Chapter 11

Files and Networking

Computer programs are only useful if they interact with the rest of the world in some way. This interaction is referred to as ***input/output***, or ***I/O***. Up until now, this book has concentrated on just one type of interaction: interaction with the user, through either a graphical user interface or a command-line interface. But the user is only one possible source of information and only one possible destination for information. We have already encountered one other type of input/output, since *TextIO* can read data from files and write data to files. However, Java has an input/output framework that provides much more power and flexibility than does *TextIO*, and that covers other kinds of I/O in addition to files. Most importantly, it supports communication over network connections. In Java, input/output involving files and networks is based on ***streams***, which are objects that support I/O commands that are similar to those that you have already used. In fact, standard output (`System.out`) and standard input (`System.in`) are examples of streams.

Working with files and networks requires familiarity with exceptions, which were covered in Chapter 8. Many of the subroutines that are used can throw exceptions that require mandatory exception handling. This generally means calling the subroutine in a `try..catch` statement that can deal with the exception if one occurs.

11.1 Streams, Readers, and Writers

Without the ability to interact with the rest of the world, a program would be useless. The interaction of a program with the rest of the world is referred to as ***input/output*** or I/O. Historically, one of the hardest parts of programming language design has been coming up with good facilities for doing input and output. A computer can be connected to many different types of input and output devices. If a programming language had to deal with each type of device as a special case, the complexity would be overwhelming. One of the major achievements in the history of programming has been to come up with good abstractions for representing I/O devices. In Java, the main I/O abstractions are called ***streams***. Other I/O abstractions, such as "files" and "channels" also exist, but in this section we will look only at streams. Every stream represents either a source of input or a destination to which output can be sent.

11.1.1 Character and Byte Streams

When dealing with input/output, you have to keep in mind that there are two broad categories of data: machine-formatted data and human-readable data. Machine-formatted data is represented in binary form, the same way that data is represented inside the computer, that is, as

strings of zeros and ones. Human-readable data is in the form of characters. When you read a number such as `3.141592654`, you are reading a sequence of characters and interpreting them as a number. The same number would be represented in the computer as a bit-string that you would find unrecognizable.

To deal with the two broad categories of data representation, Java has two broad categories of streams: ***byte streams*** for machine-formatted data and ***character streams*** for human-readable data. There are many predefined classes that represent streams of each type.

An object that **outputs** data to a byte stream belongs to one of the subclasses of the abstract class *OutputStream*. Objects that **read** data from a byte stream belong to subclasses of *InputStream*. If you write numbers to an *OutputStream*, you won't be able to read the resulting data yourself. But the data can be read back into the computer with an *InputStream*. The writing and reading of the data will be very efficient, since there is no translation involved: the bits that are used to represent the data inside the computer are simply copied to and from the streams.

For reading and writing human-readable character data, the main classes are the abstract classes *Reader* and *Writer*. All character stream classes are subclasses of one of these. If a number is to be written to a *Writer* stream, the computer must translate it into a human-readable sequence of characters that represents that number. Reading a number from a *Reader* stream into a numeric variable also involves a translation, from a character sequence into the appropriate bit string. (Even if the data you are working with consists of characters in the first place, such as words from a text editor, there might still be some translation. Characters are stored in the computer as 16-bit Unicode values. For people who use Western alphabets, character data is generally stored in files in ASCII code, which uses only 8 bits per character. The *Reader* and *Writer* classes take care of this translation, and can also handle non-western alphabets in countries that use them.)

Byte streams can be useful for direct machine-to-machine communication, and they can sometimes be useful for storing data in files, especially when large amounts of data need to be stored efficiently, such as in large databases. However, binary data is *fragile* in the sense that its meaning is not self-evident. When faced with a long series of zeros and ones, you have to know what information it is meant to represent and how that information is encoded before you will be able to interpret it. Of course, the same is true to some extent for character data, which is itself coded into binary form. But the binary encoding of character data has been standardized and is well understood, and data expressed in character form can be made meaningful to human readers. The current trend seems to be towards increased use of character data, represented in a way that will make its meaning as self-evident as possible. We'll look at how this is done in Section 11.6.

I should note that the original version of Java did not have character streams, and that for ASCII-encoded character data, byte streams are largely interchangeable with character streams. In fact, the standard input and output streams, `System.in` and `System.out`, are byte streams rather than character streams. However, you should use *Readers* and *Writers* rather than *InputStreams* and *OutputStreams* when working with character data.

The standard stream classes discussed in this section are defined in the package `java.io`, along with several supporting classes. You must `import` the classes from this package if you want to use them in your program. That means either importing individual classes or putting the directive "`import java.io.*;`" at the beginning of your source file. Streams are necessary for working with files and for doing communication over a network. They can also be used for communication between two concurrently running threads, and there are stream classes for

reading and writing data stored in the computer's memory.

The beauty of the stream abstraction is that it is as easy to write data to a file or to send data over a network as it is to print information on the screen.

* * *

The basic I/O classes *Reader*, *Writer*, *InputStream*, and *OutputStream* provide only very primitive I/O operations. For example, the *InputStream* class declares the instance method

```
public int read() throws IOException
```

for reading one byte of data, as a number in the range 0 to 255, from an input stream. If the end of the input stream is encountered, the `read()` method will return the value -1 instead. If some error occurs during the input attempt, an exception of type *IOException* is thrown. Since *IOException* is an exception class that requires mandatory exception-handling, this means that you can't use the `read()` method except inside a `try` statement or in a subroutine that is itself declared with a "`throws IOException`" clause. (Mandatory exception handling was covered in Subsection 8.3.4.)

The *InputStream* class also defines methods for reading several bytes of data in one step into an array of `bytes`. However, *InputStream* provides no convenient methods for reading other types of data, such as **int** or **double**, from a stream. This is not a problem because you'll never use an object of type *InputStream* itself. Instead, you'll use subclasses of *InputStream* that add more convenient input methods to *InputStream's* rather primitive capabilities. Similarly, the *OutputStream* class defines a primitive output method for writing one byte of data to an output stream. The method is defined as:

```
public void write(int b) throws IOException
```

The parameter is of type **int** rather than **byte**, but the parameter value is type-cast to type **byte** before it is written; this effectively discards all but the eight low order bytes of `b`. Again, in practice, you will almost always use higher-level output operations defined in some subclass of *OutputStream*.

The *Reader* and *Writer* classes provide identical low-level `read` and `write` methods. As in the byte stream classes, the parameter of the `write(c)` method in *Writer* and the return value of the `read()` method in *Reader* are of type **int**, but in these character-oriented classes, the I/O operations read and write characters rather than bytes. The return value of `read()` is `-1` if the end of the input stream has been reached. Otherwise, the return value must be type-cast to type **char** to obtain the character that was read. In practice, you will ordinarily use higher level I/O operations provided by sub-classes of *Reader* and *Writer*, as discussed below.

11.1.2 PrintWriter

One of the neat things about Java's I/O package is that it lets you add capabilities to a stream by "wrapping" it in another stream object that provides those capabilities. The wrapper object is also a stream, so you can read from or write to it—but you can do so using fancier operations than those available for basic streams.

For example, *PrintWriter* is a subclass of *Writer* that provides convenient methods for outputting human-readable character representations of all of Java's basic data types. If you have an object belonging to the *Writer* class, or any of its subclasses, and you would like to use *PrintWriter* methods to output data to that *Writer*, all you have to do is wrap the *Writer* in a *PrintWriter* object. You do this by constructing a new *PrintWriter* object, using the *Writer* as input to the constructor. For example, if `charSink` is of type *Writer*, then you could say

```
PrintWriter printableCharSink = new PrintWriter(charSink);
```

When you output data to `printableCharSink`, using the high-level output methods in *PrintWriter*, that data will go to exactly the same place as data written directly to `charSink`. You've just provided a better interface to the same output stream. For example, this allows you to use *PrintWriter* methods to send data to a file or over a network connection.

For the record, if `out` is a variable of type *PrintWriter*, then the following methods are defined:

- `out.print(x)` — prints the value of `x`, represented in the form of a string of characters, to the output stream; `x` can be an expression of any type, including both primitive types and object types. An object is converted to string form using its `toString()` method. A `null` value is represented by the string "null".
- `out.println()` — outputs an end-of-line to the output stream.
- `out.println(x)` — outputs the value of `x`, followed by an end-of-line; this is equivalent to `out.print(x)` followed by `out.println()`.
- `out.printf(formatString, x1, x2, ...)` — does formated output of `x1`, `x2`, `...` to the output stream. The first parameter is a string that specifies the format of the output. There can be any number of additional parameters, of any type, but the types of the parameters must match the formatting directives in the format string. Formatted output for the standard output stream, `System.out`, was introduced in Subsection 2.4.4, and `out.printf` has the same functionality.

Note that none of these methods will ever throw an *IOException*. Instead, the *PrintWriter* class includes the method

```
public boolean checkError()
```

which will return `true` if any error has been encountered while writing to the stream. The *PrintWriter* class catches any *IOExceptions* internally, and sets the value of an internal error flag if one occurs. The `checkError()` method can be used to check the error flag. This allows you to use *PrintWriter* methods without worrying about catching exceptions. On the other hand, to write a fully robust program, you should call `checkError()` to test for possible errors whenever you use a *PrintWriter*.

11.1.3 Data Streams

When you use a *PrintWriter* to output data to a stream, the data is converted into the sequence of characters that represents the data in human-readable form. Suppose you want to output the data in byte-oriented, machine-formatted form? The `java.io` package includes a byte-stream class, *DataOutputStream* that can be used for writing data values to streams in internal, binary-number format. *DataOutputStream* bears the same relationship to *OutputStream* that *PrintWriter* bears to *Writer*. That is, whereas *OutputStream* only has methods for outputting bytes, *DataOutputStream* has methods `writeDouble(double x)` for outputting values of type **double**, `writeInt(int x)` for outputting values of type **int**, and so on. Furthermore, you can wrap any *OutputStream* in a *DataOutputStream* so that you can use the higher level output methods on it. For example, if `byteSink` is of type *OutputStream*, you could say

```
DataOutputStream dataSink = new DataOutputStream(byteSink);
```

to wrap `byteSink` in a *DataOutputStream*, `dataSink`.

For input of machine-readable data, such as that created by writing to a *DataOutputStream*, `java.io` provides the class *DataInputStream*. You can wrap any *InputStream* in a *DataInputStream* object to provide it with the ability to read data of various types from the byte-stream. The methods in the *DataInputStream* for reading binary data are called `readDouble()`, `readInt()`, and so on. Data written by a *DataOutputStream* is guaranteed to be in a format that can be read by a *DataInputStream*. This is true even if the data stream is created on one type of computer and read on another type of computer. The cross-platform compatibility of binary data is a major aspect of Java's platform independence.

In some circumstances, you might need to read character data from an *InputStream* or write character data to an *OutputStream*. This is not a problem, since characters, like all data, are represented as binary numbers. However, for character data, it is convenient to use *Reader* and *Writer* instead of *InputStream* and *OutputStream*. To make this possible, you can **wrap** a byte stream in a character stream. If `byteSource` is a variable of type *InputStream* and `byteSink` is of type *OutputStream*, then the statements

```
Reader charSource = new InputStreamReader( byteSource );
Writer charSink   = new OutputStreamWriter( byteSink );
```

create character streams that can be used to read character data from and write character data to the byte streams. In particular, the standard input stream `System.in`, which is of type *InputStream* for historical reasons, can be wrapped in a *Reader* to make it easier to read character data from standard input:

```
Reader charIn = new InputStreamReader( System.in );
```

As another application, the input and output streams that are associated with a network connection are byte streams rather than character streams, but the byte streams can be wrapped in character streams to make it easy to send and receive character data over the network. We will encounter network I/O in Section 11.4.

11.1.4 Reading Text

Still, the fact remains that much I/O is done in the form of human-readable characters. In view of this, it is surprising that Java does **not** provide a standard character input class that can read character data in a manner that is reasonably symmetrical with the character output capabilities of *PrintWriter*. There is one basic case that is easily handled by a standard class. The *BufferedReader* class has a method

```
public String readLine() throws IOException
```

that reads one line of text from its input source. If the end of the stream has been reached, the return value is `null`. When a line of text is read, the end-of-line marker is read from the input stream, but it is not part of the string that is returned. Different input streams use different characters as end-of-line markers, but the `readLine` method can deal with all the common cases. (Traditionally, Unix computers, including Linux and Mac OS X, use a line feed character, `'\n'`, to mark an end of line; classic Macintosh used a carriage return character, `'\r'`; and Windows uses the two-character sequence "`\r\n`". In general, modern computers can deal correctly with all of these possibilities.)

Line-by-line processing is very common. Any *Reader* can be wrapped in a *BufferedReader* to make it easy to read full lines of text. If `reader` is of type *Reader*, then a *BufferedReader* wrapper can be created for `reader` with

```
BufferedReader in = new BufferedReader( reader );
```

This can be combined with the *InputStreamReader* class that was mentioned above to read lines of text from an *InputStream*. For example, we can apply this to `System.in`:

```
BufferedReader in;  // BufferedReader for reading from standard input.
in = new BufferedReader( new InputStreamReader( System.in ) );
try {
   String line = in.readLine();
   while ( line != null && line.length() > 0 ) {
      processOneLineOfInput( line );
      line = in.readLine();
   }
}
catch (IOException e) {
}
```

This code segment reads and processes lines from standard input until either an empty line or an end-of-stream is encountered. (An end-of-stream is possible even for interactive input. For example, on at least some computers, typing a `Control-D` generates an end-of-stream on the standard input stream.) The `try..catch` statement is necessary because the `readLine` method can throw an exception of type *IOException*, which requires mandatory exception handling; an alternative to `try..catch` would be to declare that the method that contains the code "`throws IOException`". Also, remember that *BufferedReader*, *InputStreamReader*, and *IOException* must be imported from the package `java.io`.

* * *

Previously in this book, we have used the non-standard class *TextIO* for input both from users and from files. The advantage of *TextIO* is that it makes it fairly easy to read data values of any of the primitive types. Disadvantages include the fact that *TextIO* can only read from one file at a time, that it can't do I/O operations on network connections, and that it does not follow the same pattern as Java's built-in input/output classes.

I have written a class named *TextReader* to fix some of these disadvantages, while providing input capabilities similar to those of *TextIO*. Like *TextIO*, *TextReader* is a non-standard class, so you have to be careful to make it available to any program that uses it. The source code for the class can be found in the file *TextReader.java*

Just as for many of Java's stream classes, an object of type *TextReader* can be used as a wrapper for an existing input stream, which becomes the source of the characters that will be read by the *TextReader*. (Unlike the standard classes, however, a *TextReader* is not itself a stream and cannot be wrapped inside other stream classes.) The constructors

```
public TextReader(Reader characterSource)
```

and

```
public TextReader(InputStream byteSource)
```

create objects that can be used to read human-readable data from the given *Reader* or *InputStream* using the convenient input methods of the *TextReader* class. In *TextIO*, the input methods were static members of the class. The input methods in the *TextReader* class are instance methods. The instance methods in a *TextReader* object read from the data source that was specified in the object's constructor. This makes it possible for several *TextReader* objects to exist at the same time, reading from different streams; as a result, *TextReader* can be used to read data from more than one file at the same time.

A *TextReader* object has essentially the same set of input methods as the *TextIO* class. One big difference is how errors are handled. When a *TextReader* encounters an error in the input, it throws an exception of type *IOException*. This follows the standard pattern that is used by Java's standard input streams. *IOExceptions* require mandatory exception handling, so *TextReader* methods are generally called inside `try..catch` statements. If an *IOException* is thrown by the input stream that is wrapped inside a *TextReader*, that *IOException* is simply passed along. However, other types of errors can also occur. One such possible error is an attempt to read data from the input stream when there is no more data left in the stream. A *TextReader* throws an exception of type *TextReader.EndOfStreamException* when this happens. The exception class in this case is a nested class in the *TextReader* class; it is a subclass of *IOException*, so a `try..catch` statement that handles *IOExceptions* will also handle end-of-stream exceptions. However, having a class to represent end-of-stream errors makes it possible to detect such errors and provide special handling for them. Another type of error occurs when a *TextReader* tries to read a data value of a certain type, and the next item in the input stream is not of the correct type. In this case, the *TextReader* throws an exception of type *TextReader.BadDataException*, which is another subclass of *IOException*.

For reference, here is a list of some of the more useful instance methods in the *TextReader* class. All of these methods can throw exceptions of type *IOException*:

- `public char peek()` — looks ahead at the next character in the input stream, and returns that character. The character is not removed from the stream. If the next character is an end-of-line, the return value is '`\n`'. It is legal to call this method even if there is no more data left in the stream; in that case, the return value is the constant `TextReader.EOF`. ("EOF" stands for "End-Of-File," a term that is more commonly used than "End-Of-Stream", even though not all streams are files.)
- `public boolean eoln()` and `public boolean eof()`— convenience methods for testing whether the next thing in the file is an end-of-line or an end-of-file. Note that these methods do **not** skip whitespace. If `eof()` is false, you know that there is still at least one character to be read, but there might not be any more **non-blank** characters in the stream.
- `public void skipBlanks()` and `public void skipWhiteSpace()` — skip past whitespace characters in the input stream; `skipWhiteSpace()` skips all whitespace characters, including end-of-line while `skipBlanks()` only skips spaces and tabs.
- `public String getln()` — reads characters up to the next end-of-line (or end-of-stream), and returns those characters in a string. The end-of-line marker is read but it not part of the returned string. This will throw an exception if there are no more characters in the stream.
- `public char getAnyChar()` — reads and returns the next character from the stream. The character can be a whitespace character such as a blank or end-of-line. If this method is called after all the characters in the stream have been read, an exception is thrown.
- `public int getlnInt()`, `public double getlnDouble()`, `public char getlnChar()`, etc. — skip any whitespace characters in the stream, including end-of-lines, then read a value of the specified type, which will be the return value of the method. Any remaining characters on the line are then discarded, including the end-of-line marker. There is a method for each primitive type. An exception occurs if it's not possible to read a data value of the requested type.
- `public int getInt()`, `public double getDouble()`, `public char getChar()`, etc. —

skip any whitespace characters in the stream, including end-of-lines, then read and return a value of the specified type. Extra characters on the line are **not** discarded and are still available to be read by subsequent input methods. There is a method for each primitive type. An exception occurs if it's not possible to read a data value of the requested type.

