CHAPTER 3

# Processes

This covers material related to interprocess communication using Java including shared memory and message passing. It also presents material on sockets, as well as Java's remote method invocation RMI.

## 3.1 Process Creation in Java

When a Java program begins execution, an instance of the Java virtual machine is created. On most systems, the JVM appears as an ordinary application running as a separate process on the host operating system. Each instance of the JVM provides support for multiple threads of control; but Java does not support a process model, which would allow the JVM to create several processes within the same virtual machine. Although there is considerable ongoing research in this area, the primary reason why Java currently does not support a process model is that it is difficult to isolate one process's memory from that of another within the same virtual machine.

It is possible to create a process external to the JVM, however, by using the `ProcessBuilder` class, which allows a Java program to specify a process that is native to the operating system (such as `/usr/bin/ls` or `C:\\WINDOWS\\system32\\mspaint.exe`). This is illustrated in Figure 3.1. Running this program involves passing the name of the program that is to run as an external process on the command line.

We create the new process by invoking the `start()` method of the `ProcessBuilder` class, which returns an instance of a `Process` object. This process will run external to the virtual machine and cannot affect the virtual machine—and vice versa. Communication between the virtual machine and the external process occurs through the `InputStream` and `OutputStream` of the external process.

## 3.2 Interprocess Communication via Shared Memory

We now illustrate a solution to the producer–consumer problem using shared memory. Such solutions may implement the `Buffer` interface shown in Figure

```
import java.io.*;

public class OSProcess
{
 public static void main(String[] args) throws IOException {
  if (args.length != 1) {
   System.err.println("Usage: java OSProcess <command>");
   System.exit(0);
  }

  // args[0] is the command that is run in a separate process
  ProcessBuilder pb = new ProcessBuilder(args[0]);
  Process process = pb.start();

  // obtain the input stream
  InputStream is = process.getInputStream();
  InputStreamReader isr = new InputStreamReader(is);
  BufferedReader br = new BufferedReader(isr);

  // read the output of the process
  String line;
  while ( (line = br.readLine()) != null)
    System.out.println(line);

  br.close();
 }
}
```

**Figure 3.1** Creating an external process using the Java API.

3.2. The producer process invokes the `insert()` method (Figure 3.3) when it wishes to enter an item in the buffer, and the consumer calls the `remove()` method (Figure 3.4) when it wants to consume an item from the buffer.

Although Java does not provide support for shared memory, we can design a solution to the producer–consumer problem in Java that emulates shared memory by allowing the producer and consumer processes to share an instance of the `BoundedBuffer` class (Figure 3.5), which implements the `Buffer` interface. Such sharing involves passing a reference to an instance

```
public interface Buffer <E>
{
  // Producers call this method
  public void insert(E item);

  // Consumers call this method
  public E remove();
}
```

**Figure 3.2** Interface for buffer implementations.

```
// Producers call this method
public void insert(E item) {
   while (count == BUFFER_SIZE)
        ; // do nothing -- no free space

   // add an item to the buffer
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   ++count;
}
```

**Figure 3.3** The `insert()` method.

of the `BoundedBuffer` class to the producer and consumer processes. This is illustrated in Figure 3.6.

The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The variable `count` is the number of items currently in the buffer. The buffer is empty when `count == 0` and is full when `count == BUFFER_SIZE`. Note that both the producer and the consumer will block in the `while` loop if the buffer is not usable to them. In Chapter 5, we discuss how synchronization among cooperating processes can be implemented effectively in a shared-memory environment.

## 3.3 Message-Passing Systems

Now, we'll examine a solution to the producer–consumer problem that uses message passing. Our solution will implement the `Channel` interface shown in Figure 3.7. The producer and consumer will communicate indirectly using the shared mailbox illustrated in Figure 3.8.

```
// Consumers call this method
public E remove() {
   E item;

   while (count == 0)
        ; // do nothing -- nothing to consume

   // remove an item from the buffer
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   --count;

   return item;
}
```

**Figure 3.4** The `remove()` method.

```
import java.util.*;

public class BoundedBuffer<E> implements Buffer<E>
{
   private static final int BUFFER_SIZE = 5;
   private int count; // number of items in the buffer
   private int in; // points to the next free position
   private int out; // points to the next full position
   private E[] buffer;

   public BoundedBuffer() {
      // buffer is initially empty
      count = 0;
      in = 0;
      out = 0;

      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   // producers calls this method
   public void insert(E item) {
      // Figure Figure 3.3
   }

   // consumers calls this method
   public E remove() {
      // Figure Figure 3.4
   }
}
```

**Figure 3.5** Shared-memory solution to the producer–consumer problem.

The buffer is implemented using the `java.util.Vector` class, meaning that it will be a buffer of unbounded capacity. Also note that both the `send()` and `receive()` methods are nonblocking.

