systems, you would probably add `&` to put the process in the background. On Windows, you would probably precede the command with `start`, as in `start java RemServer`.

5. **Start the client.** Issue the following command.

```
Prompt> java RemClient
Here is a remote message.
```

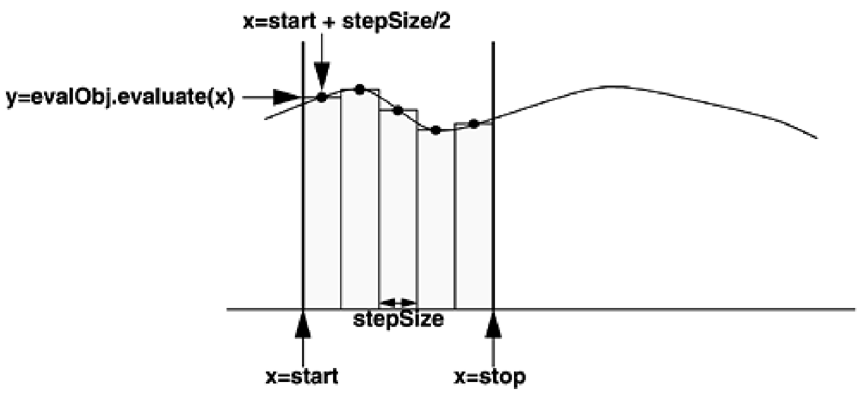
A Realistic Example: A Server for Numeric Integration

Listing 17.25 shows a class that provides two methods. The first method, `sum`, calculates

$$\sum_{x = start}^{stop} f(x)$$

The definition of *f*(*x*) is provided by an `Evaluatable` object (Listing 17.26). The second method, `integrate`, uses the midpoint rule (Figure 17-3) to approximate the integral

Figure 17-3. The `integrate` method approximates the area under the curve by adding up the area of many small rectangles that have width `stepSize` and whose length *y = f(x)* is evaluated at the midpoint of each width.



$$\int_{start}^{stop} f(x)dx$$

Listing 17.25 **Integral.java**

```
/** A class to calculate summations and numeric integrals. The
 *  integral is calculated according to the midpoint rule.
 */

public class Integral {
  /** Returns the sum of f(x) from x=start to x=stop, where the
   *  function f is defined by the evaluate method of the
   *  Evaluatable object.
   */

  public static double sum(double start, double stop,
                           double stepSize,
                           Evaluatable evalObj) {
    double sum = 0.0, current = start;
    while (current <= stop) {
      sum += evalObj.evaluate(current);
      current += stepSize;
    }
    return(sum);
  }

  /** Returns an approximation of the integral of f(x) from
   *  start to stop, using the midpoint rule. The function f is
   *  defined by the evaluate method of the Evaluatable object.
   */

  public static double integrate(double start, double stop,
                                 int numSteps,
                                 Evaluatable evalObj) {
    double stepSize = (stop - start) / (double)numSteps;
    start = start + stepSize / 2.0;
    return(stepSize * sum(start, stop, stepSize, evalObj));
  }
}
```

Listing 17.26 `Evaluatable.java`

```
/** An interface for evaluating functions y = f(x) at a specific
 *  value. Both x and y are double-precision floating-point
 *  numbers.
 */

public interface Evaluatable {
  public double evaluate(double value);
}
```

Now suppose that you have a powerful workstation that has very fast floating-point capabilities and a variety of slower PCs that need to run an interface that makes use of numerical integration. A natural approach is to make the workstation the integration server. RMI makes this solution very simple.

A Realistic Example of the Four Required Classes

In this section we provide listings for the four classed required to establish a workstation and integration server.

1. **The RemoteIntegral interface.** Listing 17.27 shows the interface that will be shared by the client and server.

   Listing 17.27 `RemoteIntegral.java`

   ```
   import java.rmi.*;

   /** Interface for remote numeric integration object. */

   public interface RemoteIntegral extends Remote {

     public double sum(double start, double stop, double stepSize,
                       Evaluatable evalObj)

       throws RemoteException;

     public double integrate(double start, double stop,
                             int numSteps, Evaluatable evalObj)

       throws RemoteException;
   }
   ```

2. **The RemoteIntegral client.** Listing 17.28 shows the RMI client. It obtains a `RemoteIntegral` from the specified host, then uses it to approximate a variety of integrals. Note that the `Evaluatable` instances (`Sin`, `Cos`, `Quadratic`) implement `Serializable` in addition to `Evaluatable` so that these objects can be transmitted over the network. The `Sin`, `Cos`, and `Quadratic` classes are shown in Listing 17.29, 17.30, and 17.31, respectively. The `toString` method in each of the three classes is used later in the RMI Applet example.