11.1.5 The Scanner Class

Since its introduction, Java has been notable for its lack of built-in support for basic input, and for its reliance on fairly advanced techniques for the support that it does offer. (This is my opinion, at least.) The *Scanner* class was introduced in Java 5.0 to make it easier to read basic data types from a character input source. It does not (again, in my opinion) solve the problem completely, but it is a big improvement. The *Scanner* class is in the package `java.util`.

Input routines are defined as instance methods in the *Scanner* class, so to use the class, you need to create a *Scanner* object. The constructor specifies the source of the characters that the *Scanner* will read. The scanner acts as a wrapper for the input source. The source can be a *Reader*, an *InputStream*, a *String*, or a *File*. (If a *String* is used as the input source, the *Scanner* will simply read the characters in the string from beginning to end, in the same way that it would process the same sequence of characters from a stream. The *File* class will be covered in the next section.) For example, you can use a *Scanner* to read from standard input by saying:

```
Scanner standardInputScanner = new Scanner( System.in );
```

and if `charSource` is of type *Reader*, you can create a *Scanner* for reading from `charSource` with:

```
Scanner scanner = new Scanner( charSource );
```

When processing input, a scanner usually works with ***tokens***. A token is a meaningful string of characters that cannot, for the purposes at hand, be further broken down into smaller meaningful pieces. A token can, for example, be an individual word or a string of characters that represents a value of type **double**. In the case of a scanner, tokens must be separated by "delimiters." By default, the delimiters are whitespace characters such as spaces and end-of-line markers. In normal processing, whitespace characters serve simply to separate tokens and are discarded by the scanner. A scanner has instance methods for reading tokens of various types. Suppose that `scanner` is an object of type *Scanner*. Then we have:

- `scanner.next()` — reads the next token from the input source and returns it as a *String*.
- `scanner.nextInt()`, `scanner.nextDouble()`, and so on — reads the next token from the input source and tries to convert it to a value of type **int**, **double**, and so on. There are methods for reading values of any of the primitive types.
- `scanner.nextLine()` — reads an entire line from the input source, up to the next end-of-line and returns the line as a value of type *String*. The end-of-line marker is read but is not part of the return value. Note that this method is **not** based on tokens. An entire line is read and returned, including any whitespace characters in the line.

All of these methods can generate exceptions. If an attempt is made to read past the end of input, an exception of type *NoSuchElementException* is thrown. Methods such as `scanner.getInt()` will throw an exception of type *InputMismatchException* if the next token in the input does not represent a value of the requested type. The exceptions that can be generated do not require mandatory exception handling.

The *Scanner* class has very nice look-ahead capabilities. You can query a scanner to determine whether more tokens are available and whether the next token is of a given type. If `scanner` is of type *Scanner*:

- `scanner.hasNext()` — returns a **boolean** value that is true if there is at least one more token in the input source.
- `scanner.hasNextInt()`, `scanner.hasNextDouble()`, and so on — returns a **boolean** value that is true if there is at least one more token in the input source and that token represents a value of the requested type.
- `scanner.hasNextLine()` — returns a **boolean** value that is true if there is at least one more line in the input source.

Although the insistence on defining tokens only in terms of delimiters limits the usability of scanners to some extent, they are easy to use and are suitable for many applications.

11.1.6 Serialized Object I/O

The classes *PrintWriter*, *TextReader*, *Scanner*, *DataInputStream*, and *DataOutputStream* allow you to easily input and output all of Java's primitive data types. But what happens when you want to read and write **objects**? Traditionally, you would have to come up with some way of encoding your object as a sequence of data values belonging to the primitive types, which can then be output as bytes or characters. This is called ***serializing*** the object. On input, you have to read the serialized data and somehow reconstitute a copy of the original object. For complex objects, this can all be a major chore. However, you can get Java to do all the work for you by using the classes *ObjectInputStream* and *ObjectOutputStream*. These are subclasses of *InputStream* and *Outputstream* that can be used for writing and reading serialized objects.

ObjectInputStream and *ObjectOutputStream* are wrapper classes that can be wrapped around arbitrary *InputStreams* and *OutputStreams*. This makes it possible to do object input and output on any byte stream. The methods for object I/O are `readObject()`, in *ObjectInputStream*, and `writeObject(Object obj)`, in *ObjectOutputStream*. Both of these methods can throw *IOExceptions*. Note that `readObject()` returns a value of type *Object*, which generally has to be type-cast to a more useful type.

ObjectOutputStream also has methods `writeInt()`, `writeDouble()`, and so on, for outputting primitive type values to the stream, and *ObjectInputStream* has corresponding methods for reading primitive type values.

Object streams are byte streams. The objects are represented in binary, machine-readable form. This is good for efficiency, but it does suffer from the fragility that is often seen in binary data. They suffer from the additional problem that the binary format of Java objects is very specific to Java, so the data in object streams is not easily available to programs written in other programming languages. For these reasons, object streams are appropriate mostly for short-term storage of objects and for transmitting objects over a network connection from one Java program to another. For long-term storage and for communication with non-Java programs, other approaches to object serialization are usually better. (See Subsection 11.6.2 for a character-based approach.)

ObjectInputStream and *ObjectOutputStream* only work with objects that implement an interface named *Serializable*. Furthermore, all of the instance variables in the object must be serializable. However, there is little work involved in making an object serializable, since the *Serializable* interface does not declare any methods. It exists only as a marker for the compiler,

to tell it that the object is meant to be writable and readable. You only need to add the words "`implements Serializable`" to your class definitions. Many of Java's standard classes are already declared to be serializable, including all the component classes and many other classes in Swing and in the AWT. One of the programming examples in Section 11.3 uses object IO.

11.2 Files

THE DATA AND PROGRAMS in a computer's main memory survive only as long as the power is on. For more permanent storage, computers use ***files***, which are collections of data stored on a hard disk, on a USB memory stick, on a CD-ROM, or on some other type of storage device. Files are organized into ***directories*** (sometimes called ***folders***). A directory can hold other directories, as well as files. Both directories and files have names that are used to identify them.

Programs can read data from existing files. They can create new files and can write data to files. In Java, such input and output can be done using streams. Human-readable character data is read from a file using an object belonging to the class *FileReader*, which is a subclass of *Reader*. Similarly, data is written to a file in human-readable format through an object of type *FileWriter*, a subclass of *Writer*. For files that store data in machine format, the appropriate I/O classes are *FileInputStream* and *FileOutputStream*. In this section, I will only discuss character-oriented file I/O using the *FileReader* and *FileWriter* classes. However, *FileInputStream* and *FileOutputStream* are used in an exactly parallel fashion. All these classes are defined in the `java.io` package.

It's worth noting right at the start that applets which are downloaded over a network connection are not allowed to access files (unless you have made a very foolish change to your web browser's configuration). This is a security consideration. You can download and run an applet just by visiting a Web page with your browser. If downloaded applets had access to the files on your computer, it would be easy to write an applet that would destroy all the data on a computer that downloads it. To prevent such possibilities, there are a number of things that downloaded applets are not allowed to do. Accessing files is one of those forbidden things. Standalone programs written in Java, however, have the same access to your files as any other program. When you write a standalone Java application, you can use all the file operations described in this section.

11.2.1 Reading and Writing Files

The *FileReader* class has a constructor which takes the name of a file as a parameter and creates an input stream that can be used for reading from that file. This constructor will throw an exception of type *FileNotFoundException* if the file doesn't exist. It requires mandatory exception handling, so you have to call the constructor in a `try..catch` statement (or inside a routine that is declared to throw the exception). For example, suppose you have a file named "`data.txt`", and you want your program to read data from that file. You could do the following to create an input stream for the file:

```
FileReader data;   // (Declare the variable before the
                   //   try statement, or else the variable
                   //   is local to the try block and you won't
                   //   be able to use it later in the program.)

try {
   data = new FileReader("data.txt");  // create the stream
```

```
}
catch (FileNotFoundException e) {
   ... // do something to handle the error---maybe, end the program
}
```

The *FileNotFoundException* class is a subclass of *IOException*, so it would be acceptable to catch *IOExceptions* in the above `try...catch` statement. More generally, just about any error that can occur during input/output operations can be caught by a `catch` clause that handles *IOException*.

Once you have successfully created a *FileReader*, you can start reading data from it. But since *FileReaders* have only the primitive input methods inherited from the basic *Reader* class, you will probably want to wrap your *FileReader* in a *Scanner*, in a *TextReader*, or in some other wrapper class. (The *TextReader* class is not a standard part of Java; it is described in Subsection 11.1.4. *Scanner* is discussed in Subsection 11.1.5.) To create a *TextReader* for reading from a file named `data.dat`, you could say:

```
TextReader data;

try {
   data = new TextReader( new FileReader("data.dat") );
}
catch (FileNotFoundException e) {
   ... // handle the exception
}
```

Once you have a *TextReader* named `data`, you can read from it using such methods as `data.getInt()` and `data.peek()`, exactly as you would from any other *TextReader*.

Working with output files is no more difficult than this. You simply create an object belonging to the class *FileWriter*. You will probably want to wrap this output stream in an object of type *PrintWriter*. For example, suppose you want to write data to a file named "`result.dat`". Since the constructor for *FileWriter* can throw an exception of type *IOException*, you should use a `try..catch` statement:

```
PrintWriter result;

try {
   result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
   ... // handle the exception
}
```

If no file named `result.dat` exists, a new file will be created. If the file already exists, then the current contents of the file will be erased and replaced with the data that your program writes to the file. This will be done without any warning. To avoid overwriting a file that already exists, you can check whether a file of the same name already exists before trying to create the stream, as discussed later in this section. An *IOException* might occur in the *PrintWriter* constructor if, for example, you are trying to create a file on a disk that is "write-protected," meaning that it cannot be modified.

After you are finished using a file, it's a good idea to ***close*** the file, to tell the operating system that you are finished using it. You can close a file by calling the `close()` method of the associated stream. Once a file has been closed, it is no longer possible to read data from it or write data to it, unless you open it again as a new stream. (Note that for most stream

classes, the `close()` method can throw an *IOException*, which must be handled; however, both *PrintWriter* and *TextReader* override this method so that it cannot throw such exceptions.) If you forget to close a file, the file will ordinarily be closed automatically when the program terminates or when the file object is garbage collected, but in the case of an output file, some of the data that has been written to the file might be lost. This can occur because data that is written to a file can be ***buffered***; that is, the data is not sent immediately to the file but is retained in main memory (in a "buffer") until a larger chunk of data is ready to be written. This is done for efficiency. The `close()` method of an output stream will cause all the data in the buffer to be sent to the file. Every output stream also has a `flush()` method that can be called to force any data in the buffer to be written to the file without closing the file.

As a complete example, here is a program that will read numbers from a file named `data.dat`, and will then write out the same numbers in reverse order to another file named `result.dat`. It is assumed that `data.dat` contains only one number on each line. Exception-handling is used to check for problems along the way. Although the application is not a particularly useful one, this program demonstrates the basics of working with files. (By the way, at the end of this program, you'll find our first useful example of a `finally` clause in a `try` statement. When the computer executes a `try` statement, the commands in its `finally` clause are guaranteed to be executed, no matter what.)

```
import java.io.*;
import java.util.ArrayList;

/**
 * Reads numbers from a file named data.dat and writes them to a file
 * named result.dat in reverse order.  The input file should contain
 * exactly one real number per line.
 */
public class ReverseFile {

   public static void main(String[] args) {

      TextReader data;     // Character input stream for reading data.
      PrintWriter result;  // Character output stream for writing data.

      ArrayList<Double> numbers;  // An ArrayList for holding the data.

      numbers = new ArrayList<Double>();

      try {  // Create the input stream.
         data = new TextReader(new FileReader("data.dat"));
      }
      catch (FileNotFoundException e) {
         System.out.println("Can't find file data.dat!");
         return;  // End the program by returning from main().
      }

      try {  // Create the output stream.
         result = new PrintWriter(new FileWriter("result.dat"));
      }
      catch (IOException e) {
         System.out.println("Can't open file result.dat!");
         System.out.println("Error: " + e);
         data.close();  // Close the input file.
         return;        // End the program.
      }
```

```
            try {

                // Read numbers from the input file, adding them to the ArrayList.

                while ( data.eof() == false ) {  // Read until end-of-file.
                   double inputNumber = data.getlnDouble();
                   numbers.add( inputNumber );
                }

                // Output the numbers in reverse order.

                for (int i = numbers.size()-1; i >= 0; i--)
                   result.println(numbers.get(i));

                System.out.println("Done!");

             }
             catch (IOException e) {
                   // Some problem reading the data from the input file.
                System.out.println("Input Error: " + e.getMessage());
             }
             finally {
                   // Finish by closing the files, whatever else may have happened.
                data.close();
                result.close();
             }

          }  // end of main()

       } // end of class
```

11.2.2 Files and Directories

The subject of file names is actually more complicated than I've let on so far. To fully specify a file, you have to give both the name of the file and the name of the directory where that file is located. A simple file name like "data.dat" or "result.dat" is taken to refer to a file in a directory that is called the ***current directory*** (also known as the "default directory" or "working directory"). The current directory is not a permanent thing. It can be changed by the user or by a program. Files not in the current directory must be referred to by a ***path name***, which includes both the name of the file and information about the directory where it can be found.

To complicate matters even further, there are two types of path names, ***absolute path names*** and ***relative path names***. An absolute path name uniquely identifies one file among all the files available to the computer. It contains full information about which directory the file is in and what the file's name is. A relative path name tells the computer how to locate the file starting from the current directory.

Unfortunately, the syntax for file names and path names varies somewhat from one type of computer to another. Here are some examples:

- `data.dat` — on any computer, this would be a file named "data.dat" in the current directory.
- `/home/eck/java/examples/data.dat` — This is an absolute path name in a UNIX operating system, including Linux and Mac OS X. It refers to a file named data.dat in a directory named examples, which is in turn in a directory named java, ...

- `C:\eck\java\examples\data.dat` — An absolute path name on a Windows computer.
- `Hard Drive:java:examples:data.dat` — Assuming that "Hard Drive" is the name of a disk drive, this would be an absolute path name on a computer using a classic Macintosh operating system such as Mac OS 9.
- `examples/data.dat` — a relative path name under UNIX. "examples" is the name of a directory that is contained within the current directory, and data.dat is a file in that directory. The corresponding relative path name for Windows would be `examples\data.dat`.
- `../examples/data.dat` — a relative path name in UNIX that means "go to the directory that contains the current directory, then go into a directory named examples inside that directory, and look there for a file named data.data." In general, "`..`" means "go up one directory."

It's reasonably safe to say, though, that if you stick to using simple file names only, and if the files are stored in the same directory with the program that will use them, then you will be OK. Later in this section, we'll look at a convenient way of letting the user specify a file in a GUI program, which allows you to avoid the issue of path names altogether.

It is possible for a Java program to find out the absolute path names for two important directories, the current directory and the user's home directory. The names of these directories are ***system properties***, and they can be read using the function calls:

- `System.getProperty("user.dir")` — returns the absolute path name of the current directory as a *String*.
- `System.getProperty("user.home")` — returns the absolute path name of the user's home directory as a *String*.

To avoid some of the problems caused by differences in path names between platforms, Java has the class `java.io.File`. An object belonging to this class represents a file. More precisely, an object of type *File* represents a file **name** rather than a file as such. The file to which the name refers might or might not exist. Directories are treated in the same way as files, so a *File* object can represent a directory just as easily as it can represent a file.

A *File* object has a constructor, `new File(String)`, that creates a *File* object from a path name. The name can be a simple name, a relative path, or an absolute path. For example, `new File("data.dat")` creates a *File* object that refers to a file named data.dat, in the current directory. Another constructor, `new File(File,String)`, has two parameters. The first is a *File* object that refers to the directory that contains the file. The second can be the name of the file or a relative path from the directory to the file.

File objects contain several useful instance methods. Assuming that `file` is a variable of type *File*, here are some of the methods that are available:

- `file.exists()` — This **boolean**-valued function returns `true` if the file named by the *File* object already exists. You can use this method if you want to avoid overwriting the contents of an existing file when you create a new *FileWriter*.
- `file.isDirectory()` — This **boolean**-valued function returns `true` if the *File* object refers to a directory. It returns `false` if it refers to a regular file or if no file with the given name exists.
- `file.delete()` — Deletes the file, if it exists. Returns a **boolean** value to indicate whether the file was successfully deleted.

- `file.list()` — If the `File` object refers to a directory, this function returns an array of type `String[]` containing the names of the files in that directory. Otherwise, it returns `null`.

Here, for example, is a program that will list the names of all the files in a directory specified by the user. Just for fun, I have used a *Scanner* (Subsection 11.1.5) to read the user's input:

```
import java.io.File;
import java.util.Scanner;

/**
 * This program lists the files in a directory specified by
 * the user.  The user is asked to type in a directory name.
 * If the name entered by the user is not a directory, a
 * message is printed and the program ends.
 */
public class DirectoryList {


   public static void main(String[] args) {

      String directoryName;  // Directory name entered by the user.
      File directory;        // File object referring to the directory.
      String[] files;        // Array of file names in the directory.
      Scanner scanner;       // For reading a line of input from the user.

      scanner = new Scanner(System.in);  // scanner reads from standard input.

      System.out.print("Enter a directory name: ");
      directoryName = scanner.nextLine().trim();
      directory = new File(directoryName);

      if (directory.isDirectory() == false) {
          if (directory.exists() == false)
             System.out.println("There is no such directory!");
          else
             System.out.println("That file is not a directory.");
      }
      else {
          files = directory.list();
          System.out.println("Files in directory \"" + directory + "\":");
          for (int i = 0; i < files.length; i++)
             System.out.println("   " + files[i]);
      }

   } // end main()

} // end class DirectoryList
```

All the classes that are used for reading data from files and writing data to files have constructors that take a *File* object as a parameter. For example, if `file` is a variable of type *File*, and you want to read character data from that file, you can create a *FileReader* to do so by saying `new FileReader(file)`. If you want to use a *TextReader* to read from the file, you could say:

```
TextReader data;

try {
   data = new TextReader( new FileReader(file) );
}
catch (FileNotFoundException e) {
   ... // handle the exception
}
```

11.2.3 File Dialog Boxes

In many programs, you want the user to be able to select the file that is going to be used for input or output. If your program lets the user type in the file name, you will just have to assume that the user understands how to work with files and directories. But in a graphical user interface, the user expects to be able to select files using a ***file dialog box***, which is a window that a program can open when it wants the user to select a file for input or output. Swing includes a platform-independent technique for using file dialog boxes in the form of a class called *JFileChooser*. This class is part of the package `javax.swing`. We looked at using some basic dialog boxes in Subsection 6.8.2. File dialog boxes are similar to those, but are a little more complicated to use.