When the producer generates an item, it places that item in the mailbox via the `send()` method. The code for the producer is shown in Figure 3.9.

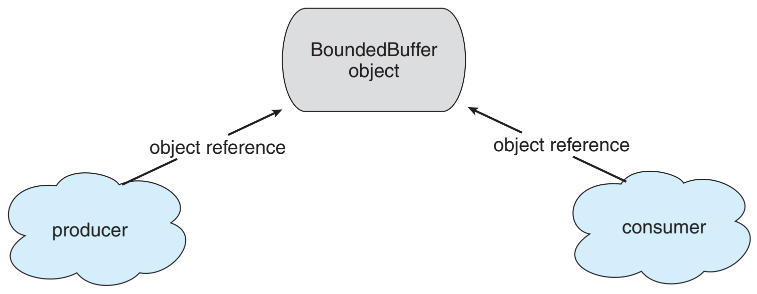

**Figure 3.6** Simulating shared memory in Java.

```
public interface Channel<E>
{
   // Send a message to the channel
   public void send(E item);

   // Receive a message from the channel
   public E receive();
}
```

**Figure 3.7** Interface for message passing.

The consumer obtains an item from the mailbox using the `receive()` method. Because `receive()` is nonblocking, the consumer must evaluate the value of the `Object` returned from `receive()`. If it is `null`, the mailbox is empty. The code for the consumer is shown in Figure 3.10.

```
import java.util.Vector;

public class MessageQueue<E> implements Channel<E>
{
    private Vector<E> queue;

    public MessageQueue() {
       queue = new Vector<E>();
    }

    // This implements a nonblocking send
    public void send(E item) {
       queue.addElement(item);
    }

    // This implements a nonblocking receive
    public E receive() {
       if (queue.size() == 0)
          return null;
       else
          return queue.remove(0);
    }
}
```

**Figure 3.8** Mailbox for message passing.

```
Channel<Date> mailBox;

while (true) {
    Date message = new Date();
    mailBox.send(message); }
```

**Figure 3.9** The producer process.

Chapter 4 shows how to implement the producer and consumer as separate threads of control and how to allow the mailbox to be shared between the threads.

## 3.4 Sockets

Sockets is another form of communication in a Client−Server Systems environment. To explore socket programming further, we turn next to an illustration using Java. Java provides an easy interface for socket programming and has a rich library for additional networking utilities.

Java provides three different types of sockets. **Connection-oriented (TCP) sockets** are implemented with the `Socket` class. **Connectionless (UDP) sockets** use the `DatagramSocket` class. Finally, the `MulticastSocket` class is a subclass of the `DatagramSocket` class. A multicast socket allows data to be sent to multiple recipients.

Our example describes a date server that uses connection-oriented TCP sockets. The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary number greater than 1024. When a connection is received, the server returns the date and time to the client.

The date server is shown in Figure 3.11. The server creates a `ServerSocket` that specifies it will listen to port 6013. The server then begins listening to the port with the `accept()` method. The server blocks on the `accept()` method waiting for a client to request a connection. When a connection request is received, `accept()` returns a socket that the server can use to communicate with the client.

The details of how the server communicates with the socket are as follows. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the routine `print()` and `println()` methods for output. The server process sends the date to the client, calling the method `println()`. Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. We implement such a client in the Java program shown in Figure 3.12. The client creates a `Socket` and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream

```
Channel<Date> mailBox;

while (true) {
   Date message = mailBox.receive();
   if (message != null)
      // consume the message
}
```

**Figure 3.10** The consumer process.

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

public class DateServer
{
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      // now listen for connections
      while (true) {
        Socket client = sock.accept();

        PrintWriter pout = new
         PrintWriter(client.getOutputStream(), true);

        // write the Date to the socket
        pout.println(new java.util.Date().toString());

        // close the socket and resume
        // listening for connections
        client.close();
      }
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

**Figure 3.11** Date server.

I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the **loopback**. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date server. In addition to an IP address, an actual host name, such as `www.westminstercollege.edu`, can be used as well.