   Listing 17.28 `RemoteIntegralClient.java`

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

```
/** This class calculates a variety of numerical integration
 *  values, printing the results of successively more accurate
 *  approximations. The actual computation is performed on a
 *  remote machine whose hostname is specified as a command-
 *  line argument.
 */

public class RemoteIntegralClient {
  public static void main(String[] args) {
    try {
      String host = (args.length > 0) ? args[0] : "localhost";
      RemoteIntegral remoteIntegral =
        (RemoteIntegral)Naming.lookup("rmi://" + host +
                                      "/RemoteIntegral");
      for(int steps=10; steps<=10000; steps*=10) {
        System.out.println
          ("Approximated with " + steps + " steps:" +
           "\n  Integral from 0 to pi of sin(x)=" +
           remoteIntegral.integrate(0.0, Math.PI,
                                    steps, new Sin()) +
           "\n  Integral from pi/2 to pi of cos(x)=" +
           remoteIntegral.integrate(Math.PI/2.0, Math.PI,
                                    steps, new Cos()) +
           "\n  Integral from 0 to 5 of x^2=" +
           remoteIntegral.integrate(0.0, 5.0, steps,
                                    new Quadratic()));
      }
      System.out.println
        ("'Correct' answer using Math library:" +
         "\n  Integral from 0 to pi of sin(x)=" +
         (-Math.cos(Math.PI) - -Math.cos(0.0)) +
         "\n  Integral from pi/2 to pi of cos(x)=" +
         (Math.sin(Math.PI) - Math.sin(Math.PI/2.0)) +
         "\n  Integral from 0 to 5 of x^2=" +
         (Math.pow(5.0, 3.0) / 3.0));
    } catch(RemoteException re) {
      System.out.println("RemoteException: " + re);
    } catch(NotBoundException nbe) {
      System.out.println("NotBoundException: " + nbe);
    } catch(MalformedURLException mfe) {
      System.out.println("MalformedURLException: " + mfe);
    }
  }
}
```

Listing 17.29 `Sin.java`

```
import java.io.Serializable;

/** An evaluatable version of sin(x). */

class Sin implements Evaluatable, Serializable {
  public double evaluate(double val) {
    return(Math.sin(val));
  }
  public String toString() {
    return("Sin");
  }
}
```

Listing 17.30 **`Cos.java`**

```
import java.io.Serializable;

/** An evaluatable version of cos(x). */

class Cos implements Evaluatable, Serializable {
  public double evaluate(double val) {
    return(Math.cos(val));
  }

  public String toString() {
    return("Cosine");
  }
}
```

Listing 17.31 **`Quadratic.java`**

```
import java.io.Serializable;

/** An evaluatable version of x^2. */

class Quadratic implements Evaluatable, Serializable {
  public double evaluate(double val) {
    return(val * val);
  }

  public String toString() {
    return("Quadratic");
  }
}
```

3. **The RemoteIntegral implementation.** Listing 17.32 shows the implementation of the `RemoteIntegral` interface. It simply uses methods in the `Integral` class.

Listing 17.32 **`RemoteIntegralImpl.java`**

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

/** The actual implementation of the RemoteIntegral interface.
 */

public class RemoteIntegralImpl extends UnicastRemoteObject
                                implements RemoteIntegral {

  /** Constructor must throw RemoteException. */

  public RemoteIntegralImpl() throws RemoteException {}
  /** Returns the sum of f(x) from x=start to x=stop, where the
   *  function f is defined by the evaluate method of the
   *  Evaluatable object.
   */

  public double sum(double start, double stop, double stepSize,
                    Evaluatable evalObj) {
    return(Integral.sum(start, stop, stepSize, evalObj));
  }