A file dialog box shows the user a list of files and sub-directories in some directory, and makes it easy for the user to specify a file in that directory. The user can also navigate easily from one directory to another. The most common constructor for *JFileChooser* has no parameter and sets the starting directory in the dialog box to be the user's home directory. There are also constructors that specify the starting directory explicitly:

```
new JFileChooser( File startDirectory )

new JFileChooser( String pathToStartDirectory )
```

Constructing a *JFileChooser* object does not make the dialog box appear on the screen. You have to call a method in the object to do that. There are two different methods that can be used because there are two types of file dialog: An ***open file dialog*** allows the user to specify an existing file to be opened for reading data into the program; a ***save file dialog*** lets the user specify a file, which might or might not already exist, to be opened for writing data from the program. File dialogs of these two types are opened using the `showOpenDialog` and `showSaveDialog` methods. These methods make the dialog box appear on the screen; the methods do not end until the user selects a file or cancels the dialog.

A file dialog box always has a ***parent***, another component which is associated with the dialog box. The parent is specified as a parameter to the `showOpenDialog` or `showSaveDialog` methods. The parent is a GUI component, and can often be specified as "`this`" in practice, since file dialogs are often used in instance methods of GUI component classes. (The parameter can also be `null`, in which case an invisible component is created to be used as the parent.) Both `showOpenDialog` and `showSaveDialog` have a return value, which will be one of the constants `JFileChooser.CANCEL_OPTION`, `JFileChooser.ERROR_OPTION`, or `JFileChooser.APPROVE_OPTION`. If the return value is `JFileChooser.APPROVE_OPTION`, then the user has selected a file. If the return value is something else, then the user did not select a file. The user might have clicked a "Cancel" button, for example. You should always check the return value, to make sure that the user has, in fact, selected a file. If that is the case, then you can find out which file was selected by calling the *JFileChooser's* `getSelectedFile()` method, which returns an object of type *File* that represents the selected file.

Putting all this together, we can look at a typical subroutine that reads data from a file that is selected using a *JFileChooser*:

```
public void readFile() {
   if (fileDialog == null)   // (fileDialog is an instance variable)
      fileDialog = new JFileChooser();
   fileDialog.setDialogTitle("Select File for Reading");
   fileDialog.setSelectedFile(null);  // No file is initially selected.
   int option = fileDialog.showOpenDialog(this);
       // (Using "this" as a parameter to showOpenDialog() assumes that the
       //  readFile() method is an instance method in a GUI component class.)
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   File selectedFile = fileDialog.getSelectedFile();
   TextReader in;  // (or use some other wrapper class)
   try {
      FileReader stream = new FileReader(selectedFile); // (or a FileInputStream)
      in = new TextReader( stream );
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
          "Sorry, but an error occurred while trying to open the file:\n" + e);
      return;
   }
   try {
      .
      .  // Read and process the data from the input stream, in.
      .
     in.close();
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
          "Sorry, but an error occurred while trying to read the data:\n" + e);
   }
}
```

One fine point here is that the variable `fileDialog` is an instance variable of type *JFileChooser*. This allows the file dialog to continue to exist between calls to `readFile()`. The main effect of this is that the dialog box will keep the same selected directory from one call of `readFile()` to the next. When the dialog reappears, it will show the same directory that the user selected the previous time it appeared. This is probably what the user expects.

Note that it's common to do some configuration of a *JFileChooser* before calling `showOpenDialog` or `showSaveDialog`. For example, the instance method `setDialogTitle(String)` is used to specify a title to appear in the title bar of the window. And `setSelectedFile(File)` is used to set the file that is selected in the dialog box when it appears. This can be used to provide a default file choice for the user. In the `readFile()` method, above, `fileDialog.setSelectedFile(null)` specifies that no file is pre-selected when the dialog box appears.

Writing data to a file is similar, but it's a good idea to add a check to determine whether the output file that is selected by the user already exists. In that case, ask the user whether to replace the file. Here is a typical subroutine for writing to a user-selected file:

```
public void writeFile() {
   if (fileDialog == null)
      fileDialog = new JFileChooser();  // (fileDialog is an instance variable)
   File selectedFile = new File("(default file name)");
   fileDialog.setSelectedFile(selectedFile);  // Specify a default file name.
   fileDialog.setDialogTitle("Select File for Writing");
   int option = fileDialog.showSaveDialog(this);
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   selectedFile = fileDialog.getSelectedFile();
   if (selectedFile.exists()) {  // Ask the user whether to replace the file.
      int response = JOptionPane.showConfirmDialog( this,
            "The file \"" + selectedFile.getName()
                + "\" already exists.\nDo you want to replace it?",
            "Confirm Save",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.WARNING_MESSAGE );
      if (response != JOptionPane.YES_OPTION)
         return;  // User does not want to replace the file.
   }
   PrintWriter out;  // (or use some other wrapper class)
   try {
      FileWriter stream = new FileWriter(selectedFile); // (or FileOutputStream)
      out = new PrintWriter( stream );
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
          "Sorry, but an error occurred while trying to open the file:\n" + e);
      return;
   }
   try {
      .
      .  // Write data to the output stream, out.
      .
     out.close();
     if (out.checkError())   // (need to check for errors in PrintWriter)
        throw new IOException("Error check failed.");
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
          "Sorry, but an error occurred while trying to write the data:\n" + e);
   }
}
```

The `readFile()` and `writeFile()` routines presented here can be used, with just a few changes, when you need to read or write a file in a GUI program. We'll look at some more complete examples of using files and file dialogs in the next section.

11.3 Programming With Files

In this section, we look at several programming examples that work with files, using the techniques that were introduced in Section 11.1 and Section 11.2.

11.3.1 Copying a File

As a first example, we look at a simple command-line program that can make a copy of a file. Copying a file is a pretty common operation, and every operating system already has a command for doing it. However, it is still instructive to look at a Java program that does the same thing. Many file operations are similar to copying a file, except that the data from the input file is processed in some way before it is written to the output file. All such operations can be done by programs with the same general form.

Since the program should be able to copy any file, we can't assume that the data in the file is in human-readable form. So, we have to use *InputStream* and *OutputStream* to operate on the file rather than *Reader* and *Writer*. The program simply copies all the data from the *InputStream* to the *OutputStream*, one byte at a time. If `source` is the variable that refers to the *InputStream*, then the function `source.read()` can be used to read one byte. This function returns the value -1 when all the bytes in the input file have been read. Similarly, if `copy` refers to the *OutputStream*, then `copy.write(b)` writes one byte to the output file. So, the heart of the program is a simple `while` loop. As usual, the I/O operations can throw exceptions, so this must be done in a `try..catch` statement:

```
while(true) {
   int data = source.read();
   if (data < 0)
      break;
   copy.write(data);
}
```

The file-copy command in an operating system such as UNIX uses command line arguments to specify the names of the files. For example, the user might say "`copy original.dat backup.dat`" to copy an existing file, `original.dat`, to a file named `backup.dat`. Command-line arguments can also be used in Java programs. The command line arguments are stored in the array of strings, `args`, which is a parameter to the `main()` routine. The program can retrieve the command-line arguments from this array. (See Subsection 7.2.3.) For example, if the program is named `CopyFile` and if the user runs the program with the command "`java CopyFile work.dat oldwork.dat`", then in the program, `args[0]` will be the string `"work.dat"` and `args[1]` will be the string `"oldwork.dat"`. The value of `args.length` tells the program how many command-line arguments were specified by the user.

My `CopyFile` program gets the names of the files from the command-line arguments. It prints an error message and exits if the file names are not specified. To add a little interest, there are two ways to use the program. The command line can simply specify the two file names. In that case, if the output file already exists, the program will print an error message and end. This is to make sure that the user won't accidently overwrite an important file. However, if the command line has three arguments, then the first argument must be "`-f`" while the second and third arguments are file names. The `-f` is a ***command-line option***, which is meant to modify the behavior of the program. The program interprets the `-f` to mean that it's OK to overwrite an existing program. (The "f" stands for "force," since it forces the file to be copied in spite of what would otherwise have been considered an error.) You can see in the source code how the command line arguments are interpreted by the program:

```
import java.io.*;

/**
 *  Makes a copy of a file.  The original file and the name of the
```

```
 *  copy must be given as command-line arguments.  In addition, the
 *  first command-line argument can be "-f"; if present, the program
 *  will overwrite an existing file; if not, the program will report
 *  an error and end if the output file already exists.  The number
 *  of bytes that are copied is reported.
 */
public class CopyFile {

   public static void main(String[] args) {

      String sourceName;   // Name of the source file,
                           //    as specified on the command line.
      String copyName;     // Name of the copy,
                           //    as specified on the command line.
      InputStream source;  // Stream for reading from the source file.
      OutputStream copy;   // Stream for writing the copy.
      boolean force;  // This is set to true if the "-f" option
                      //    is specified on the command line.
      int byteCount;  // Number of bytes copied from the source file.

      /* Get file names from the command line and check for the
         presence of the -f option.  If the command line is not one
         of the two possible legal forms, print an error message and
         end this program. */

      if (args.length == 3 && args[0].equalsIgnoreCase("-f")) {
         sourceName = args[1];
         copyName = args[2];
         force = true;
      }
      else if (args.length == 2) {
         sourceName = args[0];
         copyName = args[1];
         force = false;
      }
      else {
         System.out.println(
                 "Usage:  java CopyFile <source-file> <copy-name>");
         System.out.println(
                 "    or  java CopyFile -f <source-file> <copy-name>");
         return;
      }

      /* Create the input stream.  If an error occurs, end the program. */

      try {
         source = new FileInputStream(sourceName);
      }
      catch (FileNotFoundException e) {
         System.out.println("Can't find file \"" + sourceName + "\".");
         return;
      }

      /* If the output file already exists and the -f option was not
         specified, print an error message and end the program. */

      File file = new File(copyName);
```

```
        if (file.exists() && force == false) {
            System.out.println(
                 "Output file exists.  Use the -f option to replace it.");
            return;
        }

        /* Create the output stream.  If an error occurs, end the program. */

        try {
           copy = new FileOutputStream(copyName);
        }
        catch (IOException e) {
           System.out.println("Can't open output file \"" + copyName + "\".");
           return;
        }

        /* Copy one byte at a time from the input stream to the output
           stream, ending when the read() method returns -1 (which is
           the signal that the end of the stream has been reached).  If any
           error occurs, print an error message.  Also print a message if
           the file has been copied successfully.  */

        byteCount = 0;

        try {
           while (true) {
              int data = source.read();
              if (data < 0)
                 break;
              copy.write(data);
              byteCount++;
           }
           source.close();
           copy.close();
           System.out.println("Successfully copied " + byteCount + " bytes.");
        }
        catch (Exception e) {
           System.out.println("Error occurred while copying.  "
                                    + byteCount + " bytes copied.");
           System.out.println("Error: " + e);
        }

     }  // end main()

  } // end class CopyFile
```

11.3.2 Persistent Data

Once a program ends, any data that was stored in variables and objects in the program is gone. In many cases, it would be useful to have some of that data stick around so that it will be available when the program is run again. The problem is, how to make the data ***persistent*** between runs of the program? The answer, of course, is to store the data in a file (or, for some applications, in a database—but the data in a database is itself stored in files).

Consider a "phone book" program that allows the user to keep track of a list of names and associated phone numbers. The program would make no sense at all if the user had to create

the whole list from scratch each time the program is run. It would make more sense to think of the phone book as a persistent collection of data, and to think of the program as an interface to that collection of data. The program would allow the user to look up names in the phone book and to add new entries. Any changes that are made should be preserved after the program ends.

The sample program *PhoneDirectoryFileDemo.java* is a very simple implementation of this idea. It is meant only as an example of file use; the phone book that it implements is a "toy" version that is not meant to be taken seriously. This program stores the phone book data in a file named "`.phone_book_demo`" in the user's home directory. To find the user's home directory, it uses the `System.getProperty()` method that was mentioned in Subsection 11.2.2. When the program starts, it checks whether the file already exists. If it does, it should contain the user's phone book, which was saved in a previous run of the program, so the data from the file is read and entered into a *TreeMap* named `phoneBook` that represents the phone book while the program is running. (See Subsection 10.3.1.) In order to store the phone book in a file, some decision must be made about how the data in the phone book will be represented. For this example, I chose a simple representation in which each line of the file contains one entry consisting of a name and the associated phone number. A percent sign (`'%'`) separates the name from the number. The following code at the beginning of the program will read the phone book data file, if it exists and has the correct format:

```
File userHomeDirectory = new File( System.getProperty("user.home") );
File dataFile = new File( userHomeDirectory, ".phone_book_data" );

if ( ! dataFile.exists() ) {
   System.out.println("No phone book data file found.");
   System.out.println("A new one will be created.");
   System.out.println("File name:  " + dataFile.getAbsolutePath());
}
else {
   System.out.println("Reading phone book data...");
   try {
      Scanner scanner = new Scanner( dataFile );
      while (scanner.hasNextLine()) {
             // Read one line from the file, containing one name/number pair.
         String phoneEntry = scanner.nextLine();
         int separatorPosition = phoneEntry.indexOf('%');
         if (separatorPosition == -1)
            throw new IOException("File is not a phonebook data file.");
         name = phoneEntry.substring(0, separatorPosition);
         number = phoneEntry.substring(separatorPosition+1);
         phoneBook.put(name,number);
      }
   }
   catch (IOException e) {
      System.out.println("Error in phone book data file.");
      System.out.println("File name:  " + dataFile.getAbsolutePath());
      System.out.println("This program cannot continue.");
      System.exit(1);
   }
}
```

The program then lets the user do various things with the phone book, including making modifications. Any changes that are made are made only to the *TreeMap* that holds the data. When the program ends, the phone book data is written to the file (if any changes have been made while the program was running), using the following code:

```
if (changed) {
   System.out.println("Saving phone directory changes to file " +
         dataFile.getAbsolutePath() + " ...");
   PrintWriter out;
   try {
      out = new PrintWriter( new FileWriter(dataFile) );
   }
   catch (IOException e) {
      System.out.println("ERROR: Can't open data file for output.");
      return;
   }
   for ( Map.Entry<String,String> entry : phoneBook.entrySet() )
      out.println(entry.getKey() + "%" + entry.getValue() );
   out.close();
   if (out.checkError())
      System.out.println("ERROR: Some error occurred while writing data file.");
   else
      System.out.println("Done.");
}
```

The net effect of this is that all the data, including the changes, will be there the next time the program is run. I've shown you all the file-handling code from the program. If you would like to see the rest of the program, including an example of using a *Scanner* to read integer-valued responses from the user, see the source code file, *PhoneDirectoryFileDemo.java*.

11.3.3 Files in GUI Programs

The previous examples in this section use a command-line interface, but graphical user interface programs can also manipulate files. Programs typically have an "Open" command that reads the data from a file and displays it in a window and a "Save" command that writes the data from the window into a file. We can illustrate this in Java with a simple text editor program, *TrivialEdit.java*. The window for this program uses a *JTextArea* component to display some text that the user can edit. It also has a menu bar, with a "File" menu that includes "Open" and "Save" commands. These commands are implemented using the techniques for reading and writing files that were covered in Section 11.2.

When the user selects the Open command from the File menu in the `TrivialEdit` program, the program pops up a file dialog box where the user specifies the file. It is assumed that the file is a text file. A limit of 10000 characters is put on the size of the file, since a *JTextArea* is not meant for editing large amounts of text. The program reads the text contained in the specified file, and sets that text to be the content of the *JTextArea*. In this case, I decided to use a *BufferedReader* to read the file line-by-line. The program also sets the title bar of the window to show the name of the file that was opened. All this is done in the following method, which is just a variation of the `readFile()` method presented in Section 11.2:

```
/**
 * Carry out the Open command by letting the user specify a file to be opened
 * and reading up to 10000 characters from that file.  If the file is read
```

```
     * successfully and is not too long, then the text from the file replaces the
     * text in the JTextArea.
     */
    public void doOpen() {
       if (fileDialog == null)
          fileDialog = new JFileChooser();
       fileDialog.setDialogTitle("Select File to be Opened");
       fileDialog.setSelectedFile(null);  // No file is initially selected.
       int option = fileDialog.showOpenDialog(this);
       if (option != JFileChooser.APPROVE_OPTION)
          return;  // User canceled or clicked the dialog's close box.
       File selectedFile = fileDialog.getSelectedFile();
       BufferedReader in;
       try {
          FileReader stream = new FileReader(selectedFile);
          in = new BufferedReader( stream );
       }
       catch (Exception e) {
          JOptionPane.showMessageDialog(this,
                "Sorry, but an error occurred while trying to open the file:\n" + e);
          return;
       }
       try {
          String input = "";
          while (true) {
             String lineFromFile = in.readLine();
             if (lineFromFile == null)
                break;  // End-of-file has been reached.
             input = input + lineFromFile + '\n';
             if (input.length() > 10000)
                throw new IOException("Input file is too large for this program.");
          }
          in.close();
          text.setText(input);
          editFile = selectedFile;
          setTitle("TrivialEdit: " + editFile.getName());
       }
       catch (Exception e) {
          JOptionPane.showMessageDialog(this,
                "Sorry, but an error occurred while trying to read the data:\n" + e);
       }
    }
```

In this program, the instance variable `editFile` is used to keep track of the file that is currently being edited, if any, and the `setTitle()` method (from class *JFrame*) is used to set the title of the window to show the name of the file.

