**3.8** Modify the date server shown in Figure 3.26 so that it delivers random fortunes rather than the current date. Allow the fortunes to contain multiple lines. The date client shown in Figure 3.27 can be used to read the multi-line fortunes returned by the fortune server.

**3.9** An **echo server** is a server that echoes back whatever it receives from a client. For example, if a client sends the server the string *Hello there!* the server will respond with the exact data it received from the client—that is, *Hello there!*

Write an echo server using the Java networking API described in Section 3.6.1. This server will wait for a client connection using the `accept()` method. When a client connection is received, the server will loop, performing the following steps:

- Read data from the socket into a buffer.
- Write the contents of the buffer back to the client.

The server will break out of the loop only when it has determined that the client has closed the connection.

The date server is shown in Figure 3.26 uses the class `java.io.BufferedReader`. `BufferedReader` extends the class `java.io.Reader`, which is used for reading character streams. However, the echo server cannot guarantee that it will read characters from clients; it may receive binary data as well. The class `java.io.InputStream` deals with data at the byte level rather than the character level. Thus, this echo server must use an object that extends `java.io.InputStream`. The `read()` method in the `java.io.InputStream` class returns $-1$ when the client has closed its end of the socket connection.

**3.10** Write an RMI application in which the server delivers random one-line fortunes. The interface for the remote object appears as

```
import java.rmi.*;

public interface RemoteFortune extends Remote
{
   public abstract String getFortune()
      throws RemoteException;
}
```

A client invoking the `getFortune()` method will receive a random one-line fortune from the remote object.

## Project—Creating a Shell Interface

This project consists of modifying a Java program so that it serves as a shell interface that accepts user commands and then executes each command in a separate process external to the Java virtual machine. A shell interface provides the user with a prompt, after which the user enters the next command. The example below illustrates the prompt `jsh>` and the user's next command: `cat`

```
import java.io.*;

public class SimpleShell
{
 public static void main(String[] args) throws java.io.IOException
  String commandLine;
  BufferedReader console = new BufferedReader
     (new InputStreamReader(System.in));

  // we break out with <control><C>
  while (true) {
   // read what the user entered
   System.out.print("jsh>");
   commandLine = console.readLine();

   // if the user entered a return, just loop again
   if (commandLine.equals(""))
     continue;

   /** The steps are:
   (1) parse the input to obtain the command and any parameters
   (2) create a ProcessBuilder object
   (3) start the process
   (4) obtain the output stream
   (5) output the contents returned by the command */
  }
 }
}
```

**Figure 3.35** Outline of simple shell.

`Prog.java`. This command displays the file `Prog.java` on the terminal using the UNIX `cat` command.

```
jsh> cat Prog.java
```

Perhaps the easiest technique for implementing a shell interface is to have the program first read what the user enters on the command line (here, `cat Prog.java`) and then create a separate external process that performs the command. We create the separate process using the `ProcessBuilder()` object, as illustrated in Figure 3.13. In our example, this separate process is external to the JVM and begins execution when its `run()` method is invoked.

### Simple Shell

A Java program that provides the basic operations of a command-line shell is supplied in Figure 3.35. The `main()` method presents the prompt `jsh>` (for java shell) and waits to read input from the user. The program is terminated when the user enters `<Control><C>`.

This project is organized into three parts: (1) creating the external process and executing the command in that process, (2) modifying the shell to allow changing directories, and (3) adding a history feature.

### Creating an External Process

The first part of this project is to modify the `main()` method in Figure 3.35 so that an external process is created and executes the command specified by the user. Initially, the command must be parsed into separate parameters and passed to the constructor for the `ProcessBuilder` object. For example, if the user enters the command

```
jsh> cat Prog.java
```

the parameters are (1) `cat` and (2) `Prog.java`, and these parameters must be passed to the `ProcessBuilder` constructor. Perhaps the easiest strategy for doing this is to use the constructor with the following signature:

```
public ProcessBuilder (List<String> command)
```

A `java.util.ArrayList`—which implements the `java.util.List` interface—can be used in this instance, where the first element of the list is `cat` and the second element is `Prog.java`. This is an especially useful strategy because the number of arguments passed to UNIX commands may vary (the `cat` command accepts one argument; the `cp` command accepts two, and so forth).

If the user enters an invalid command, the `start()` method in the `ProcessBuilder` class throws an `java.io.IOException`. If this occurs, your program should output an appropriate error message and resume waiting for further command from the user.

### Changing Directories

The next task is to modify the program in Figure 3.35 so that it changes directories. In UNIX systems, we encounter the concept of the *current working directory*, which is simply the directory you are currently in. The `cd` command allows a user to change current directories. Your shell interface must support this command. For example, if the current directory is `/usr/tom` and the user enters `cd music`, the current directory becomes `/usr/tom/music`. Subsequent commands relate to this current directory. For example, entering `ls` will output all the files in `/usr/tom/music`.

The `ProcessBuilder` class provides the following method for changing the working directory:

```
public ProcessBuilder directory(File directory)
```

When the `start()` method of a subsequent process is invoked, the new process will use this as the current working directory. For example, if one process with a current working directory of `/usr/tom` invokes the command `cd music`, subsequent processes must set their working directories to `/usr/tom/music` before beginning execution. It is important to note that your program must

first make sure the new path being specified is a valid directory. If not, your program should output an appropriate error message.

If the user enters the command `cd`, change the current working directory to the user's home directory. The home directory for the current user can be obtained by invoking the static `getProperty()` method in the `System` class as follows:

```
System.getProperty("user.dir");
```

### Adding a History Feature

Many UNIX shells provide a *history* feature that allows users to see the history of commands they have entered and to rerun a command from that history. The history includes all commands that have been entered by the user since the shell was invoked. For example, if the user entered the `history` command and saw as output:

```
0 pwd
1 ls -l
2 cat Prog.java
```

the history would list `pwd` as the first command entered, `ls -l` as the second command, and so on.

Modify your shell program so that commands are entered into a history. (Hint: The `java.util.ArrayList` provides a useful data structure for storing these commands.)

Your program must allow users to rerun commands from their history by supporting the following two techniques:

1. When the user enters `!!`, run the previous command in the history. If there is no previous command, output an appropriate error message.
2. When the user enters `!<integer value` *i*`>`, run the *i*th command in the history. For example, entering `!4` would re-run the 4th command in the command history. Make sure you perform proper error checking to ensure that the integer value is a valid number in the command history.

## Bibliographical Notes

Interprocess communication in the RC 4000 system was discussed by Brinch-Hansen [1970]. Schlichting and Schneider [1982] discussed asynchronous message-passing primitives. The IPC facility implemented at the user level was described by Bershad et al. [1990].

Details of interprocess communication in UNIX systems were presented by Gray [1997]. Barrera [1991] and Vahalia [1996] described interprocess communication in the Mach system. Solomon and Russinovich [2000] and Stevens [1999] outlined interprocess communication in Windows 2000 and UNIX respectively.