  /** Returns an approximation of the integral of f(x) from
   *  start to stop, using the midpoint rule. The function f is
   *  defined by the evaluate method of the Evaluatable object.
   * @see #sum
   */

  public double integrate(double start, double stop, int numSteps,
                          Evaluatable evalObj) {
    return(Integral.integrate(start, stop, numSteps, evalObj));
  }
}
```

4. **The RemoteIntegral server.** Listing 17.33 shows the server that creates a `RemoteIntegralImpl` object and registers the object with the URL `RemoteIntegral` on the local system.

Listing 17.33 `RemoteIntegralServer.java`

```
import java.rmi.*;
import java.net.*;

/** Creates a RemoteIntegralImpl object and registers it under
 *  the name 'RemoteIntegral' so that remote clients can connect
 *  to it for numeric integration results. The idea is to place
 *  this server on a workstation with very fast floating-point
 *  capabilities, while slower interfaces can run on smaller
 *  computers but still use the integration routines.
 */

public class RemoteIntegralServer {
  public static void main(String[] args) {
    try {
      RemoteIntegralImpl integral =  new RemoteIntegralImpl();
      Naming.rebind("rmi:///RemoteIntegral", integral);
    } catch(RemoteException re) {
      System.out.println("RemoteException: " + re);
    } catch(MalformedURLException mfe) {
      System.out.println("MalformedURLException: " + mfe);
    }
  }
}
```

Compiling and Running the System for the Realistic Example

For this example, you must start the RMI registry, the server (`RemoteIntegralServer`), and the client (`RemoteIntegralClient`) in the same host directory. If the client and server are started from different directories, then the RMI protocol requires a `SecurityManager` on the client to load the stub files. Configuration of RMI to run the client and server on different hosts (or different directories) is explained following this example.

Core Note

For the following example to execute properly, the RMI registry, server, and client must be started from the same directory.

1. **Compile the client and server.** At the prompt, enter these commands:

```
Prompt> javac RemoteIntegralClient.java
Prompt> javac RemoteIntegralServer.java
```

2. **Generate the client Stub and server Skeleton.** At the prompt, enter this command.

```
Prompt> rmic -v1.2 RemoteIntegralImpl
```

   The classes required by the client are: `RemoteIntegral.class`, `RemoteIntegralClient.class`. and `RemoteIntegralImpl_Stub.class`. The classes required by the server are: `RemoteIntegral.class`, `RemoteIntegralImpl.class`, and `RemoteIntegralServer.class`. If the server and client are both running JDK 1.1, use the `-v1.1` switch to produce the RMI 1.1 skeleton stub, `RemoteIntegralImpl_Skeleton`, required by the server.

3. **Start the RMI registry.** The following command starts the RMI registry:

```
Prompt> rmiregistry
```

4. **Start the Server.** At the prompt, enter this command:

```
Prompt> java RemoteIntegralServer
```

5. **Start the client.** At the prompt, enter this command to obtain the output listed below:

```
Prompt> java RemoteIntegralClient
Approximated with 10 steps:
  Integral from 0 to pi of sin(x)=2.0082484079079745
  Integral from pi/2 to pi of cos(x)=-1.0010288241427086
  Integral from 0 to 5 of x^2=41.5625
Approximated with 100 steps:
  Integral from 0 to pi of sin(x)=2.0000822490709877
  Integral from pi/2 to pi of cos(x)=-1.000010280911902
  Integral from 0 to 5 of x^2=41.665624999999906
Approximated with 1000 steps:
  Integral from 0 to pi of sin(x)=2.0000008224672983
  Integral from pi/2 to pi of cos(x)=-1.000000102808351
  Integral from 0 to 5 of x^2=41.666656249998724
Approximated with 10000 steps:
  Integral from 0 to pi of sin(x)=2.00000000822436
  Integral from pi/2 to pi of cos(x)=-1.0000000010278831
  Integral from 0 to 5 of x^2=41.666666562504055

'Correct' answer using Math library:
  Integral from 0 to pi of sin(x)=2.0
  Integral from pi/2 to pi of cos(x)=-0.9999999999999999
```