Similarly, the response to the Save command is a minor variation on the `writeFile()` method from Section 11.2. I will not repeat it here. If you would like to see the entire program, you will find the source code in the file *TrivialEdit.java*.

11.3.4 Storing Objects in Files

Whenever data is stored in files, some definite format must be adopted for representing the data. As long as the output routine that writes the data and the input routine that reads the data use the same format, the files will be usable. However, as usual, correctness is not the end of the story. The representation that is used for data in files should also be robust. (See Section 8.1.) To see what this means, we will look at several different ways of representing the same data. This example builds on the example *SimplePaint2.java* from Subsection 7.3.4. In that program, the user could use the mouse to draw simple sketches. Now, we will add file input/output capabilities to that program. This will allow the user to save a sketch to a file and later read the sketch back from the file into the program so that the user can continue to work on the sketch. The basic requirement is that all relevant data about the sketch must be saved in the file, so that the sketch can be exactly restored when the file is read by the program.

The new version of the program can be found in the source code file *SimplePaintWithFiles.java*. A "File" menu has been added to the new version. It contains two sets of Save/Open commands, one for saving and reloading sketch data in text form and one for data in binary form. We will consider both possibilities here, in some detail.

The data for a sketch consists of the background color of the picture and a list of the curves that were drawn by the user. A curve consists of a list of *Points*. (*Point* is a standard class in package `java.awt`; a *Point* `pt` has instance variables `x` and `y` of type **int** that represent the coordinates of a point on the xy-plane.) Each curve can be a different color. Furthermore, a curve can be "symmetric," which means that in addition to the curve itself, the horizontal and vertical reflections of the curve are also drawn.) The data for each curve is stored in an object of type *CurveData*, which is defined in the program as:

```
/**
 * An object of type CurveData represents the data required to redraw one
 * of the curves that have been sketched by the user.
 */
private static class CurveData implements Serializable {
   Color color;  // The color of the curve.
   boolean symmetric;  // Are horizontal and vertical reflections also drawn?
   ArrayList<Point> points;  // The points on the curve.
}
```

Note that this class has been declared to "`implement Serializable`". This allows objects of type *CurveData* to be written in binary form to an *ObjectOutputStream*. See Subsection 11.1.6.

Let's think about how the data for a sketch could be saved to an *ObjectOuputStream*. The sketch is displayed on the screen in an object of type *SimplePaintPanel*, which is a subclass of *JPanel*. All the data needed for the sketch is stored in instance variables of that object. One possibility would be to simply write the entire *SimplePaintPanel* component as a single object to the stream. The could be done in a method in the *SimplePaintPanel* class with the statement

```
outputStream.writeObject(this);
```

where `outputStream` is the *ObjectOutputStream* and "`this`" refers to the *SimplePaintPanel* itself. This statement saves the entire current state of the panel. To read the data back into the program, you would create an *ObjectInputStream* for reading the object from the file, and you would retrieve the object from the file with the statement

```
SimplePaintPanel newPanel = (SimplePaintPanel)in.readObject();
```

where `in` is the `ObjectInputStream`. Note that the type-cast is necessary because the method `in.readObject()` returns a value of type *Object*. (To get the saved sketch to appear on the screen, the `newPanel` must replace the current content pane in the program's window; furthermore, the menu bar of the window must be replaced, because the menus are associated with a particular *SimplePaintPanel* object.)

It might look tempting to be able to save data and restore it with a single command, but in this case, it's not a good idea. The main problem with doing things this way is that **the serialized form of objects that represent Swing components can change** from one version of Java to the next. This means that data files that contain serialized components such as a *SimplePaintPanel* might become unusable in the future, and the data that they contain will be effectively lost. This is an important consideration for any serious application.

Taking this into consideration, my program uses a different format when it creates a binary file. The data written to the file consists of (1) the background color of the sketch, (2) the number of curves in the sketch, and (3) all the *CurveData* objects that describe the individual curves. The method that saves the data is similar to the `writeFile()` method from Subsection 11.2.3. Here is the complete `doSaveAsBinary()` from `SimplePaintWithFiles`, with the changes from the generic `readFile()` method shown in italic:

```
/**
 * Save the user's sketch to a file in binary form as serialized
 * objects, using an ObjectOutputStream.  Files created by this method
 * can be read back into the program using the doOpenAsBinary() method.
 */
private void doSaveAsBinary() {
   if (fileDialog == null)
      fileDialog = new JFileChooser();
   File selectedFile;  //Initially selected file name in the dialog.
   if (editFile == null)
      selectedFile = new File("sketchData.binary");
   else
      selectedFile = new File(editFile.getName());
   fileDialog.setSelectedFile(selectedFile);
   fileDialog.setDialogTitle("Select File to be Saved");
   int option = fileDialog.showSaveDialog(this);
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   selectedFile = fileDialog.getSelectedFile();
   if (selectedFile.exists()) {  // Ask the user whether to replace the file.
      int response = JOptionPane.showConfirmDialog( this,
            "The file \"" + selectedFile.getName()
            + "\" already exists.\nDo you want to replace it?",
            "Confirm Save",
            JOptionPane.YES_NO_OPTION,
            JOptionPane.WARNING_MESSAGE );
      if (response != JOptionPane.YES_OPTION)
         return;  // User does not want to replace the file.
   }
   ObjectOutputStream out;
   try {
      FileOutputStream stream = new FileOutputStream(selectedFile);
      out = new ObjectOutputStream( stream );
   }
```

```
      catch (Exception e) {
         JOptionPane.showMessageDialog(this,
            "Sorry, but an error occurred while trying to open the file:\n" + e);
         return;
      }
      try {
         out.writeObject(getBackground());
         out.writeInt(curves.size());
         for ( CurveData curve : curves )
            out.writeObject(curve);
         out.close();
         editFile = selectedFile;
         setTitle("SimplePaint: " + editFile.getName());
      }
      catch (Exception e) {
         JOptionPane.showMessageDialog(this,
            "Sorry, but an error occurred while trying to write the text:\n" + e);
      }
   }
```

The heart of this method consists of the following lines, which do the actual writing of the data to the file:

```
out.writeObject(getBackground()); // Writes the panel's background color.
out.writeInt(curves.size());      // Writes the number of curves.
for ( CurveData curve : curves )  // For each curve...
   out.writeObject(curve);        //   write the corresponding CurveData object.
```

The `doOpenAsBinary()` method, which is responsible for reading sketch data back into the program from an *ObjectInputStream*, has to read exactly the same data that was written, in the same order, and use that data to build the data structures that will represent the sketch while the program is running. Once the data structures have been successfully built, they replace the data structures that describe the previous contents of the panel. This is done as follows:

```
/* Read data from the file into local variables */

Color newBackgroundColor = (Color)in.readObject();
int curveCount = in.readInt();
ArrayList<CurveData> newCurves = new ArrayList<CurveData>();
for (int i = 0; i < curveCount; i++)
   newCurves.add( (CurveData)in.readObject() );
in.close();

/* Copy the data that was read into the instance variables that
   describe the sketch that is displayed by the program.*/

curves = newCurves;
setBackground(newBackgroundColor);
repaint();
```

This is only a little harder than saving the entire *SimplePaintPanel* component to the file in one step, and it is more robust since the serialized form of the objects that are saved to file is unlikely to change in the future. But it still suffers from the general fragility of binary data.

* * *

An alternative to using object streams is to save the data in human-readable, character form. The basic idea is the same: All the data necessary to reconstitute a sketch must be saved to the output file in some definite format. The method that reads the file must follow exactly the same format as it reads the data, and it must use the data to rebuild the data structures that represent the sketch while the program is running.

When writing character data, we can't write out entire objects in one step. All the data has to be expressed, ultimately, in terms of simple data values such as strings and primitive type values. A color, for example, can be expressed in terms of three integers giving the red, green, and blue components of the color. The first (not very good) idea that comes to mind might be to just dump all the necessary data, in some definite order, into the file. Suppose that `out` is a *PrintWriter* that is used to write to the file. We could then say:

```
Color bgColor = getBackground();    // Write the background color to the file.
out.println( bgColor.getRed() );
out.println( bgColor.getGreen() );
out.println( bgColor.getBlue() );

out.println( curves.size() );       // Write the number of curves.

for ( CurveData curve : curves ) {  // For each curve, write...
   out.println( curve.color.getRed() );      // the color of the curve
   out.println( curve.color.getGreen() );
   out.println( curve.color.getBlue() );
   out.println( curve.symmetric ? 0 : 1 );   // the curve's symmetry property
   out.println( curve.points.size() );       // the number of points on curve
   for ( Point pt : curve.points ) {         // the coordinates of each point
      out.println( pt.x );
      out.println( pt.y );
   }
}
```

This works in the sense that the file-reading method can read the data and rebuild the data structures. Suppose that the input method uses a *Scanner* named `scanner` to read the data file (see Subsection 11.1.5). Then it could say:

```
Color newBackgroundColor;                // Read the background Color.
int red = scanner.nextInt();
int green = scanner.nextInt();
int blue = scanner.nextInt();
newBackgroundColor = new Color(red,green,blue);

ArrayList<CurveData> newCurves = new ArrayList<CurveData>();

int curveCount = scanner.nextInt();      // The number of curves to be read.
for (int i = 0; i < curveCount; i++) {
   CurveData curve = new CurveData();
   int r = scanner.nextInt();            // Read the curve's color.
   int g = scanner.nextInt();
   int b = scanner.nextInt();
   curve.color = new Color(r,g,b);
   int symmetryCode = scanner.nextInt(); // Read the curve's symmetry property.
   curve.symmetric = (symmetryCode == 1);
   curveData.points = new ArrayList<Point>();
   int pointCount = scanner.nextInt();  // The number of points on this curve.
   for (int j = 0; j < pointCount; j++) {
```

```
            int x = scanner.nextInt();        // Read the coordinates of the point.
            int y = scanner.nextInt();
            curveData.points.add(new Point(x,y));
         }
         newCurves.add(curve);
      }

      curves = newCurves;                     // Install the new data structures.
      setBackground(newBackgroundColor);
```

Note how every piece of data that was written by the output method is read, in the same order, by the input method. While this does work, the data file is just a long string of numbers. It doesn't make much more sense to a human reader than a binary-format file would. Furthermore, it is still fragile in the sense that any small change made to the data representation in the program, such as adding a new property to curves, will render the data file useless (unless you happen to remember exactly which version of the program created the file).

So, I decided to use a more complex, more meaningful data format for the text files created by my program. Instead of just writing numbers, I add **words** to say what the numbers mean. Here is a short but complete data file for the program; just by looking at it, you can probably tell what is going on:

```
SimplePaintWithFiles 1.0
background 110 110 180

startcurve
  color 255 255 255
  symmetry true
  coords 10 10
  coords 200 250
  coords 300 10
endcurve

startcurve
  color 0 255 255
  symmetry false
  coords 10 400
  coords 590 400
endcurve
```

The first line of the file identifies the program that created the data file; when the user selects a file to be opened, the program can check the first word in the file as a simple test to make sure the the file is of the correct type. The first line also contains a version number, 1.0. If the file format changes in a later version of the program, a higher version number would be used; if the program sees a version number of 1.2 in a file, but the program only understands version 1.0, the program can explain to the user that a newer version of the program is needed to read the data file.

The second line of the file specifies the background color of the picture. The three integers specify the red, green, and blue components of the color. The word "background" at the beginning of the line makes the meaning clear. The remainder of the file consists of data for the curves that appear in the picture. The data for each curve is clearly marked with "startcurve" and "endcurve." The data consists of the color and symmetry properties of the curve and the xy-coordinates of each point on the curve. Again, the meaning is clear. Files in this format can easily be created or edited by hand. In fact, the data file shown above was actually created in

a text editor rather than by the program. Furthermore, it's easy to extend the format to allow for additional options. Future versions of the program could add a "thickness" property to the curves to make it possible to have curves that are more than one pixel wide. Shapes such as rectangles and ovals could easily be added.

Outputting data in this format is easy. Suppose that `out` is a *PrintWriter* that is being used to write the sketch data to a file. Then the output can be done with:

```
out.println("SimplePaintWithFiles 1.0"); // Version number.
Color bgColor = getBackground();
out.println( "background " + bgColor.getRed() + " " +
      bgColor.getGreen() + " " + bgColor.getBlue() );
for ( CurveData curve : curves ) {
   out.println();
   out.println("startcurve");
   out.println("  color " + curve.color.getRed() + " " +
         curve.color.getGreen() + " " + curve.color.getBlue() );
   out.println( "  symmetry " + curve.symmetric );
   for ( Point pt : curve.points )
      out.println( "  coords " + pt.x + " " + pt.y );
   out.println("endcurve");
}
```

Reading the data is somewhat harder, since the input routine has to deal with all the extra words in the data. In my input routine, I decided to allow some variation in the order in which the data occurs in the file. For example, the background color can be specified at the end of the file, instead of at the beginning. It can even be left out altogether, in which case white will be used as the default background color. This is possible because each item of data is labeled with a word that describes its meaning; the labels can be used to drive the processing of the input. Here is the complete method from *SimplePaintWithFiles.java* that reads data files in text format. It uses a *Scanner* to read items from the file:

```
private void doOpenAsText() {

   if (fileDialog == null)
      fileDialog = new JFileChooser();
   fileDialog.setDialogTitle("Select File to be Opened");
   fileDialog.setSelectedFile(null);  // No file is initially selected.
   int option = fileDialog.showOpenDialog(this);
   if (option != JFileChooser.APPROVE_OPTION)
      return;  // User canceled or clicked the dialog's close box.
   File selectedFile = fileDialog.getSelectedFile();

   Scanner scanner;  // For reading from the data file.
   try {
      Reader stream = new BufferedReader(new FileReader(selectedFile));
      scanner = new Scanner( stream );
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
            "Sorry, but an error occurred while trying to open the file:\n" + e);
      return;
   }

   try {  // Read the contents of the file.
      String programName = scanner.next();
```

```
if ( ! programName.equals("SimplePaintWithFiles") )
   throw new IOException("File is not a SimplePaintWithFiles data file.");
double version = scanner.nextDouble();
if (version > 1.0)
   throw new IOException("File requires newer version of this program.");
Color newBackgroundColor = Color.WHITE;
ArrayList<CurveData> newCurves = new ArrayList<CurveData>();
while (scanner.hasNext()) {
   String itemName = scanner.next();
   if (itemName.equalsIgnoreCase("background")) {
      int red = scanner.nextInt();
      int green = scanner.nextInt();
      int blue = scanner.nextInt();
      newBackgroundColor = new Color(red,green,blue);
   }
   else if (itemName.equalsIgnoreCase("startcurve")) {
      CurveData curve = new CurveData();
      curve.color = Color.BLACK;
      curve.symmetric = false;
      curve.points = new ArrayList<Point>();
      itemName = scanner.next();
      while ( ! itemName.equalsIgnoreCase("endcurve") ) {
         if (itemName.equalsIgnoreCase("color")) {
            int r = scanner.nextInt();
            int g = scanner.nextInt();
            int b = scanner.nextInt();
            curve.color = new Color(r,g,b);
         }
         else if (itemName.equalsIgnoreCase("symmetry")) {
            curve.symmetric = scanner.nextBoolean();
         }
         else if (itemName.equalsIgnoreCase("coords")) {
            int x = scanner.nextInt();
            int y = scanner.nextInt();
            curve.points.add( new Point(x,y) );
         }
         else {
            throw new Exception("Unknown term in input.");
         }
         itemName = scanner.next();
      }
      newCurves.add(curve);
   }
   else {
      throw new Exception("Unknown term in input.");
   }
}

scanner.close();
setBackground(newBackgroundColor);  // Install the new picture data.
curves = newCurves;
repaint();
editFile = selectedFile;
setTitle("SimplePaint: " + editFile.getName());
```

```
        }
        catch (Exception e) {
           JOptionPane.showMessageDialog(this,
                 "Sorry, but an error occurred while trying to read the data:\n" + e);
        }
     }
```

The main reason for this long discussion of file formats has been to get you to think about the problem of representing complex data in a form suitable for storing the data in a file. The same problem arises when data must be transmitted over a network. There is no one correct solution to the problem, but some solutions are certainly better than others. In Section 11.6, we will look at one solution to the data representation problem that has become increasingly common.

* * *

In addition to being able to save sketch data in both text form and binary form, `SimplePaintWithFiles` can also save the picture itself as an image file that could be, for example, printed or put on a web page. This is a preview of image-handling techniques that will be covered in Chapter 12.

11.4 Networking

As far as a program is concerned, a network is just another possible source of input data, and another place where data can be output. That does oversimplify things, because networks are not as easy to work with as files are. But in Java, you can do network communication using input streams and output streams, just as you can use such streams to communicate with the user or to work with files. Nevertheless, opening a network connection between two computers is a bit tricky, since there are two computers involved and they have to somehow agree to open a connection. And when each computer can send data to the other, synchronizing communication can be a problem. But the fundamentals are the same as for other forms of I/O.

One of the standard Java packages is called `java.net`. This package includes several classes that can be used for networking. Two different styles of network I/O are supported. One of these, which is fairly high-level, is based on the World-Wide Web, and provides the sort of network communication capability that is used by a Web browser when it downloads pages for you to view. The main classes for this style of networking are `java.net.URL` and `java.net.URLConnection`. An object of type *URL* is an abstract representation of a ***Universal Resource Locator***, which is an address for an HTML document or other resource on the Web. A *URLConnection* represents a network connection to such a resource.

The second style of I/O, which is more general and much more important, views the network at a lower level. It is based on the idea of a ***socket***. A socket is used by a program to establish a connection with another program on a network. Communication over a network involves two sockets, one on each of the computers involved in the communication. Java uses a class called `java.net.Socket` to represent sockets that are used for network communication. The term "socket" presumably comes from an image of physically plugging a wire into a computer to establish a connection to a network, but it is important to understand that a socket, as the term is used here, is simply an object belonging to the class *Socket*. In particular, a program can have several sockets at the same time, each connecting it to another program running on some other computer on the network. All these connections use the same physical network connection.

This section gives a brief introduction to these basic networking classes, and shows how they relate to input and output streams.