Communication using sockets—although common and efficient—is generally considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data. In the next two subsections, we look at two higher-level methods of communication: remote procedure calls (RPCs) and Java's remote method invocation (RMI).

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

public class DateClient
{
  public static void main(String[] args) {
    try {
      //make connection to server socket
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      // read the date from the socket
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      // close the socket connection
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

**Figure 3.12** Date client.

## 3.5 Remote Method Invocation

Another form of communication in a Client–Server Systems environment is **Remote method invocation (RMI)**. RMI is a Java feature similar to RPCs. RMI allows a thread to invoke a method on a remote object. Objects are considered remote if they reside in a different Java virtual machine (JVM). Therefore, the remote object may be in a different JVM on the same computer or on a remote host connected by a network. This situation is illustrated in Figure 3.13.

### 3.5.1 Overview

RMI and RPCs differ in two fundamental ways. First, RPCs support procedural programming, whereby only remote procedures or functions may be called. In contrast, RMI is object-based: it supports invocation of methods on remote objects. Second, the parameters to remote procedures are ordinary data structures in RPC. With RMI, it is possible to pass primitive data types (for example, `int`, `boolean`), as well as objects, as parameters to remote methods. By allowing a Java program to invoke methods on remote objects, RMI makes it possible for users to develop Java applications that are distributed across a network.

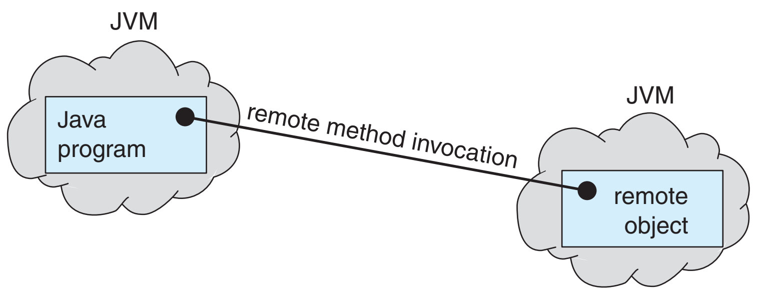

**Figure 3.13** Remote method invocation.

To make remote methods transparent to both the client and the server, RMI implements the remote object using stubs and skeletons. A **stub** is a proxy for the remote object; it resides with the client. When a client invokes a remote method, the stub for the remote object is called. This client-side stub is responsible for creating a **parcel** consisting of the name of the method to be invoked on the server and the marshalled parameters for the method. The stub then sends this parcel to the server, where the skeleton for the remote object receives it. The **skeleton** is responsible for unmarshalling the parameters and invoking the desired method on the server. The skeleton then marshals the return value (or exception, if any) into a parcel and returns this parcel to the client. The stub unmarshals the return value and passes it to the client.

Let's look more closely at how this process works. Assume that a client wishes to invoke a method on a remote object `server` with a signature `remoteMethod(Object, Object)` that returns a `boolean` value. The client executes the statement

```
boolean val = server.remoteMethod(A, B);
```

The call to `remoteMethod()` with the parameters `A` and `B` invokes the stub for the remote object. The stub marshals into a parcel the parameters `A` and `B` and the name of the method that is to be invoked on the server, then sends this parcel to the server. The skeleton on the server unmarshals the parameters and invokes the method `remoteMethod()`. The actual implementation of `remoteMethod()` resides on the server. Once the method is completed, the skeleton marshals the `boolean` value returned from `remoteMethod()` and sends this value back to the client. The stub unmarshals this return value and passes it to the client. The process is shown using the UML (Unified Modeling Language) sequence diagram in Figure 3.14.