```
      Integral from 0 to 5 of x^2=41.666666666666664
```

The actual integral value are:

$$\int_0^\pi \sin x \, dx = 2$$

$$\int_{\pi/2}^\pi \cos x \, dx = -1$$

$$\int_0^5 x^2 dx = 41\frac{2}{3}$$

As the number of steps increases, the numerical integration approaches the actual value. The benefit of RMI is that you can off-load the integration to a more powerful server.

Enterprise RMI Configuration

In the previous examples, the RMI registry, server, and client, were all assumed to be running on the same host. Certainly, this configuration does not take advantage of the distributed capabilities of RMI. However, once the server and client are running on different hosts, Java 2 requires that the client and server have an installed security manager to load the RMI classes remotely; by default, the required RMI classes can only be loaded from the local host. Furthermore, the RMI registry and server must be started on the same host. The `rmiregistry` only permits registration of remote objects from the local host.

In addition to a security manager, changes in the default security policies are required to allow the client to open connections to remote hosts. Following, we show you the steps to use RMI in an enterprise environment where the server and client are located on different machines.

To load classes remotely, the client must install an `RMISecurityManager`

```
System.setSecurityManager(new RMISecurityManager());
```

as shown in the modified client, `RemoteIntegralClient2` (Listing 17.34).

Core Note

In an enterprise configuration, the `rmiregistry` and server must be started on the same host; otherwise, an `AccessException` is thrown. Additionally, the client must provide a policy file and set an `RMISecurityManager` to remotely load the stub files.

Listing 17.34 `RemoteIntegralClient2.java`

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

/** This class is a Java 2 version of RemoteIntegralClient
 *  that imposes a SecurityManager to allow the client to
 *  connect to a remote machine for loading stub files and
 *  performing numerical integration through a remote
 *  object.
 */

public class RemoteIntegralClient2 {
  public static void main(String[] args) {
    try {
      System.setSecurityManager(new RMISecurityManager());
      String host =
        (args.length > 0) ? args[0] : "localhost";
      RemoteIntegral remoteIntegral =
        (RemoteIntegral)Naming.lookup("rmi://" + host +
                                      "/RemoteIntegral");
      for(int steps=10; steps<=10000; steps*=10) {
        System.out.println
          ("Approximated with " + steps + " steps:" +
           "\n  Integral from 0 to pi of sin(x)=" +
           remoteIntegral.integrate(0.0, Math.PI,
                                    steps, new Sin()) +
           "\n  Integral from pi/2 to pi of cos(x)=" +
           remoteIntegral.integrate(Math.PI/2.0, Math.PI,
                                    steps, new Cos()) +
           "\n  Integral from 0 to 5 of x^2=" +
           remoteIntegral.integrate(0.0, 5.0, steps,
                                    new Quadratic()));
      }
      System.out.println
        ("'Correct' answer using Math library:" +
         "\n  Integral from 0 to pi of sin(x)=" +
         (-Math.cos(Math.PI) - -Math.cos(0.0)) +
         "\n  Integral from pi/2 to pi of cos(x)=" +
         (Math.sin(Math.PI) - Math.sin(Math.PI/2.0)) +
         "\n  Integral from 0 to 5 of x^2=" +
         (Math.pow(5.0, 3.0) / 3.0));
    } catch(RemoteException re) {
      System.out.println("RemoteException: " + re);
    } catch(NotBoundException nbe) {
      System.out.println("NotBoundException: " + nbe);
    } catch(MalformedURLException mfe) {
      System.out.println("MalformedURLException: " + mfe);
    }
  }
}
```

In Java 2, in addition to imposing a security manager, the client requires a policy file to specify permission for dynamic loading of remote classes. Basically, you need to state in which host and ports the client can open a socket connection. Listing 17.35 illustrates the client permissions to connect to the `rmiregistry` and server running on *rmihost* and to an HTTP server, *webhost,* to load the stub files.

By default the `rmiregistry` listens on port 1099. When the client looks up the remote object, it first connects to the `rmiregistry` on port 1099. Afterward, the client communicates directly to the remote server on the port at which the server is listening. Note that when the server is started and registers the remote object, the source port used to communicate with the RMI registry and client is randomly selected from an available port in the range 1024–65535 on the host. As a consequence, to allow connection to the rmihost, permission is required over the complete range of possible ports, 1024–65535, not just port 1099. Also, the stub classes are often placed on an HTTP server, so the policy file should also grant permission for the client to connect to the webhost.