11.4.1 URLs and URLConnections

The *URL* class is used to represent resources on the World-Wide Web. Every resource has an address, which identifies it uniquely and contains enough information for a Web browser to find the resource on the network and retrieve it. The address is called a "url" or "universal resource locator."

An object belonging to the *URL* class represents such an address. Once you have a *URL* object, you can use it to open a *URLConnection* to the resource at that address. A url is ordinarily specified as a string, such as "`http://math.hws.edu/eck/index.html`". There are also ***relative url's***. A relative url specifies the location of a resource relative to the location of another url, which is called the ***base*** or ***context*** for the relative url. For example, if the context is given by the url `http://math.hws.edu/eck/`, then the incomplete, relative url "`index.html`" would really refer to `http://math.hws.edu/eck/index.html`.

An object of the class *URL* is not simply a string, but it can be constructed from a string representation of a url. A *URL* object can also be constructed from another *URL* object, representing a context, and a string that specifies a url relative to that context. These constructors have prototypes

```
public URL(String urlName) throws MalformedURLException
```

and

```
public URL(URL context, String relativeName) throws MalformedURLException
```

Note that these constructors will throw an exception of type *MalformedURLException* if the specified strings don't represent legal url's. The *MalformedURLException* class is a subclass of *IOException*, and it requires mandatory exception handling. That is, you must call the constructor inside a `try..catch` statement that handles the exception or in a subroutine that is declared to throw the exception.

The second constructor is especially convenient when writing applets. In an applet, two methods are available that provide useful URL contexts. The method `getDocumentBase()`, defined in the *Applet* and *JApplet* classes, returns an object of type *URL*. This *URL* represents the location from which the HTML page that contains the applet was downloaded. This allows the applet to go back and retrieve other files that are stored in the same location as that document. For example,

```
URL url = new URL(getDocumentBase(), "data.txt");
```

constructs a *URL* that refers to a file named `data.txt` on the same computer and in the same directory as the source file for the web page on which the applet is running. Another method, `getCodeBase()`, returns a *URL* that gives the location of the applet class file (which is not necessarily the same as the location of the document).

Once you have a valid *URL* object, you can call its `openConnection()` method to set up a connection. This method returns a *URLConnection*. The *URLConnection* object can, in turn, be used to create an *InputStream* for reading data from the resource represented by the URL. This is done by calling its `getInputStream()` method. For example:

```
URL url = new URL(urlAddressString);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
```

The `openConnection()` and `getInputStream()` methods can both throw exceptions of type *IOException*. Once the *InputStream* has been created, you can read from it in the usual way, including wrapping it in another input stream type, such as *TextReader*, or using a *Scanner*. Reading from the stream can, of course, generate exceptions.

One of the other useful instance methods in the *URLConnection* class is `getContentType()`, which returns a *String* that describes the type of information available from the URL. The return value can be `null` if the type of information is not yet known or if it is not possible to determine the type. The type might not be available until after the input stream has been created, so you should generally call `getContentType()` after `getInputStream()`. The string returned by `getContentType()` is in a format called a ***mime type***. Mime types include "text/plain", "text/html", "image/jpeg", "image/gif", and many others. All mime types contain two parts: a general type, such as "text" or "image", and a more specific type within that general category, such as "html" or "gif". If you are only interested in text data, for example, you can check whether the string returned by `getContentType()` starts with "text". (Mime types were first introduced to describe the content of email messages. The name stands for "Multipurpose Internet Mail Extensions." They are now used almost universally to specify the type of information in a file or other resource.)

Let's look at a short example that uses all this to read the data from a URL. This subroutine opens a connection to a specified URL, checks that the type of data at the URL is text, and then copies the text onto the screen. Many of the operations in this subroutine can throw exceptions. They are handled by declaring that the subroutine "`throws IOException`" and leaving it up to the main program to decide what to do when an error occurs.

```
static void readTextFromURL( String urlString ) throws IOException {

   /* Open a connection to the URL, and get an input stream
      for reading data from the URL. */

   URL url = new URL(urlString);
   URLConnection connection = url.openConnection();
   InputStream urlData = connection.getInputStream();

   /* Check that the content is some type of text. */

   String contentType = connection.getContentType();
   if (contentType == null || contentType.startsWith("text") == false)
      throw new IOException("URL does not seem to refer to a text file.");

   /* Copy lines of text from the input stream to the screen, until
      end-of-file is encountered  (or an error occurs). */

   BufferedReader in;  // For reading from the connection's input stream.
   in = new BufferedReader( new InputStreamReader(urlData) );

   while (true) {
      String line = in.readLine();
      if (line == null)
         break;
      System.out.println(line);
   }

} // end readTextFromURL()
```

A complete program that uses this subroutine can be found in the file *ReadURL.java*. When using the program, note that you have to specify a complete url, including the "`http://`" at

the beginning. There is also an applet version of the program, which you can find in the on-line version of this section.

11.4.2 TCP/IP and Client/Server

Communication over the Internet is based on a pair of protocols called the ***Transmission Control Protocol*** and the ***Internet Protocol***, which are collectively referred to as ***TCP/IP***. (In fact, there is a more basic communication protocol called UDP that can be used instead of TCP in certain applications. UDP is supported in Java, but for this discussion, I'll stick to the full TCP/IP, which provides reliable two-way communication between networked computers.)

For two programs to communicate using TCP/IP, each program must create a socket, as discussed earlier in this section, and those sockets must be connected. Once such a connection is made, communication takes place using input streams and output streams. Each program has its own input stream and its own output stream. Data written by one program to its output stream is transmitted to the other computer. There, it enters the input stream of the program at the other end of the network connection. When that program reads data from its input stream, it is receiving the data that was transmitted to it over the network.

The hard part, then, is making a network connection in the first place. Two sockets are involved. To get things started, one program must create a socket that will wait passively until a connection request comes in from another socket. The waiting socket is said to be ***listening*** for a connection. On the other side of the connection-to-be, another program creates a socket that sends out a connection request to the listening socket. When the listening socket receives the connection request, it responds, and the connection is established. Once that is done, each program can obtain an input stream and an output stream for sending data over the connection. Communication takes place through these streams until one program or the other ***closes*** the connection.

A program that creates a listening socket is sometimes said to be a ***server***, and the socket is called a ***server socket***. A program that connects to a server is called a ***client***, and the socket that it uses to make a connection is called a ***client socket***. The idea is that the server is out there somewhere on the network, waiting for a connection request from some client. The server can be thought of as offering some kind of service, and the client gets access to that service by connecting to the server. This is called the ***client/server model*** of network communication. In many actual applications, a server program can provide connections to several clients at the same time. When a client connects to a server's listening socket, that socket does not stop listening. Instead, it continues listening for additional client connections at the same time that the first client is being serviced. To do this, it is necessary to use threads (Section 8.5). We'll look at how it works in the next section.

The *URL* class that was discussed at the beginning of this section uses a client socket behind the scenes to do any necessary network communication. On the other side of that connection is a server program that accepts a connection request from the *URL* object, reads a request from that object for some particular file on the server computer, and responds by transmitting the contents of that file over the network back to the *URL* object. After transmitting the data, the server closes the connection.

* * *

A client program has to have some way to specify which computer, among all those on the network, it wants to communicate with. Every computer on the Internet has an ***IP address*** which identifies it uniquely among all the computers on the net. Many computers can also

be referred to by ***domain names*** such as math.hws.edu or www.whitehouse.gov. (See Section 1.7.) Traditional (or ***IPv4***) IP addresses are 32-bit integers. They are usually written in the so-called "dotted decimal" form, such as `69.9.161.200`, where each of the four numbers in the address represents an 8-bit integer in the range 0 through 255. A new version of the Internet Protocol, ***IPv6***, is currently being introduced. IPv6 addresses are 128-bit integers and are usually written in hexadecimal form (with some colons and maybe some extra information thrown in). In actual use, IPv6 addresses are still fairly rare.

A computer can have several IP addresses, and can have both IPv4 and IPv6 addresses. Usually, one of these is the ***loopback address***, which can be used when a program wants to communicate with another program *on the same computer*. The loopback address has IPv4 address 127.0.0.1 and can also, in general, be referred to using the domain name ***localhost***. In addition, there can be one or more IP addresses associated with physical network connections. Your computer probably has some utility for displaying your computer's IP addresses. I have written a small Java program, *ShowMyNetwork.java*, that does the same thing. When I run `ShowMyNetwork` on my computer, the output is:

```
en1 :  /192.168.1.47  /fe80:0:0:0:211:24ff:fe9c:5271%5
lo0 :  /127.0.0.1  /fe80:0:0:0:0:0:0:1%1  /0:0:0:0:0:0:0:1%0
```

The first thing on each line is a network interface name, which is really meaningful only to the computer's operating system. The output also contains the IP addresses for that interface. In this example, `lo0` refers to the loopback address, which has IPv4 address 127.0.0.1 as usual. The most important number here is `192.168.1.47`, which is the IPv4 address that can be used for communication over the network.

Now, a single computer might have several programs doing network communication at the same time, or one program communicating with several other computers. To allow for this possibility, a network connection is actually identified by a ***port number*** in combination with an IP address. A port number is just a 16-bit integer. A server does not simply listen for connections—it listens for connections *on a particular port*. A potential client must know both the Internet address (or domain name) of the computer on which the server is running and the port number on which the server is listening. A Web server, for example, generally listens for connections on port 80; other standard Internet services also have standard port numbers. (The standard port numbers are all less than 1024, and are reserved for particular services. If you create your own server programs, you should use port numbers greater than 1024.)

11.4.3 Sockets

To implement TCP/IP connections, the `java.net` package provides two classes, *ServerSocket* and *Socket*. A *ServerSocket* represents a listening socket that waits for connection requests from clients. A *Socket* represents one endpoint of an actual network connection. A *Socket* can be a client socket that sends a connection request to a server. But a *Socket* can also be created by a server to handle a connection request from a client. This allows the server to create multiple sockets and handle multiple connections. A *ServerSocket* does not itself participate in connections; it just listens for connection requests and creates *Sockets* to handle the actual connections.

When you construct a `ServerSocket` object, you have to specify the port number on which the server will listen. The specification for the constructor is

```
public ServerSocket(int port) throws IOException
```

The port number must be in the range 0 through 65535, and should generally be greater than 1024. (A value of 0 tells the server socket to listen on any available port.) The constructor might throw a *SecurityException* if a smaller port number is specified. An *IOException* can occur if, for example, the specified port number is already in use.

As soon as a *ServerSocket* is created, it starts listening for connection requests. The `accept()` method in the *ServerSocket* class accepts such a request, establishes a connection with the client, and returns a *Socket* that can be used for communication with the client. The `accept()` method has the form

```
public Socket accept() throws IOException
```

When you call the `accept()` method, it will not return until a connection request is received (or until some error occurs). The method is said to ***block*** while waiting for the connection. (While the method is blocked, the thread that called the method can't do anything else. However, other threads in the same program can proceed.) You can call `accept()` repeatedly to accept multiple connection requests. The *ServerSocket* will continue listening for connections until it is closed, using its `close()` method, or until some error occurs, or until the program is terminated in some way.

Suppose that you want a server to listen on port 1728, and suppose that you've written a method `provideService(Socket)` to handle the communication with one client. Then the basic form of the server program would be:

```
try {
   ServerSocket server = new ServerSocket(1728);
   while (true) {
      Socket connection = server.accept();
      provideService(connection);
   }
}
catch (IOException e) {
   System.out.println("Server shut down with error: " + e);
}
```

On the client side, a client socket is created using a constructor in the *Socket* class. To connect to a server on a known computer and port, you would use the constructor

```
public Socket(String computer, int port) throws IOException
```

The first parameter can be either an IP number or a domain name. This constructor will block until the connection is established or until an error occurs.

Once you have a connected socket, no matter how it was created, you can use the *Socket* methods `getInputStream()` and `getOutputStream()` to obtain streams that can be used for communication over the connection. These methods return objects of type *InputStream* and *OutputStream*, respectively. Keeping all this in mind, here is the outline of a method for working with a client connection:

```
/**
 * Open a client connection to a specified server computer and
 * port number on the server, and then do communication through
 * the connection.
 */
void doClientConnection(String computerName, int serverPort) {
   Socket connection;
   InputStream in;
```

```
        OutputStream out;
        try {
           connection = new Socket(computerName,serverPort);
           in = connection.getInputStream();
           out = connection.getOutputStream();
        }
        catch (IOException e) {
           System.out.println(
               "Attempt to create connection failed with error: " + e);
           return;
        }
         .
         .  // Use the streams, in and out, to communicate with server.
         .
        try {
           connection.close();
               // (Alternatively, you might depend on the server
               //  to close the connection.)
        }
        catch (IOException e) {
        }
    }  // end doClientConnection()
```

All this makes network communication sound easier than it really is. (And if you think it sounded hard, then it's even harder.) If networks were completely reliable, things would be almost as easy as I've described. The problem, though, is to write robust programs that can deal with network and human error. I won't go into detail here. However, what I've covered here should give you the basic ideas of network programming, and it is enough to write some simple network applications. Let's look at a few working examples of client/server programming.

11.4.4 A Trivial Client/Server

The first example consists of two programs. The source code files for the programs are *DateClient.java* and *DateServer.java*. One is a simple network client and the other is a matching server. The client makes a connection to the server, reads one line of text from the server, and displays that text on the screen. The text sent by the server consists of the current date and time on the computer where the server is running. In order to open a connection, the client must know the computer on which the server is running and the port on which it is listening. The server listens on port number 32007. The port number could be anything between 1025 and 65535, as long the server and the client use the same port. Port numbers between 1 and 1024 are reserved for standard services and should not be used for other servers. The name or IP number of the computer on which the server is running must be specified as a command-line argument For example, if the server is running on a computer named math.hws.edu, then you would typically run the client with the command "`java DateClient math.hws.edu`". Here is the complete client program:

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

/**
 * This program opens a connection to a computer specified
 * as the first command-line argument.  The connection is made to
 * the port specified by LISTENING_PORT.  The program reads one
```

```
 * line of text from the connection and then closes the
 * connection.  It displays the text that it read on
 * standard output.  This program is meant to be used with
 * the server program, DateServer, which sends the current
 * date and time on the computer where the server is running.
 */
public class DateClient {

   public static final int LISTENING_PORT = 32007;

   public static void main(String[] args) {

      String hostName;         // Name of the server computer to connect to.
      Socket connection;       // A socket for communicating with server.
      BufferedReader incoming; // For reading data from the connection.

      /* Get computer name from command line. */

      if (args.length > 0)
         hostName = args[0];
      else {
            // No computer name was given.  Print a message and exit.
         System.out.println("Usage:  java DateClient <server_host_name>");
         return;
      }

      /* Make the connection, then read and display a line of text. */

      try {
         connection = new Socket( hostName, LISTENING_PORT );
         incoming = new BufferedReader(
                          new InputStreamReader(connection.getInputStream()) );
         String lineFromServer = incoming.readLine();
         if (lineFromServer == null) {
               // A null from incoming.readLine() indicates that
               // end-of-stream was encountered.
            throw new IOException("Connection was opened, " +
                  "but server did not send any data.");
         }
         System.out.println();
         System.out.println(lineFromServer);
         System.out.println();
         incoming.close();
      }
      catch (Exception e) {
         System.out.println("Error:  " + e);
      }

   }  // end main()

} //end class DateClient
```

Note that all the communication with the server is done in a `try..catch` statement. This will catch the *IOExceptions* that can be generated when the connection is opened or closed and when data is read from the input stream. The connection's input stream is wrapped in a *BufferedReader*, which has a `readLine()` method that makes it easy to read one line of text. (See Subsection 11.1.4.)

In order for this program to run without error, the server program must be running on the computer to which the client tries to connect. By the way, it's possible to run the client and the server program on the same computer. For example, you can open two command windows, start the server in one window and then run the client in the other window. To make things like this easier, most computers will recognize the domain name `localhost` and the IP number `127.0.0.1` as referring to "this computer." This means that the command "`java DateClient localhost`" will tell the *DateClient* program to connect to a server running on the same computer. If that command doesn't work, try "`java DateClient 127.0.0.1`".

The server program that corresponds to the *DateClient* client program is called *DateServer*. The *DateServer* program creates a *ServerSocket* to listen for connection requests on port 32007. After the listening socket is created, the server will enter an infinite loop in which it accepts and processes connections. This will continue until the program is killed in some way—for example by typing a `CONTROL-C` in the command window where the server is running. When a connection is received from a client, the server calls a subroutine to handle the connection. In the subroutine, any *Exception* that occurs is caught, so that it will not crash the server. Just because a connection to one client has failed for some reason, it does not mean that the server should be shut down; the error might have been the fault of the client. The connection-handling subroutine creates a *PrintWriter* for sending data over the connection. It writes the current date and time to this stream and then closes the connection. (The standard class `java.util.Date` is used to obtain the current time. An object of type *Date* represents a particular date and time. The default constructor, "`new Date()`", creates an object that represents the time when the object is created.) The complete server program is as follows:

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

/**
 * This program is a server that takes connection requests on
 * the port specified by the constant LISTENING_PORT.  When a
 * connection is opened, the program sends the current time to
 * the connected socket.  The program will continue to receive
 * and process connections until it is killed (by a CONTROL-C,
 * for example).  Note that this server processes each connection
 * as it is received, rather than creating a separate thread
 * to process the connection.
 */
public class DateServer {

   public static final int LISTENING_PORT = 32007;

   public static void main(String[] args) {

      ServerSocket listener;  // Listens for incoming connections.
      Socket connection;      // For communication with the connecting program.

      /* Accept and process connections forever, or until some error occurs.
         (Note that errors that occur while communicating with a connected
         program are caught and handled in the sendDate() routine, so
         they will not crash the server.) */

      try {
         listener = new ServerSocket(LISTENING_PORT);
         System.out.println("Listening on port " + LISTENING_PORT);
```

```
            while (true) {
                   // Accept next connection request and handle it.
               connection = listener.accept();
               sendDate(connection);
            }
         }
         catch (Exception e) {
            System.out.println("Sorry, the server has shut down.");
            System.out.println("Error:  " + e);
            return;
         }

      }  // end main()

      /**
       * The parameter, client, is a socket that is already connected to another
       * program.  Get an output stream for the connection, send the current time,
       * and close the connection.
       */
      private static void sendDate(Socket client) {
         try {
            System.out.println("Connection from " +
                                          client.getInetAddress().toString() );
            Date now = new Date();  // The current date and time.
            PrintWriter outgoing;   // Stream for sending data.
            outgoing = new PrintWriter( client.getOutputStream() );
            outgoing.println( now.toString() );
            outgoing.flush();  // Make sure the data is actually sent!
            client.close();
         }
         catch (Exception e){
            System.out.println("Error: " + e);
         }
      } // end sendDate()

   } //end class DateServer
```

When you run *DateServer* in a command-line interface, it will sit and wait for connection requests and report them as they are received. To make the *DateServer* service permanently available on a computer, the program really should be run as a ***daemon***. A daemon is a program that runs continually on a computer, independently of any user. The computer can be configured to start the daemon automatically as soon as the computer boots up. It then runs in the background, even while the computer is being used for other purposes. For example, a computer that makes pages available on the World Wide Web runs a daemon that listens for requests for pages and responds by transmitting the pages. It's just a souped-up analog of the *DateServer* program! However, the question of how to set up a program as a daemon is not one I want to go into here. For testing purposes, it's easy enough to start the program by hand, and, in any case, my examples are not really robust enough or full-featured enough to be run as serious servers. (By the way, the word "daemon" is just an alternative spelling of "demon" and is usually pronounced the same way.)