Fortunately, the level of abstraction that RMI provides makes the stubs and skeletons transparent, allowing Java developers to write programs that invoke distributed methods just as they would invoke local methods. It is crucial, however, to understand a few rules about the behavior of parameter passing and return values:

- **Local** (or **nonremote**) objects are passed by copy using a technique known as **object serialization** which allows the state of an object to be written to a byte stream. The only requirement for object serialization is that an object must implement the `java.io.Serializable` interface. Most objects in

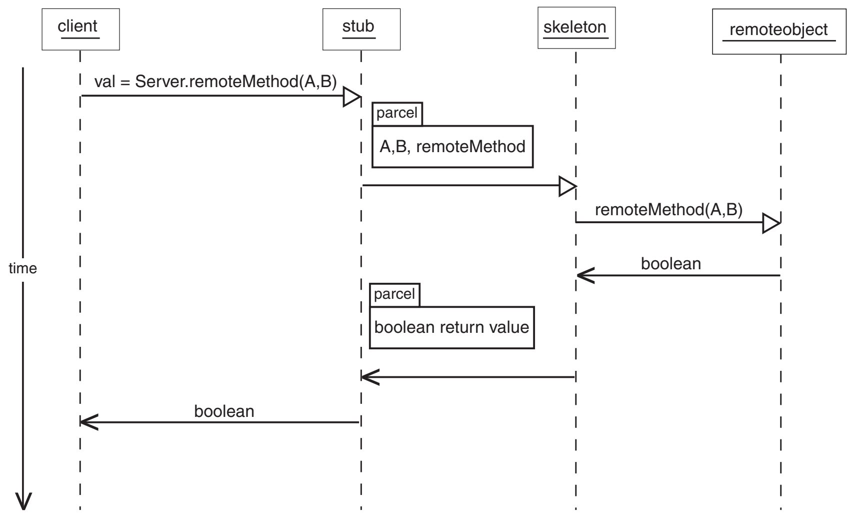

**Figure 3.14** Marshalling parameters.

the core Java API implement this interface, allowing them to be used with RMI.

- Remote objects are passed by reference. Passing an object by reference allows the receiver to alter the state of the remote object as well as invoke its remote methods.

In our example, if `A` is a local object and `B` a remote object, `A` is serialized and passed by copy, and `B` is passed by reference. This will allow the server to invoke methods on `B` remotely.

### 3.5.2 RMI Example

Next, using RMI, we'll build an application that returns the current date and time, similar to the socket-based program shown in Section 3.4.

#### 3.5.2.1 Remote Objects

Before building a distributed application, we must first define the necessary remote objects. We begin by declaring an interface that specifies the methods that can be invoked remotely. In our example of a date server, the remote method will be named `getDate()` and will return a `java.utiil.Date` containing the current date. To provide for remote objects, this interface must also extend the `java.rmi.Remote` interface, which identifies objects implementing the interface as being remote. Further, each method declared in the interface must throw the exception `java.rmi.RemoteException`. For remote objects, we provide the `RemoteDate` interface shown in Figure 3.15.

The class that defines the remote object must implement the `RemoteDate` interface (Figure 3.16). In addition to defining the `getDate()` method, the class must also extend `java.rmi.server.UnicastRemoteObject`. Extending `UnicastRemoteObject` allows the creation of a single remote object that listens for network requests using RMI's default scheme of sockets for network

```
import java.rmi.*;
import java.util.Date;

public interface RemoteDate extends Remote
{
   public Date getDate() throws RemoteException;
}
```

**Figure 3.15** The `RemoteDate` interface.

communication. This class also includes a `main()` method. The `main()` method creates an instance of the object and registers with the RMI registry running on the server via the `rebind()` method. In this case, the object instance registers itself with the name "`RMIDateObject`." Also note that we must create a default constructor for the `RemoteDateImpl` class, and it must throw a `RemoteException` if a communication or network failure prevents RMI from exporting the remote object.

#### 3.5.2.2 Access to the Remote Object

Once the remote object is registered on the server, a client (as shown in Figure 3.17) can get a proxy reference to the object from the RMI registry running on the server by using the static method `lookup()` in the `Naming` class. RMI provides a

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

public class RemoteDateImpl extends UnicastRemoteObject
      implements RemoteDate
{
   public RemoteDateImpl() throws RemoteException { }

   public Date getDate() throws RemoteException {
      return new Date();
   }

   public static void main(String[] args) {
      try {
         RemoteDate dateServer = new RemoteDateImpl();

         // Bind this object instance to the name "RMIDateObject"
         Naming.rebind("RMIDateObject", dateServer);
      }
      catch (Exception e) {
         System.err.println(e);
      }
   }
}
```

**Figure 3.16** Implementation of the `RemoteDate` interface.