Listing 17.35 Policy file for client, `rmiclient.policy`

```
grant {
  // rmihost - RMI registry and the server
  // webhost - HTTP server for stub classes
  permission java.net.SocketPermission
    "rmihost:1024-65535", "connect";
  permission java.net.SocketPermission
    "webhost:80", "connect";
};
```

You can specify the security policy file through a command-line argument when executing the client, as in

```
java -Djava.security.policy=rmiclient.policy RemoteIntegralClient2
```

Or, you can add the permission statements to the `java.policy` file used by the Java Virtual Machine and avoid the command-line argument. For JDK 1.3, the `java.policy` file is located in the directory `/root/jdk1.3/lib/security/`.

When starting the server that registers the remote object, you also need to specify the codebase location (HTTP server) to load the stub files. As with the policy file, you can specify the codebase on the command line through the system property, `java.rmi.server.codebase`, as in

```
java -Djava.rmi.server.codebase=http://webhost:port/directory/
     RemoteIntegralServer
```

The `java.rmi.server.codebase` property tells the server that any stub files required by the client or the RMI registry should be loaded from the HTTP server at `http://webhost:port/directory/`>. The port parameter is required only if the HTTP server is not running on standard port 80.

Compiling and Running the System for an Enterprise RMI Configuration

1. **Compile the client and server.** At the prompt, enter these commands:

   ```
   Prompt> javac RemoteIntegralClient2.java
   Prompt> javac RemoteIntegralServer.java
   ```

2. **Generate the client Stub and server Skeleton.** At the prompt, enter this command.

   ```
   Prompt> rmic -v1.2 RemoteIntegralImpl
   ```

3. **Place the appropriate files on the correct machines.** Table 17.1 summarizes the locations to place the class files. The client requires `RemoteIntegralClient2` and all other instantiated or referenced classes, including `RemoteIntegral`, `Sin`, `Cos`, and `Quadratic`. As the later three classes inherit from `Evaluatable`, that class is also needed on the client machine; `Evaluatable` is not downloaded from the server. Also, the client needs the policy file, `rmipolicy.client`.

   The server that instantiates and registers the remote object requires `RemoteIntegralServer`, and when it creates the remote object, the server requires `RemoteIntegralImpl`, `RemoteIntegral` (inherited class), and `Evaluatable`; these last three classes are not downloaded from the codebase directory (HTTP server). The class `Integral` is required on the server, since the static methods, `Integer.sum` and `Integer.evaluate`, are called in `RemoteIntegralImpl`. The server also requires `RemoteIntegralImpl_Stub` when registering the remote object. Finally, `Sin`, `Cos`, and `Quadratic` are required on the server; these classes override the `evaulate` method defined in the interface `Evaluatable` and are later required through dynamic binding in `Integral` when `evaluate` is called. These class definitions are not sent from the client.

   The HTTP server requires the stub file, `RemoteIntegralImpl_Stub`, for downloading to the client. In addition, you must place any `RemoteIntegralImpl` dependencies, `RemoteIntegral` and `Evaluatable`, on the HTTP server for use when the remote object is registered; the `rmiregistry` does not receive these files from the server Table 17.1 instantiating the remote object.

Table 17.1. Required location for class files

Client	*Server*	*HTTP Server*
`RemoteIntegralClient2`	`RemoteIntegralServer`	`RemoteIntegralImpl_Stub`
`RemoteIntegral`	`RemoteIntegralImpl`	`RemoteIntegral`
`Evaluatable`	`RemoteIntegralImpl_Stub`	`Evaluatable`
`Sin`	`RemoteIntegral`	
`Cos`	`Integral`	
`Quadratic`	`Evaluatable`	
	`Sin`	
	`Cos`	
	`Quadratic`	

4. **Start the HTTP server.** Place `RemoteIntegral_Stub.class`, `RemoteIntegeral.class`, and `Evaluatable.class` on an HTTP server and verify that you can access the files through a browser.

5. **Start the RMI registry.** Start the RMI registry:

```
Server> /somedirectory/rmiregistry
```

When you start the `rmiregistry`, make sure that none of the class files are in the directory in which you started the registry or available through the classpath.

Core Warning

The client and server may not be able to load the stub files from the correct location if the RMI registry is able to locate the files through the classpath. Always start the `rmiregistry`, without a set classpath, in a different directory than that of the server.