Note that after calling `out.println()` to send a line of data to the client, the server program calls `out.flush()`. The `flush()` method is available in every output stream class. Calling it

ensures that data that has been written to the stream is actually sent to its destination. You should generally call this function every time you use an output stream to send data over a network connection. If you don't do so, it's possible that the stream will collect data until it has a large batch of data to send. This is done for efficiency, but it can impose unacceptable delays when the client is waiting for the transmission. It is even possible that some of the data might remain untransmitted when the socket is closed, so it is especially important to call `flush()` before closing the connection. This is one of those unfortunate cases where different implementations of Java can behave differently. If you fail to flush your output streams, it is possible that your network application will work on some types of computers but not on others.

11.4.5 A Simple Network Chat

In the *DateServer* example, the server transmits information and the client reads it. It's also possible to have two-way communication between client and server. As a first example, we'll look at a client and server that allow a user on each end of the connection to send messages to the other user. The program works in a command-line interface where the users type in their messages. In this example, the server waits for a connection from a single client and then closes down its listener so that no other clients can connect. After the client and server are connected, both ends of the connection work in much the same way. The user on the client end types a message, and it is transmitted to the server, which displays it to the user on that end. Then the user of the server types a message that is transmitted to the client. Then the client user types another message, and so on. This continues until one user or the other enters "quit" when prompted for a message. When that happens, the connection is closed and both programs terminate. The client program and the server program are very similar. The techniques for opening the connections differ, and the client is programmed to send the first message while the server is programmed to receive the first message. The client and server programs can be found in the files *CLChatClient.java* and *CLChatServer.java*. (The name "CLChat" stands for "command-line chat.") Here is the source code for the server:

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

/**
 * This program is one end of a simple command-line interface chat program.
 * It acts as a server which waits for a connection from the CLChatClient
 * program.  The port on which the server listens can be specified as a
 * command-line argument.  If it is not, then the port specified by the
 * constant DEFAULT_PORT is used.  Note that if a port number of zero is
 * specified, then the server will listen on any available port.
 * This program only supports one connection.  As soon as a connection is
 * opened, the listening socket is closed down.  The two ends of the connection
 * each send a HANDSHAKE string to the other, so that both ends can verify
 * that the program on the other end is of the right type.  Then the connected
 * programs alternate sending messages to each other.  The client always sends
 * the first message.  The user on either end can close the connection by
 * entering the string "quit" when prompted for a message.  Note that the first
 * character of any string sent over the connection must be 0 or 1; this
 * character is interpreted as a command.
 */
public class CLChatServer {

   /**
```

```
 * Port to listen on, if none is specified on the command line.
 */
static final int DEFAULT_PORT = 1728;

/**
 * Handshake string. Each end of the connection sends this  string to the
 * other just after the connection is opened.  This is done to confirm that
 * the program on the other side of the connection is a CLChat program.
 */
static final String HANDSHAKE = "CLChat";

/**
 * This character is prepended to every message that is sent.
 */
static final char MESSAGE = '0';

/**
 * This character is sent to the connected program when the user quits.
 */
static final char CLOSE = '1';


public static void main(String[] args) {

   int port;   // The port on which the server listens.

   ServerSocket listener;  // Listens for a connection request.
   Socket connection;      // For communication with the client.

   BufferedReader incoming;  // Stream for receiving data from client.
   PrintWriter outgoing;     // Stream for sending data to client.
   String messageOut;        // A message to be sent to the client.
   String messageIn;         // A message received from the client.

   BufferedReader userInput; // A wrapper for System.in, for reading
                             // lines of input from the user.

   /* First, get the port number from the command line,
      or use the default port if none is specified. */

   if (args.length == 0)
      port = DEFAULT_PORT;
   else {
      try {
         port= Integer.parseInt(args[0]);
         if (port < 0 || port > 65535)
            throw new NumberFormatException();
      }
      catch (NumberFormatException e) {
         System.out.println("Illegal port number, " + args[0]);
         return;
      }
   }

   /* Wait for a connection request.  When it arrives, close
      down the listener.  Create streams for communication
      and exchange the handshake. */

   try {
```

```
      listener = new ServerSocket(port);
      System.out.println("Listening on port " + listener.getLocalPort());
      connection = listener.accept();
      listener.close();
      incoming = new BufferedReader(
                      new InputStreamReader(connection.getInputStream()) );
      outgoing = new PrintWriter(connection.getOutputStream());
      outgoing.println(HANDSHAKE);  // Send handshake to client.
      outgoing.flush();
      messageIn = incoming.readLine();  // Receive handshake from client.
      if (! HANDSHAKE.equals(messageIn) ) {
         throw new Exception("Connected program is not a CLChat!");
      }
      System.out.println("Connected.  Waiting for the first message.");
   }
   catch (Exception e) {
      System.out.println("An error occurred while opening connection.");
      System.out.println(e.toString());
      return;
   }

   /* Exchange messages with the other end of the connection until one side
      or the other closes the connection.  This server program waits for
      the first message from the client.  After that, messages alternate
      strictly back and forth. */

   try {
      userInput = new BufferedReader(new InputStreamReader(System.in));
      System.out.println("NOTE: Enter 'quit' to end the program.\n");
      while (true) {
         System.out.println("WAITING...");
         messageIn = incoming.readLine();
         if (messageIn.length() > 0) {
                // The first character of the message is a command. If
                // the command is CLOSE, then the connection is closed.
                // Otherwise, remove the command character from the
                // message and proceed.
            if (messageIn.charAt(0) == CLOSE) {
               System.out.println("Connection closed at other end.");
               connection.close();
               break;
            }
            messageIn = messageIn.substring(1);
         }
         System.out.println("RECEIVED:  " + messageIn);
         System.out.print("SEND:      ");
         messageOut = userInput.readLine();
         if (messageOut.equalsIgnoreCase("quit"))  {
               // User wants to quit.  Inform the other side
               // of the connection, then close the connection.
            outgoing.println(CLOSE);
            outgoing.flush();  // Make sure the data is sent!
            connection.close();
            System.out.println("Connection closed.");
```

```
                break;
             }
             outgoing.println(MESSAGE + messageOut);
             outgoing.flush(); // Make sure the data is sent!
             if (outgoing.checkError()) {
                throw new IOException("Error occurred while transmitting message.");
             }
          }
       }
       catch (Exception e) {
          System.out.println("Sorry, an error has occurred.  Connection lost.");
          System.out.println("Error:  " + e);
          System.exit(1);
       }

    }  // end main()

} //end class CLChatServer
```

This program is a little more robust than *DateServer*. For one thing, it uses a ***handshake*** to make sure that a client who is trying to connect is really a *CLChatClient* program. A handshake is simply information sent between client and server as part of setting up the connection, before any actual data is sent. In this case, each side of the connection sends a string to the other side to identify itself. The handshake is part of the ***protocol*** that I made up for communication between *CLChatClient* and *CLChatServer*. A protocol is a detailed specification of what data and messages can be exchanged over a connection, how they must be represented, and what order they can be sent in. When you design a client/server application, the design of the protocol is an important consideration. Another aspect of the `CLChat` protocol is that after the handshake, every line of text that is sent over the connection begins with a character that acts as a command. If the character is 0, the rest of the line is a message from one user to the other. If the character is 1, the line indicates that a user has entered the "quit" command, and the connection is to be shut down.

Remember that if you want to try out this program on a single computer, you can use two command-line windows. In one, give the command "`java CLChatServer`" to start the server. Then, in the other, use the command "`java CLChatClient localhost`" to connect to the server that is running on the same machine.

11.5 Network Programming and Threads

In the previous section, we looked at several examples of network programming. Those examples showed how to create network connections and communicate through them, but they didn't deal with one of the fundamental characteristics of network programming, the fact that network communication is fundamentally asynchronous. From the point of view of a program on one end of a network connection, messages can arrive from the other side of the connection at any time; the arrival of a message is an *event* that is not under the control of the program that is receiving the message. Certainly, it is possible to design a network communication protocol that proceeds in a synchronous, step-by-step process from beginning to end—but whenever the process gets to a point in the protocol where it needs to read a message from the other side of the connection, it has to *wait* for that message to arrive. Essentially, the process has to wait

for a message-arrival event to occur before it can proceed. While it is waiting for the message, we say that the process is ***blocked***.

Perhaps an event-oriented networking API would be a good approach to dealing with the asynchronous nature of network communication, but that is not the approach that is taken in Java (or, typically, in other languages). Instead, a serious network program in Java uses **threads**. Threads were introduced in Section 8.5. A thread is a separate computational process that can run in parallel with other threads. When a program uses threads to do network communication, it is possible that some threads will be blocked, waiting for incoming messages, but other threads will still be able to continue performing useful work.

11.5.1 A Threaded GUI Chat Program.

The command-line chat programs, *CLChatClient.java* and *CLChatServer.java*, from the previous section use a straight-through, step-by-step protocol for communication. After a user on one side of a connection enters a message, the user must wait for a reply from the other side of the connection. An asynchronous chat program would be much nicer. In such a program, a user could just keep typing lines and sending messages without waiting for any response. Messages that arrive—asynchronously—from the other side would be displayed as soon as they arrive. It's not easy to do this in a command-line interface, but it's a natural application for a graphical user interface. The basic idea for a GUI chat program is to create a thread whose job is to read messages that arrive from the other side of the connection. As soon as the message arrives, it is displayed to the user; then, the message-reading thread blocks until the next incoming message arrives. While it is blocked, however, other threads can continue to run. In particular, the GUI event-handling thread that responds to user actions keeps running; that thread can send outgoing messages as soon as the user generates them.

The sample program *GUIChat.java* is an example of this. `GUIChat` is a two-way network chat program that allows two users to send messages to each other over the network. In this chat program, each user can send messages at any time, and incoming messages are displayed as soon as they are received.

The `GUIChat` program can act as both the client end and the server end of a connection. When `GUIChat` is started, a window appears on the screen. This window has a "Listen" button that the user can click to create a server socket that will listen for an incoming connection request; this makes the program act as a server. It also has a "Connect" button that the the user can click to send a connection request; this makes the program act as a client. As usual, the server listens on a specified port number. The client needs to know the computer on which the server is running and the port on which the server is listening. There are input boxes in the `GUIChat` window where the user can enter this information. Once a connection has been established between two `GUIChat` windows, each user can send messages to the other. The window has an input box where the user types the message. Pressing return while typing in this box sends the message. This means that the sending of the message is handled by the usual event-handling thread, in response to an event generated by a user action. Messages are received by a separate thread that just sits around waiting for incoming messages. This thread blocks while waiting for a message to arrive; when a message does arrive, it displays that message to the user. The window contains a large transcript area that displays both incoming and outgoing messages, along with other information about the network connection.

I urge you to compile the source code, *GUIChat.java*, and try the program. To make it easy to try it on a single computer, you can make a connection between one window and another window on the same computer, using "localhost" or "127.0.0.1" as the name of the computer.

(Once you have one `GUIChat` window open, you can open a second one by clicking the "New" button.) I also urge you to read the source code. I will discuss only parts of it here.

The program uses a nested class, *ConnectionHandler*, to handle most network-related tasks. *ConnectionHandler* is a subclass of *Thread*. The *ConnectionHandler* thread is responsible for opening the network connection and then for reading incoming messages once the connection has been opened. (By putting the connection-opening code in a separate thread, we make sure that the GUI is not blocked while the connection is being opened. Like reading incoming messages, opening a connection is a blocking operation that can take some time to complete.) A *ConnectionHandler* is created when the user clicks the "Listen" or "Connect" button. The "Listen" button should make the thread act as a server, while "Connect" should make it act as a client. To distinguish these two cases, the *ConnectionHandler* class has two constructors:

```
/**
 * Listen for a connection on a specified port.  The constructor
 * does not perform any network operations; it just sets some
 * instance variables and starts the thread.  Note that the
 * thread will only listen for one connection, and then will
 * close its server socket.
 */
ConnectionHandler(int port) {
   state = ConnectionState.LISTENING;
   this.port = port;
   postMessage("\nLISTENING ON PORT " + port + "\n");
   start();
}

/**
 * Open a connection to specified computer and port.  The constructor
 * does not perform any network operations; it just sets some
 * instance variables and starts the thread.
 */
ConnectionHandler(String remoteHost, int port) {
   state = ConnectionState.CONNECTING;
   this.remoteHost = remoteHost;
   this.port = port;
   postMessage("\nCONNECTING TO " + remoteHost + " ON PORT " + port + "\n");
   start();
}
```

Here, `state` is an instance variable whose type is defined by an enumerated type

```
enum ConnectionState { LISTENING, CONNECTING, CONNECTED, CLOSED };
```

The values of this `enum` represent different possible states of the network connection. It is often useful to treat a network connection as a state machine (see Subsection 6.5.4), since the response to various events can depend on the state of the connection when the event occurs. Setting the `state` variable to `LISTENING` or `CONNECTING` tells the thread whether it should act as a server or as a client. Note that the `postMessage()` method posts a message to the transcript area of the window, where it will be visible to the user.

Once the thread has been started, it executes the following `run()` method:

```
/**
 * The run() method that is executed by the thread.  It opens a
 * connection as a client or as a server (depending on which
```

```
     * constructor was used).
     */
    public void run() {
       try {
          if (state == ConnectionState.LISTENING) {
                // Open a connection as a server.
             listener = new ServerSocket(port);
             socket = listener.accept();
             listener.close();
          }
          else if (state == ConnectionState.CONNECTING) {
                // Open a connection as a client.
             socket = new Socket(remoteHost,port);
          }
          connectionOpened();  // Sets up to use the connection (including
                               // creating a BufferedReader, in, for reading
                               // incoming messages).
          while (state == ConnectionState.CONNECTED) {
                // Read one line of text from the other side of
                // the connection, and report it to the user.
             String input = in.readLine();
             if (input == null)
                connectionClosedFromOtherSide();
             else
                received(input);  // Report message to user.
          }
       }
       catch (Exception e) {
             // An error occurred.  Report it to the user, but not
             // if the connection has been closed (since the error
             // might be the expected error that is generated when
             // a socket is closed).
          if (state != ConnectionState.CLOSED)
             postMessage("\n\n ERROR:  " + e);
       }
       finally {  // Clean up before terminating the thread.
          cleanUp();
       }
    }
```

This method calls several other methods to do some of its work, but you can see the general outline of how it works. After opening the connection as either a server or client, the `run()` method enters a `while` loop in which it receives and processes messages from the other side of the connection until the connection is closed. It is important to understand how the connection can be closed. The `GUIChat` window has a "Disconnect" button that the user can click to close the connection. The program responds to this event by closing the socket that represents the connection. It is likely that when this happens, the connection-handling thread is blocked in the `in.readLine()` method, waiting for an incoming message. When the socket is closed by another thread, this method will fail and will throw an exception; this exception causes the thread to terminate. (If the connection-handling thread happens to be between calls to `in.readLine()` when the socket is closed, the `while` loop will terminate because the connection state changes

from `CONNECTED` to `CLOSED`.) Note that closing the window will also close the connection in the same way.

It is also possible for the user on the other side of the connection to close the connection. When that happens, the stream of incoming messages ends, and the `in.readLine()` on this side of the connection returns the value `null`, which indicates end-of-stream and acts as a signal that the connection has been closed by the remote user.

For a final look into the `GUIChat` code, consider the methods that send and receive messages. These methods are called from different threads. The `send()` method is called by the event-handling thread in response to a user action. Its purpose is to transmit a message to the remote user. It uses a *PrintWriter*, `out`, that writes to the socket's output stream. Synchronization of this method prevents the connection state from changing in the middle of the send operation:

```
/**
 * Send a message to the other side of the connection, and post the
 * message to the transcript.  This should only be called when the
 * connection state is ConnectionState.CONNECTED; if it is called at
 * other times, it is ignored.
 */
synchronized void send(String message) {
   if (state == ConnectionState.CONNECTED) {
      postMessage("SEND:  " + message);
      out.println(message);
      out.flush();
      if (out.checkError()) {
         postMessage("\nERROR OCCURRED WHILE TRYING TO SEND DATA.");
         close();  // Closes the connection.
      }
   }
}
```

The `received()` method is called by the connection-handling thread **after** a message has been read from the remote user. Its only job is to display the message to the user, but again it is synchronized to avoid the race condition that could occur if the connection state were changed by another thread while this method is being executed:

```
/**
 * This is called by the run() method when a message is received from
 * the other side of the connection.  The message is posted to the
 * transcript, but only if the connection state is CONNECTED.  (This
 * is because a message might be received after the user has clicked
 * the "Disconnect" button; that message should not be seen by the
 * user.)
 */
synchronized private void received(String message) {
   if (state == ConnectionState.CONNECTED)
      postMessage("RECEIVE:  " + message);
}
```

11.5.2 A Multithreaded Server

There is still one big problem with the `GUIChat` program. In order to open a connection to another computer, a user must know that there is a `GUIChat` program listening on some

particular port on some particular computer. Except in rather contrived situations, there is no way for a user to know that. It would be nice if it were possible to discover, somehow, who's out there on the Internet waiting for a connection. Unfortunately, this is not possible. And yet, applications such as AOL Instant Messenger seem to do just that—they can show you a list of users who are available to receive messages. How can they do that?

I don't know the details of instant messenger protocols, but it has to work something like this: When you start the client program, that program contacts a server program that runs constantly on some particular computer and on some particular port. Since the server is always available at the same computer and port, the information needed to contact it can be built into the client program or otherwise made available to the users of the program. The purpose of the server is to keep a list of available users. When your client program contacts the server, it gets a list of available users, along with whatever information is necessary to send messages to those users. At the same time, your client program registers you with the server, so that the server can tell other users that you are on-line. When you shut down the client program, you are removed from the server's list of users, and other users can be informed that you have gone off-line.

Of course, in an application like AOL server, you only get to see a list of available users from your "buddy list," a list of your friends who are also AOL users. To implement this, you need to have an account on the AOL server. The server needs to keep a database of information about all user accounts, including the buddy list for each user. This makes the server program rather complicated, and I won't consider that aspect of its functionality here. However, it is not very difficult to write a scaled-down application that uses the network in a similar way. I call my scaled-down version "`BuddyChat`." It doesn't keep separate buddy lists for each user; it assumes that you're willing to be buddies with anyone who happens to connect to the server. In this application, the server keeps a list of connected users and makes that list available to each connected user. A user can connect to another user and chat with that user, using a window that is very similar to the chat window in `GUIChat`. `BuddyChat` is still just a toy, compared to serious network applications, but it does illustrate some core ideas.

The `BuddyChat` application comes in several pieces. *BuddyChatServer.java* is the server program, which keeps the list of available users and makes that list available to clients. Ideally, the server program would run constantly (as a daemon) on a computer and port that are known to all the possible client users. For testing, of course, it can simply be stated like any other program. The client program is *BuddyChat.java*. This program is to be run by any user who wants to use the `BuddyChat` service. When a user starts the client program, it connects to the server, and it gets from the server a list of other users who are currently connected. The list is displayed to the user of the client program, who can send a request for a chat connection with any user on the list. The client can also receive incoming chat connection requests from other users. The window that is used for chatting is defined by *BuddyChatWindow.java*, which is not itself a program but just a subclass of *JFrame* that defines the chat window. (There is also a fourth piece, *BuddyChatServerShutdown.java*. This is a program that can be run to shut down the `BuddyChatServer` gracefully. I will not discuss it further here. See the source code for more information, if you are interested.)

I urge you to compile the programs and try them out. For testing, you can try them on a single computer (although all the windows can get a little confusing). First, start `BuddyChatServer`. The server has no GUI interface, but it does print some information to standard output as it runs. Then start the `BuddyChat` client program. When `BuddyChat` starts up, it presents a window where you can enter the name and port number for the server and your

"handle," which is just a name that will identify you in the server's list of users. The server info is already set up to connect to a server on the same machine. When you hit the "Connect" button, a new window will open with a list, currently empty, of other users connected to the server. Now, start another copy of the `BuddyChat` client program. When you click "Connect", you'll have two client list windows, one for each copy of the client program that you've started. (One of these windows will be exactly on top of the other, so you'll have to move it to see the second window.) Each client window will display the other client in its list of users. You can run additional copies of the client program, if you want, and you might want to try connecting from another computer if one is available.

At this point, there is a network connection in place between the server and each client. Whenever a client connects to or disconnects from the server, the server sends a notification of the event to each connected client, so that the client can modify its own list of connected users. The server also maintains a listening socket that listens for connection requests from new clients. In order to manage all this, the server is running several threads. One thread waits for connection requests on the listening socket. In addition to this, there are two threads for each connected client—one thread for sending messages to the client and one thread for reading messages sent by the client to the server.

Back to trying out the program. Remember that the whole point was to provide each user with a list of potential chat partners. Click on a user in one of the client user lists, and then click the "Connect to Selected Buddy" button. When you do this, your *BuddyChat* program sends a connection request to the *BuddyChat* program that is being run by the selected user. Each *BuddyChat* program, one on each side of the connection, opens a chat window (of type *BuddyChatWindow*). A network connection between these two windows is set up without any further action on the part of the two users, and the users can use the windows to send messages back and forth to each other. The *BuddyChatServer* program has nothing to do with opening, closing, or using the connection between its two clients (although a different design might have had the messages go through the server).

In order to open the chat connection from one program to another, the second program must be listening for connection requests and the first program must know the computer and port on which the first user is listening. In the `BuddyChat` system, the *BuddyChatServer* knows this information and provides it to each *BuddyChat* client program. The **users** of the client programs never have to be aware of this information.

How does the server know about the clients' computers and port numbers? When a *BuddyChat* client program is run, in addition to opening a connection to the *BuddyChatServer*, the client also creates a listening socket to accept connection requests from other users. When the client registers with the server, it tells the server the port number of the client's listening socket. The server also knows the IP address of the computer on which the client is running, since it has a network connection to that computer. This means that the *BuddyChatServer* knows the IP address and listening socket port number of every *BuddyChat* client. A copy of this information is provided (along with the users' handles) to each connected client program. The net result is that every *BuddyChat* client program has the information that it needs to contact all the other clients.

The basic techniques used in the `BuddyChat` system are the same as those used in previous networking examples: server sockets, client sockets, input and output streams for sending messages over the network, and threads to handle the communication. The important difference is how these basic building blocks are combined to build a more complex application. I have tried to explain the logic of that application here. I will not discuss the `BuddyChat` source code

here, since it is locally similar to examples that we have already looked at, but I encourage you to study the source code if you are interested in network programming.

`BuddyChat` seems to have a lot of functionality, yet I said it was still a "toy" program. What exactly makes it a toy? There are at least two big problems. First of all, it is not ***scalable***. A network program is scalable if it will work well for a large number of simultaneous users. `BuddyChat` would have problems with a large number of users because it uses so many threads (two for each user). It takes a certain amount of processing for a computer to switch its attention from one thread to another. On a very busy server, the constant switching between threads would soon start to degrade the performance. One solution to this is to use a more advanced network API. Java has a class *SelectableChannel* that makes it possible for one thread to manage communication over a large number of network connections. This class is part of the package `java.nio` that provides a number of advanced I/O capabilities for working with files and networking. However, I will not cover those capabilities in this book.

But the biggest problem is that `BuddyChat` offers absolutely no defense against ***denial of service*** attacks. In a denial of service, a malicious user attacks a network server in some way that prevents other users from accessing the service or severely degrades the performance of the service for those users. It would be simple to launch a denial of service attack on `BuddyChat` by making a huge number of connections to the server. The server would then spend most of its time servicing those bogus connections. The server could guard against this to some extent by putting a limit on the number of simultaneous connections that it will accept from a given IP address. It would also be helpful to add some security to the server by requiring users to know a password in order to connect. However, neither of these measures would fully solve the problem, and it is very difficult to find a complete defense against denial of service attacks.

11.5.3 Distributed Computing

In Section 8.5, we saw how threads can be used to do parallel processing, where a number of processors work together to complete some task. In that section, it was assumed that all the processors were inside one multi-processor computer. But parallel processing can also be done using processors that are in different computers, as long as those computers are connected to a network over which they can communicate. This type of parallel processing—in which a number of computers work together on a task and communicate over a network—is called ***distributed computing***.

In some sense, the whole Internet is an immense distributed computation, but here I am interested in how computers on a network can cooperate to solve some computational problem. There are several approaches to distributed computing that are supported in Java. ***RMI*** and ***CORBA*** are standards that enable a program running on one computer to call methods in objects that exist on other computers. This makes it possible to design an object-oriented program in which different parts of the program are executed on different computers. RMI (Remote Method Invocation) only supports communication between Java objects. CORBA (Common Object Request Broker Architecture) is a more general standard that allows objects written in various programming languages, including Java, to communicate with each other. As is commonly the case in networking, there is the problem of locating services (where in this case, a "service" means an object that is available to be called over the network). That is, how can one computer know which computer a service is located on and what port it is listening on? RMI and CORBA solve this problem using something like our little `BuddyChatServer` example—a server running at a known location keeps a list of services that are available on other computers. Computers that offer services register those services with the server; computers that

need services contact the server to find out where they are located.

RMI and CORBA are complex systems that are not very easy to use. I mention them here because they are part of Java's standard network API, but I will not discuss them further. Instead, we will look at a relatively simple demonstration of distributed computing that uses only basic networking.

The problem that we will look at uses the simplest type of parallel programming, in which the problem can be broken down into tasks that can be performed independently, with no communication between the tasks. To apply distributed computing to this type of problem, we can use one "master" program that divides the problem into tasks and sends those tasks over the network to "worker" programs that do the actual work. The worker programs send their results back to the master program, which combines the results from all the tasks into a solution of the overall problem. In this context, the worker programs are often called "slaves," and the program uses the so-called ***master/slave*** approach to distributed computing.

The demonstration program is defined by three source code files: *CLMandelbrotMaster.java* defines the master program; *CLMandelbrotWorker.java* defines the worker programs; and *CLMandelbrotTask.java* defines the class, *CLMandelbrotTask*, that represents an individual task that is performed by the workers. To run the demonstration, you must start the `CLMandelbrotWorker` program on several computers (probably by running it on the command line). This program uses *CLMandelbrotTask*, so both class files, `CLMandelbrotWorker.class` and `CLMandelbrotTask.class`, must be present on the worker computers. You can then run `CLMandelbrotMaster` on the master computer. Note that this program also requires the class *CLMandelbrotTask*. You must specify the host name or IP address of each of the worker computers as command line arguments for `CLMandelbrotMaster`. A worker program listens for connection requests from the master program, and the master program must be told where to send those requests. For example, if the worker program is running on three computers with IP addresses 172.30.217.101, 172.30.217.102, and 172.30.217.103, then you can run `CLMandelbrotMaster` with the command

```
java  CLMandelbrotMaster  172.30.217.101  172.30.217.102  172.30.217.103
```

The master will make a network connection to the worker at each IP address; these connections will be used for communication between the master program and the workers.

It is possible to run several copies of `CLMandelbrotWorker` on the same computer, but they must listen for network connections on different ports. It is also possible to run `CLMandelbrotWorker` on the same computer as `CLMandelbrotMaster`. You might even see some speed-up when you do this, if your computer has several processors. See the comments in the program source code files for more information, but here are some commands that you can use to run the master program and two copies of the worker program on the same computer. Give these commands in separate command windows:

```
java  CLMandelbrotWorker                                (Listens on default port)

java  CLMandelbrotWorker  1501                          (Listens on port 1501)

java  CLMandelbrotMaster  localhost  localhost:1501
```

Every time `CLMandelbrotMaster` is run, it solves exactly the same problem. (For this demonstration, the nature of the problem is not important, but the problem is to compute the data needed for a picture of a small piece of the famous "Mandelbrot Set." If you are interested in seeing the picture that is produced, uncomment the call to the `saveImage()` method at the

end of the `main()` routine in *CLMandelbrotMaster.java*. We will encounter the Mandelbrot Set again as an example in Chapter 12.)

You can run `CLMandelbrotMaster` with different numbers of worker programs to see how the time required to solve the problem depends on the number of workers. (Note that the worker programs continue to run after the master program exists, so you can run the master program several times without having to restart the workers.) In addition, if you run `CLMandelbrotMaster` with no command line arguments, it will solve the entire problem on its own, so you can see how long it takes to do so without using distributed computing. In a trial that I ran, it took 40 seconds for `CLMandelbrotMaster` to solve the problem on its own. Using just one worker, it took 43 seconds. The extra time represents extra work involved in using the network; it takes time to set up a network connection and to send messages over the network. Using two workers (on different computers), the problem was solved in 22 seconds. In this case, each worker did about half of the work, and their computations were performed in parallel, so that the job was done in about half the time. With larger numbers of workers, the time continued to decrease, but only up to a point. The master program itself has a certain amount of work to do, no matter how many workers there are, and the total time to solve the problem can never be less than the time it takes for the master program to do its part. In this case, the minimum time seemed to be about five seconds.

* * *

Let's take a look at how this distributed application is programmed. The master program divides the overall problem into a set of tasks. Each task is represented by an object of type *CLMandelbrotTask*. These tasks have to be communicated to the worker programs, and the worker programs must send back their results. Some protocol is needed for this communication. I decided to use character streams. The master encodes a task as a line of text, which is sent to a worker. The worker decodes the text (into an object of type *CLMandelbrotTask*) to find out what task it is supposed to perform. It performs the assigned task. It encodes the results as another line of text, which it sends back to the master program. Finally, the master decodes the results and combines them with the results from other tasks. After all the tasks have been completed and their results have been combined, the problem has been solved.

The problem is divided into a fairly large number of tasks. A worker receives not just one task, but a sequence of tasks. Each time it finishes a task and sends back the result, it is assigned a new task. After all tasks are complete, the worker receives a "close" command that tells it to close the connection. In *CLMandelbrotWorker.java*, all this is done in a method named `handleConnection()` that is called to handle a connection that has already been opened to the master program. It uses a method `readTask()` to decode a task that it receives from the master and a method `writeResults()` to encode the results of the task for transmission back to the master. It must also handle any errors that occur:

```
private static void handleConnection(Socket connection) {
   try {
      BufferedReader in = new BufferedReader( new InputStreamReader(
                                          connection.getInputStream()) );
      PrintWriter out = new PrintWriter(connection.getOutputStream());
      while (true) {
         String line = in.readLine();  // Message from the master.
         if (line == null) {
               // End-of-stream encountered -- should not happen.
            throw new Exception("Connection closed unexpectedly.");
         }
```

```
            if (line.startsWith(CLOSE_CONNECTION_COMMAND)) {
                  // Represents the normal termination of the connection.
               System.out.println("Received close command.");
               break;
            }
            else if (line.startsWith(TASK_COMMAND)) {
                  // Represents a CLMandelbrotTask that this worker is
                  // supposed to perform.
               CLMandelbrotTask task = readTask(line);  // Decode the message.
               task.compute();  // Peform the task.
               out.println(writeResults(task));  //  Send back the results.
               out.flush();
            }
            else {
                  // No other messages are part of the protocol.
               throw new Exception("Illegal command received.");
            }
         }
      }
      catch (Exception e) {
         System.out.println("Client connection closed with error " + e);
      }
      finally {
         try {
            connection.close();  // Make sure the socket is closed.
         }
         catch (Exception e) {
         }
      }
   }
```

Note that this method is **not** executed in a separate thread. The worker has only one thing to do at a time and does not need to be multithreaded.

You might wonder why so many tasks are used. Why not just divide the problem into one task for each worker? The reason is that using a larger number of tasks makes it possible to do ***load balancing***. Not all tasks take the same amount of time to execute. This is true for many reasons. Some of the tasks might simply be more computationally complex than others. Some of the worker computers might be slower than others. Or some worker computers might be busy running other programs, so that they can only give part of their processing power to the worker program. If we assigned one task per worker, it is possible that a complex task running on a slow, busy computer would take much longer than the other tasks to complete. This would leave the other workers idle and delay the completion of the job while that worker completes its task. To complete the job as quickly as possible, we want to keep all the workers busy and have them all finish at about the same time. This is called load balancing. If we have a large number of tasks, the load will automatically be approximately balanced: A worker is not assigned a new task until it finishes the task that it is working on. A slow worker, or one that happens to receive more complex tasks, will complete fewer tasks than other workers, but all workers will be kept busy until close to the end of the job. On the other hand, individual tasks shouldn't be too small. Network communication takes some time. If it takes longer to transmit a task and its results than it does to perform the task, then using distributed computing will take **more**

time than simply doing the whole job on one computer! A problem is a good candidate for distributed computing if it can be divided into a fairly large number of fairly large tasks.

Turning to the master program, *CLMandelbrotMaster.java*, we encounter a more complex situation. The master program must communicate with several workers over several network connections. To accomplish this, the master program is multi-threaded, with one thread to manage communication with each worker. A pseudocode outline of the `main()` routine is quite simple:

```
create a list of all tasks that must be performed
if there are no command line arguments {
      // The master program does all the tasks itself.
   Perform each task.
}
else {
      // The tasks will be performed by worker programs.
   for each command line argument:
      Get information about a worker from command line argument.
      Create and start a thread to communicate with the worker.
   Wait for all threads to terminate.
}
// All tasks are now complete (assuming no error occurred).
```

The list of tasks is stored in a variable, `tasks`, of type *ArrayList<CLMandelbrotTask>*. The communication threads take tasks from this list and send them to worker programs. The method `getNextTask()` gets one task from the list. If the list is empty, it returns `null` as a signal that all tasks have been assigned and the communication thread can terminate. Since `tasks` is a resource that is shared by several threads, access to it must be controlled; this is accomplished by writing `getNextTask()` as a `synchronized` method:

```
synchronized private static CLMandelbrotTask getNextTask() {
   if (tasks.size() == 0)
      return null;
   else
      return tasks.remove(0);
}
```

(The reason for the synchronization is to avoid the race condition that could occur between the time that the value of `tasks.size()` is tested and the time that `tasks.remove()` is called. See Subsection 8.5.3 for information about parallel programming, race conditions, and `synchronized`.)

The job of a thread is to send a sequence of tasks to a worker thread and to receive the results that the worker sends back. The thread is also responsible for opening the connection in the first place. A pseudocode outline for the process executed by the thread might look like:

```
Create a socket connected to the worker program.
Create input and output streams for communicating with the worker.
while (true) {
   Let task = getNextTask().
   If task == null
      break;  // All tasks have been assigned.
   Encode the task into a message and transmit it to the worker.
   Read the response from the worker.
   Decode and process the response.
}
```

```
Send a "close" command to the worker.
Close the socket.
```

This would work OK. However, there are a few subtle points. First of all, the thread must be ready to deal with a network error. For example, a worker might shut down unexpectedly. But if that happens, the master program can continue, provided other workers are still available. (You can try this when you run the program: Stop one of the worker programs, with `CONTROL-C`, and observe that the master program still completes successfully.) A difficulty arises if an error occurs while the thread is working on a task: If the problem as a whole is going to be completed, that task will have to be reassigned to another worker. I take care of this by putting the uncompleted task back into the task list. (Unfortunately, my program does not handle all possible errors. If a network connection "hangs" indefinitely without actually generating an error, my program will also hang, waiting for a response from a worker that will never arrive. A more robust program would have some way of detecting the problem and reassigning the task.)

Another defect in the procedure outlined above is that it leaves the worker program idle while the thread is processing the worker's response. It would be nice to get a new task to the worker before processing the response from the previous task. This would keep the worker busy and allow two operations to proceed simultaneously instead of sequentially. (In this example, the time it takes to process a response is so short that keeping the worker waiting while it is done probably makes no significant difference. But as a general principle, it's desirable to have as much parallelism as possible in the algorithm.) We can modify the procedure to take this into account:

```
try {
   Create a socket connected to the worker program.
   Create input and output streams for communicating with the worker.
   Let currentTask = getNextTask().
   Encode currentTask into a message and send it to the worker.
   while (true) {
      Read the response from the worker.
      Let nextTask = getNextTask().
      If nextTask != null {
            // Send nextTask to the worker before processing the
            // response to currentTask.
         Encode nextTask into a message and send it to the worker.
      }
      Decode and process the response to currentTask.
      currentTask = nextTask.
      if (currentTask == null)
         break; // All tasks have been assigned.
   }
   Send a "close" command to the worker.
   Close the socket.
}
catch (Exception e) {
   Put uncompleted task, if any, back into the task list.
}
finally {
   Close the connection.
}
```

Finally, here is how this translates into Java. The pseudocode presented above becomes the `run()` method in the class that defines the communication threads used by the master program:

```
/**
 * This class represents one worker thread.  The job of a worker thread
 * is to send out tasks to a CLMandelbrotWorker program over a network
 * connection, and to get back the results computed by that program.
 */
private static class WorkerConnection extends Thread {

   int id;        // Identifies this thread in output statements.
   String host;   // The host to which this thread will connect.
   int port;      // The port number to which this thread will connect.