```
import java.rmi.*;

public class RMIClient
{
  static final String server = "127.0.0.1";

  public static void main(String args[]) {
    try {
      String host = "rmi://" + server + "/RMIDateObject";

      RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
      System.out.println(dateServer.getDate());
    }
    catch (Exception e) {
      System.err.println(e);
    }
  }
}
```

**Figure 3.17** The RMI client.

URL-based lookup scheme using the form `rmi://server/objectName`, where `server` is the IP name (or address) of the server on which the remote object `objectName` resides and `objectName` is the name of the remote object specified by the server in the `rebind()` method (in this case, `RMIDateObject`). Once the client has the proxy reference to the remote object, it invokes the remote method `getDate()`, which returns the current date and time. Because remote methods —as well as the `Naming.lookup()` method—can throw exceptions, they must be placed in `try-catch` blocks.

#### 3.5.2.3 Running the Programs

We now demonstrate the steps necessary to run the example programs. For simplicity, we are assuming that all programs are running on the local host—that is, IP address 127.0.0.1. However, communication is still considered remote, because the client and server programs are running in their own separate Java virutal machines.

1. **Compile all source files**. Make sure that the file `RemoteDate.class` is in the same directory as `RMIClient`.

2. **Start the registry and create the remote object**. To start the registry on UNIX platforms, the user can type

   ```
   rmiregistry &
   ```

   For Windows, the user can type

   ```
   start rmiregistry
   ```

   This command starts the registry with which the remote object will register. Next, create an instance of the remote object with

   ```
   java RemoteDateImpl
   ```

This remote object will register using the name `RMIDateObject`.

3. **Reference the remote object**. Enter the statement

```
java RMIClient
```

on the command line to start the client. This program will get a proxy reference to the remote object named `RMIDateObject` and invoke the remote method `getDate()`.

## 3.6 RMI versus Sockets

Contrast the socket-based client program shown in Figure 3.12 with the RMI client shown in Figure 3.17. The socket-based client must manage the socket connection, including opening and closing the socket and establishing an `InputStream` to read from the socket. The design of the client using RMI is much simpler. All it must do is get a proxy for the remote object, which allows it to invoke the remote method `getDate()` as it would invoke an ordinary local method.

This example illustrates the appeal of techniques such as RPCs and RMI. They provide developers of distributed systems with a communication mechanism that allows them to design distributed programs without incurring the overhead of socket management.

## Programming Problems

**3.1** Section 3.4 describes port numbers below 1024 as being well known; that is, they provide standard services. Port 17 is known as the *quote-of-the-day* service. When a client connects to port 17 on a server, the server responds with a quote for that day.

Modify the date server shown in Freffig:DateServer.java so that it delivers a quote of the day rather than the current date. The quotes should be printable ASCII characters and should contain fewer than 512 characters, although multiple lines are allowed. Since port 17 is considered well known and therefore unavailable, have your server listen to port 6017. The date client shown in Figure 3.12 may be used to read the quotes returned by your server.

**3.2** A *haiku* is a three-line poem in which the first line contains five syllables, the second line contains seven syllables, and the third line contains five syllables. Write a haiku server that listens to port 5575. When a client connects to this port, the server responds with a haiku. The date client shown in Figure 3.12 may be used to read the quotes returned by your haiku server.

**3.3** Write a client–server application using Java sockets that allows a client to write a message (as a `String`) to a socket. A server will read this message, count the number of characters and digits in the message, and send these two counts back to the client. The server will listen to port

6100. The client can obtain the String message that it is to pass to the server either from the command line or by using a prompt to the user.

One strategy for sending the two counts back to the client is for the server to construct an object containing

a. The message it receives from the client
b. A count of the number of characters in the message
c. A count of the number of digits in the message

Such an object can be modeled using the following interface:

```
public interface Message
{
   // set the counts for characters and digits
   public void setCounts();

   // return the number of characters
   public int getCharacterCount();

   // return the number of digits
   public int getDigitCount();
}
```

The server will read the String from the socket, construct a new object that implements the Message interface, count the number of characters and digits in the String, and write the contents of the Message object to the socket. The client will send a String to the server and will wait for the server to respond with a Message containing the count of the number of characters and digits in the message. Communication over the socket connection will require obtaining the InputStream and OutputStream for the socket.

Objects that are written to or read from an OutputStream or InputStream must be serialized and therefore must implement the java.io.Serializable interface. This interface is known as a **marker** interface, meaning that it actually has no methods that must be implemented; basically, any objects that implement this interface can be used with either an InputStream or an OutputStream. For this assignment, you will design an object named MessageImpl that implements both java.io.Serializable and the Message interface shown above.

Serializing an object requires obtaining a java.io.ObjectOutputStream and then writing the object using the writeObject() method in the ObjectOutputStream class. Thus, the server's activity will be organized roughly as follows:

a. Reading the string from the socket
b. Constructing a new MessageImpl object and counting the number of characters and digits in the message

c. Obtaining the `ObjectOutputStream` for the socket and writing the `MessageImpl` object to this output stream

Reading a serialized object requires obtaining a `java.io.ObjectInputStream` and then reading the serialized object using the `readObject()` method in the `java.io.ObjectInputStream` class. Therefore, the client's activity will be arranged approximately as follows:

a. Writing the string to the socket

b. Obtaining the `ObjectInputStream` from the socket and reading the `MessageImpl`

Consult the Java API for further details.

**3.4** Write an RMI application that allows a client to open and read a file residing on a remote server. The interface for accessing the remote file object appears as

```
import java.rmi.*;

public interface RemoteFileObject extends Remote
{
  public abstract void open(String fileName)
    throws RemoteException;

  public abstract String readLine()
    throws RemoteException;

  public abstract void close()
    throws RemoteException;
}
```

That is, the client will open the remote file using the `open()` method, where the name of the file being opened is provided as a parameter. The file will be accessed via the `readLine()` method. This method is implemented similarly to the `readLine()` method in the `java.io.BufferedReader` class in the Java API. That is, it will read and return a line of text that is terminated by a line feed (\n), a carriage return (\r), or a carriage return followed immediately by a line feed. Once the end of the file has been reached, `readLine()` will return `null`. Once the file has been read, the client will close the file using the `close()` method. For simplicity, we assume the file being read is a character (text) stream. The client program need only display the file to the console (`System.out`).

One issue to be addressed concerns handling exceptions. The server will have to implement the methods outlined in the interface `RemoteFileObject` using standard I/O methods provided in the Java API, most of which throw a `java.io.IOException`. However, the methods to be implemented in the `RemoteFileObject` interface are declared to throw a `RemoteException`. Perhaps the easiest way of handling this situation in the server is to place the appropriate calls to standard I/O methods using `try-catch` for `java.io.IOException`. If such an exception occurs, catch it, and then re-throw it as a `RemoteException`. The code might look like this:

```
try {
   . . .
}
catch (java.io.IOException ioe) {
   throw new RemoteException("IO Exception",ioe);
}
```

You can handle a `java.io.FileNotFoundException` similarly.

## Programming Projects

### Creating a Shell Interface Using Java

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.

#### Overview

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 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 3.1. In our example, this separate process is external to the JVM and begins execution when its `run()` method is invoked.

A Java program that provides the basic operations of a command-line shell is supplied in Figure 3.18. 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.

```
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 3.18** Outline of simple shell.

### Part 1: Creating an External Process

The first part of this project is to modify the `main()` method in Figure 3.18 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 commands from the user.

## Part 2: Changing Directories

The next task is to modify the program in Figure 3.18 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");
```

## Part 3: 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 three techniques:

1. When the user enters the command `history`, you will print out the contents of the history of commands that have been entered into the shell, along with the command numbers.
2. When the user enters `!!`, run the previous command in the history. If there is no previous command, output an appropriate error message.
3. When the user enters `!<integer value` *i*`>`, run the *i*th command in the history. For example, entering `!4` would run the fourth 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.