6. **Start the server.**

At the prompt, enter this command:

```
Server> java -Djava.rmi.server.codebase=http://webhost/rmi/
             RemoteIntegralServer
```

assuming that the stub files are placed on the HTTP server, `webhost`, and in the `rmi` subdirectory. Note that the server must be started on the same host as the `rmiregistry`, but not from within the same directory. If an exception is thrown when starting the server, correct the source of the problem, and restart the RMI registry before attempting to restart the server.

Core Note

The `rmiregistry` and server need to run on the same host or an `AccessException` is received by the server.

7. **Start the client.** At the prompt, enter the following command (where *`rmihost`* is the host in which you started the `rmiregistry` and server) to obtain the output listed below:

```
Client> java -Djava.security.policy=rmiclient.policy
             RemoteIntegralClient2 rmihost

Approximated with 10 steps:
  Integral from 0 to pi of sin(x)=2.0082484079079745
  Integral from pi/2 to pi of cos(x)=-1.0010288241427086
  Integral from 0 to 5 of x^2=41.5625
Approximated with 100 steps:
  Integral from 0 to pi of sin(x)=2.0000822490709877
  Integral from pi/2 to pi of cos(x)=-1.000010280911902
  Integral from 0 to 5 of x^2=41.665624999999906
Approximated with 1000 steps:
  Integral from 0 to pi of sin(x)=2.0000008224672983
  Integral from pi/2 to pi of cos(x)=-1.000000102808351
  Integral from 0 to 5 of x^2=41.666656249998724
Approximated with 10000 steps:
  Integral from 0 to pi of sin(x)=2.00000000822436
  Integral from pi/2 to pi of cos(x)=-1.0000000010278831
  Integral from 0 to 5 of x^2=41.666666562504055
'Correct' answer using Math library:
  Integral from 0 to pi of sin(x)=2.0
  Integral from pi/2 to pi of cos(x)=-0.9999999999999999
  Integral from 0 to 5 of x^2=41.666666666666664
```

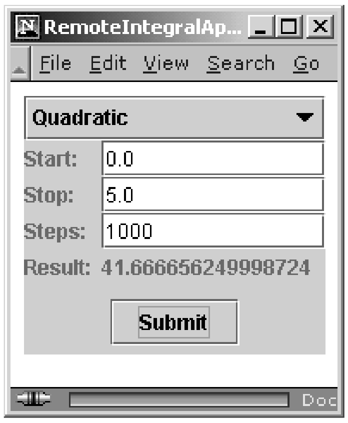
RMI Applet Example

Compared to writing an application communicating with a remote object through RMI, writing an applet that communicates through RMI is greatly simplified because the applet already invokes a security manager for loading remote files; an applet using RMI does not require a `RMISecurityManager`. In contrast though, an applet cannot open network connections other than to the server from which the applet was loaded. Therefore, the RMI registry, the server registering the remote object, and the HTTP server from which the applet and stub files are loaded *must* be the same host.

When posting the applet on an HTTP server, be sure to place all the client files in the same directory as the applet class file. Or, alternatively, you can place a single JAR file in the applet directory and specify the JAR file through the `ARCHIVE` attribute in the `APPLET` tag. See Section 7.10 (Packages, Classpath, and JAR Archives) for details on creating JAR files.

Listing 17.36 presents an applet client that communicates to a remote object through RMI. Since the RMI registry and server are located on the same host in which the applet was loaded, the host for the RMI URL is determined by a call to `getCodeBase().getHost()`. The results are shown in Figure 17-4 in Netscape 6.

Figure 17-4. Applet that communicates to a remote object through RMI in Netscape 6.



Start: 0.0
Stop: 5.0
Steps: 1000
Result: 41.666656249998724

Listing 17.36 `RemoteIntegralApplet.java`

```
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;
import java.net.*;
import java.io.*;
import javax.swing.*;

/** This class is an applet version of RemoteIntegralClient
 *  that connects to a remote machine for performing
 *  numerical integration in a sine, cosine, or quadratic
 *  equation. As an Applet imposes its own security
 *  manager, a RMISecurityManager is not needed to load
 *  the stub classes.
 */

public class RemoteIntegralApplet extends JApplet
                                  implements ActionListener {
  private Evaluatable[] shapes;
  private RemoteIntegral remoteIntegral;
  private JLabel result;
  private JTextField startInput, stopInput, stepInput;
  private JComboBox combo;
  public void init() {
    String host = getCodeBase().getHost();
    try {
      remoteIntegral =
        (RemoteIntegral)Naming.lookup("rmi://" + host +
                                      "/RemoteIntegral");

    } catch(RemoteException re) {
      reportError("RemoteException: " + re);
    } catch(NotBoundException nbe) {
      reportError("NotBoundException: " + nbe);
    } catch(MalformedURLException mfe) {
      reportError("MalformedURLException: " + mfe);
    }

    Container context = getContentPane();
    // Set up combo box.
    shapes = new Evaluatable[]{ new Sin(),
                                new Cos(),
                                new Quadratic() };
    combo = new JComboBox(shapes);
    context.add(combo, BorderLayout.NORTH);

    // Input area.
    startInput = new JTextField();
    stopInput = new JTextField();
    stepInput = new JTextField();
    result = new JLabel();
    JPanel labelPanel = new JPanel(new GridLayout(4,1));
    labelPanel.add(new JLabel("Start:"));
    labelPanel.add(new JLabel("Stop:"));
    labelPanel.add(new JLabel("Steps:"));
    labelPanel.add(new JLabel("Result:  "));
    context.add(labelPanel, BorderLayout.WEST);
    JPanel inputPanel = new JPanel(new GridLayout(4,1));
    inputPanel.add(startInput);
    inputPanel.add(stopInput);
    inputPanel.add(stepInput);
    inputPanel.add(result);
    context.add(inputPanel, BorderLayout.CENTER);
    // Set up button.
    JPanel buttonPanel = new JPanel(new FlowLayout());
    JButton submit = new JButton("Submit");
    submit.addActionListener(this);
    buttonPanel.add(submit);
    context.add(buttonPanel, BorderLayout.SOUTH);
  }

  public void actionPerformed(ActionEvent event) {
    try {
      int steps = Integer.parseInt(stepInput.getText());
      double start = Double.parseDouble(startInput.getText());
      double stop = Double.parseDouble(stopInput.getText());
      showStatus("Calculating ...");
      Evaluatable shape = (Evaluatable)combo.getSelectedItem();
      double area = remoteIntegral.integrate(start, stop,
                                             steps, shape);

      result.setText(Double.toString(area));
      showStatus("");
    } catch(NumberFormatException nfe) {
      reportError("Bad input: " + nfe);
    } catch(RemoteException re) {
      reportError("RemoteException: " + re);
    }
  }

  private void reportError(String message) {
    System.out.println(message);
    showStatus(message);
  }
}
```

Aside from Netscape 6, earlier versions of Netscape and all versions of Internet Explorer do not support Java 2 and the RMI 1.2 stub protocol without having the Java Plug-In installed (see Section 9.9, "The Java Plug-In"). Even worse, Internet Explorer does not support the RMI 1.1 stub protocol without having the RMI add-on installed. The RMI add-on is available at ftp://ftp/microsoft.com/developr/msdn/unsup-ed/. The RMI 1.1 stub protocol is supported in Netscape 4.06 and later.

17.10 Summary

Java sockets let you create network clients or servers that can communicate with general-purpose network programs written in any language. The process of building a client is straightforward: open a socket, create input and output streams from the socket, use these streams for communication, then close the socket. A server is similar but waits for an incoming connection before the socket can be created. Both types of systems frequently need to parse input they receive, and a `StringTokenizer` is a convenient tool for this. RMI is a powerful and convenient alternative when distributing processing among Java-only systems.

In the next part of the book we focus on server-side programming. First, we cover HTML forms as front ends to servlets or other server-side programs such as CGI scripts. These forms provide simple and reliable user interface controls to collect data from the user and transmit it to the servlet. Following that, we cover Java servlets, which run on a Web server, acting as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server. Next, we cover JavaServer Pages (JSP) technology to enable you to mix regular, static HTML with dynamically generated content from servlets. In addition, we show you how to write applet front ends to send data to HTTP servers through firewalls by means of HTTP tunneling. Also, we present the JDBC API, which allows to you send SQL queries to databases; and finally, we examine the Java API for XML parsing to process XML files.

CONTENTS