   /**
    * The constructor just sets the values of the instance
    * variables id, host, and port and starts the thread.
    */
   WorkerConnection(int id, String host, int port) {
      this.id = id;
      this.host = host;
      this.port = port;
      start();
   }

   /**
    * The run() method of the thread opens a connection to the host and
    * port specified in the constructor, then sends tasks to the
    * CLMandelbrotWorker program on the other side of that connection.
    * If the thread terminates normally, it outputs  the number of tasks
    * that it processed.  If it terminates with an error, it outputs
    * an error message.
    */
   public void run() {

      int tasksCompleted = 0; // How many tasks has this thread handled.
      Socket socket;  // The socket for the connection.

      try {
         socket = new Socket(host,port);  // open the connection.
      }
      catch (Exception e) {
         System.out.println("Thread " + id + " could not open connection to " +
               host + ":" + port);
         System.out.println("   Error: " + e);
         return;
      }

      CLMandelbrotTask currentTask = null;
      CLMandelbrotTask nextTask = null;

      try {
         PrintWriter out = new PrintWriter(socket.getOutputStream());
         BufferedReader in = new BufferedReader(
                              new InputStreamReader(socket.getInputStream()) );
         currentTask = getNextTask();
         if (currentTask != null) {
               // Send first task to the worker program.
            String taskString = writeTask(currentTask);
```

```
                out.println(taskString);
                out.flush();
             }
             while (currentTask != null) {
                String resultString = in.readLine(); // Get results for currentTask.
                if (resultString == null)
                   throw new IOException("Connection closed unexpectedly.");
                if (! resultString.startsWith(RESULT_COMMAND))
                   throw new IOException("Illegal string received from worker.");
                nextTask = getNextTask();  // Get next task and send it to worker.
                if (nextTask != null) {
                      // Send nextTask to worker before processing results for
                      // currentTask, so that the worker can work on nextTask
                      // while the currentTask results are processed.
                   String taskString = writeTask(nextTask);
                   out.println(taskString);
                   out.flush();
                }
                readResults(resultString, currentTask);
                finishTask(currentTask);  // Process results from currentTask.
                tasksCompleted++;
                currentTask = nextTask;   // We are finished with old currentTask.
                nextTask = null;
             }
             out.println(CLOSE_CONNECTION_COMMAND);  // Send close command to worker.
             out.flush();
          }
          catch (Exception e) {
             System.out.println("Thread " + id + " terminated because of an error");
             System.out.println("   Error: " + e);
             e.printStackTrace();
                // Put uncompleted task, if any, back into the task list.
             if (currentTask != null)
                reassignTask(currentTask);
             if (nextTask != null)
                reassignTask(nextTask);
          }
          finally {
             System.out.println("Thread " + id + " ending after completing " +
                   tasksCompleted + " tasks");
             try {
                socket.close();
             }
             catch (Exception e) {
             }
          }

       } //end run()

    } // end nested class WorkerConnection
```

11.6 A Brief Introduction to XML

When data is saved to a file or transmitted over a network, it must be represented in some way that will allow the same data to be rebuilt later, when the file is read or the transmission is received. We have seen that there are good reasons to prefer textual, character-based representations in many cases, but there are many ways to represent a given collection of data as text. In this section, we'll take a brief look at one type of character-based data representation that has become increasingly common.

XML (eXtensible Markup Language) is a syntax for creating data representation languages. There are two aspects or levels of XML. On the first level, XML specifies a strict but relatively simple syntax. Any sequence of characters that follows that syntax is a ***well-formed*** XML document. On the second level, XML provides a way of placing further restrictions on what can appear in a document. This is done by associating a ***DTD*** (Document Type Definition) with an XML document. A DTD is essentially a list of things that are allowed to appear in the XML document. A well-formed XML document that has an associated DTD and that follows the rules of the DTD is said to be a ***valid*** XML document. The idea is that XML is a general format for data representation, and a DTD specifies how to use XML to represent a particular kind of data. (There is also an alternative to DTDs, known as ***XML schemas***, for defining valid XLM documents, but let's ignore them here.)

There is nothing magical about XML. It's certainly not perfect. It's a very verbose language, and some people think it's ugly. On the other hand it's very flexible; it can be used to represent almost any type of data. It was built from the start to support all languages and alphabets. Most important, it has become an accepted standard. There is support in just about any programming language for processing XML documents. There are standard DTDs for describing many different kinds of data. There are many ways to design a data representation language, but XML is the one that has happened to come into widespread use. In fact, it has found its way into almost every corner of information technology. For example: There are XML languages for representing mathematical expressions (MathML), musical notation (MusicXML), molecules and chemical reactions (CML), vector graphics (SVG), and many other kinds of information. XML is used by OpenOffice and recent versions of Microsoft Office in the document format for office applications such as word processing, spreadsheets, and presentations. XML site syndication languages (RSS, ATOM) make it possible for web sites, newspapers, and blogs to make a list of recent headlines available in a standard format that can be used by other web sites and by web browsers; the same format is used to publish podcasts. And XML is a common format for the electronic exchange of business information.

My purpose here is not to tell you everything there is to know about XML. I will just explain a few ways in which it can be used in your own programs. In particular, I will not say anything further about DTDs and valid XML. For many purposes, it is sufficient to use well-formed XML documents with no associated DTDs.

11.6.1 Basic XML Syntax

An XML document looks a lot like an HTML document (see Subsection 6.2.3). HTML is not itself an XML language, since it does not follow all the strict XML syntax rules, but the basic ideas are similar. Here is a short, well-formed XML document:

```
<?xml version="1.0"?>
<simplepaint version="1.0">
   <background red='255' green='153' blue='51'/>
```

```
   <curve>
      <color red='0' green='0' blue='255'/>
      <symmetric>false</symmetric>
      <point x='83' y='96'/>
      <point x='116' y='149'/>
      <point x='159' y='215'/>
      <point x='216' y='294'/>
      <point x='264' y='359'/>
      <point x='309' y='418'/>
      <point x='371' y='499'/>
      <point x='400' y='543'/>
   </curve>
   <curve>
      <color red='255' green='255' blue='255'/>
      <symmetric>true</symmetric>
      <point x='54' y='305'/>
      <point x='79' y='289'/>
      <point x='128' y='262'/>
      <point x='190' y='236'/>
      <point x='253' y='209'/>
      <point x='341' y='158'/>
   </curve>
</simplepaint>
```

The first line, which is optional, merely identifies this as an XML document. This line can also specify other information, such as the character encoding that was used to encode the characters in the document into binary form. If this document had an associated DTD, it would be specified in a "DOCTYPE" directive on the next line of the file.

Aside from the first line, the document is made up of ***elements***, ***attributes***, and textual content. An element starts with a ***tag***, such as `<curve>` and ends with a matching ***end-tag*** such as `</curve>`. Between the tag and end-tag is the ***content*** of the element, which can consist of text and nested elements. (In the example, the only textual content is the `true` or `false` in the `<symmetric>` elements.) If an element has no content, then the opening tag and end-tag can be combined into a single ***empty tag***, such as `<point x='83' y='96'/>`, which is an abbreviation for `<point x='83' y='96'></point>`. A tag can include attributes such as the `x` and `y` in `<point x='83' y='96'/>` or the `version` in `<simplepaint version="1.0">`. A document can also include a few other things, such as comments, that I will not discuss here.

The basic structure should look familiar to someone familiar with HTML. The most striking difference is that in XML, **you get to choose the tags**. Whereas HTML comes with a fixed, finite set of tags, with XML you can make up meaningful tag names that are appropriate to your application and that describe the data that is being represented. (For an XML document that uses a DTD, it's the author of the DTD who gets to choose the tag names.)

Every well-formed XML document follows a strict syntax. Here are some of the most important syntax rules: Tag names and attribute names in XML are case sensitive. A name must begin with a letter and can contain letters, digits and certain other characters. Spaces and ends-of-line are significant only in textual content. Every tag must either be an empty tag or have a matching end-tag. By "matching" here, I mean that elements must be properly nested; if a tag is inside some element, then the matching end-tag must also be inside that element. A document must have a ***root element***, which contains all the other elements. The root element in the above example has tag name `simplepaint`. Every attribute must have a value, and that value must be enclosed in quotation marks; either single quotes or double

quotes can be used for this. The special characters < and &, if they appear in attribute values or textual content, must be written as `<` and `&`. "`<`" and "`&`" are examples of ***entities***. The entities `>`, `&quot;`, and `&apos;` are also defined, representing >, double quote, and single quote. (Additional entities can be defined in a DTD.)

While this description will not enable you to understand everything that you might encounter in XML documents, it should allow you to design well-formed XML documents to represent data structures used in Java programs.

11.6.2 XMLEncoder and XMLDecoder

We will look at two approaches to representing data from Java programs in XML format. One approach is to design a custom XML language for the specific data structures that you want to represent. We will consider this approach in the next subsection. First, we'll look at an easy way to store data in XML files and to read those files back into a program. The technique uses the classes *XMLEncoder* and *XMLDecoder*. These classes are defined in the package `java.beans`. An *XMLEncoder* can be used to write objects to an *OutputStream* in XML form. An *XMLDecoder* can be used to read the output of an *XMLEncoder* and reconstruct the objects that were written by it. *XMLEncoder* and *XMLDecoder* have much the same functionality as *ObjectOutputStream* and *ObjectInputStream* and are used in much the same way. In fact, you don't even have to know anything about XML to use them. However, you do need to know a little about ***Java beans***.

A Java bean is just an object that has certain characteristics. The class that defines a Java bean must be a `public` class. It must have a constructor that takes no parameters. It should have a "get" method and a "set" method for each of its important instance variables. (See Subsection 5.1.3.) The last rule is a little vague. The idea is that is should be possible to inspect all aspects of the object's state by calling "get" methods, and it should be possible to set all aspects of the state by calling "set" methods. A bean is not required to implement any particular `interface`; it is recognized as a bean just by having the right characteristics. Usually, Java beans are passive data structures that are acted upon by other objects but don't do much themselves.

XMLEncoder and *XMLDecoder* can't be used with arbitrary objects; they can only be used with beans. When an *XMLEncoder* writes an object, it uses the "get" methods of that object to find out what information needs to be saved. When an *XMLDecoder* reconstructs an object, it creates the object using the constructor with no parameters and it uses "set" methods to restore the object's state to the values that were saved by the *XMLEncoder*. (Some standard java classes are processed using additional techniques. For example, a different constructor might be used, and other methods might be used to inspect and restore the state.)

For an example, we return to the same `SimplePaint` example that was used in Subsection 11.3.4. Suppose that we want to use *XMLEncoder* and *XMLDecoder* to create and read files in that program. Part of the data for a `SimplePaint` sketch is stored in objects of type *CurveData*, defined as:

```
private static class CurveData {
   Color color;  // The color of the curve.
   boolean symmetric;  // Are reflections also drawn?
   ArrayList<Point> points;  // The points on the curve.
}
```

To use such objects with *XMLEncoder* and *XMLDecoder*, we have to modify this class so that

it follows the Java bean pattern. The class has to be public, and we need get and set methods for each instance variable. This gives:

```
public static class CurveData {
   private Color color;  // The color of the curve.
   private boolean symmetric;  // Are reflections also drawn?
   private ArrayList<Point> points;  // The points on the curve.
   public Color getColor() {
      return color;
   }
   public void setColor(Color color) {
      this.color = color;
   }
   public ArrayList<Point> getPoints() {
      return points;
   }
   public void setPoints(ArrayList<Point> points) {
      this.points = points;
   }
   public boolean isSymmetric() {
      return symmetric;
   }
   public void setSymmetric(boolean symmetric) {
      this.symmetric = symmetric;
   }
}
```

I didn't really need to make the instance variables `private`, but bean properties are usually `private` and are accessed only through their get and set methods.

At this point, we might define another bean class, *SketchData*, to hold all the necessary data for representing the user's picture. If we did that, we could write the data to a file with a single output statement. In my program, however, I decided to write the data in several pieces.

An *XMLEncoder* can be constructed to write to any output stream. The output stream is specified in the encoder's constructor. For example, to create an encoder for writing to a file:

```
XMLEncoder encoder;
try {
   FileOutputStream stream = new FileOutputStream(selectedFile);
   encoder = new XMLEncoder( stream );
     .
     .
```

Once an encoder has been created, its `writeObject()` method is used to write objects, coded into XML form, to the stream. In the `SimplePaint` program, I save the background color, the number of curves in the picture, and the data for each curve. The curve data are stored in a list of type *ArrayList<CurveData>* named `curves`. So, a complete representation of the user's picture can be created with:

```
   encoder.writeObject(getBackground());
   encoder.writeObject(new Integer(curves.size()));
   for (CurveData c : curves)
      encoder.writeObject(c);
   encoder.close();
```

When reading the data back into the program, an *XMLDecoder* is created to read from an input file stream. The objects are then read, using the decoder's `readObject()` method, in the same order in which they were written. Since the return type of `readObject()` is *Object*, the returned values must be type-cast to their correct type:

```
Color bgColor = (Color)decoder.readObject();
Integer curveCt = (Integer)decoder.readObject();
ArrayList<CurveData> newCurves = new ArrayList<CurveData>();
for (int i = 0; i < curveCt; i++) {
   CurveData c = (CurveData)decoder.readObject();
   newCurves.add(c);
}
decoder.close();
curves = newCurves; // Replace the program's data with data from the file.
setBackground(bgColor);
repaint();
```

You can look at the sample program *SimplePaintWithXMLEncoder.java* to see this code in the context of a complete program. Files are created by the method `doSaveAsXML()` and are read by `doOpenAsXML()`.

The XML format used by *XMLEncoder* and *XMLDecoder* is more robust than the binary format used for object streams and is more appropriate for long-term storage of objects in files.

11.6.3 Working With the DOM

The output produced by an *XMLEncoder* tends to be long and not very easy for a human reader to understand. It would be nice to represent data in a more compact XML format that uses meaningful tag names to describe the data and makes more sense to human readers. We'll look at yet another version of `SimplePaint` that does just that. See *SimplePaintWithXML.java* for the source code. The sample XML document shown earlier in this section was produced by this program. I designed the format of that document to represent all the data needed to reconstruct a picture in `SimplePaint`. The document encodes the background color of the picture and a list of curves. Each `<curve>` element contains the data from one object of type *CurveData*.

It is easy enough to write data in a customized XML format, although we have to be very careful to follow all the syntax rules. Here is how I write the data for a `SimplePaint` picture to a *PrintWriter*, `out`:

```
out.println("<?xml version=\"1.0\"?>");
out.println("<simplepaint version=\"1.0\">");
Color bgColor = getBackground();
out.println("   <background red='" + bgColor.getRed() + "' green='" +
      bgColor.getGreen() + "' blue='" + bgColor.getBlue() + "'/>");
for (CurveData c : curves) {
   out.println("   <curve>");
   out.println("      <color red='" + c.color.getRed() + "' green='" +
         c.color.getGreen() + "' blue='" + c.color.getBlue() + "'/>");
   out.println("      <symmetric>" + c.symmetric + "</symmetric>");
   for (Point pt : c.points)
      out.println("      <point x='" + pt.x + "' y='" + pt.y + "'/>");
   out.println("   </curve>");
}
out.println("</simplepaint>");
```

Reading the data back into the program is another matter. To reconstruct the data structure represented by the XML Document, it is necessary to parse the document and extract the data from it. Fortunately, Java has a standard API for parsing and processing XML Documents. (Actually, it has two, but we will only look at one of them.)

A well-formed XML document has a certain structure, consisting of elements containing attributes, nested elements, and textual content. It's possible to build a data structure in the computer's memory that corresponds to the structure and content of the document. Of course, there are many ways to do this, but there is one common standard representation known as the ***Document Object Model***, or DOM. The DOM specifies how to build data structures to represent XML documents, and it specifies some standard methods for accessing the data in that structure. The data structure is a kind of tree whose structure mirrors the structure of the document. The tree is constructed from ***nodes*** of various types. There are nodes to represent elements, attributes, and text. (The tree can also contain several other types of node, representing aspects of XML that we can ignore here.) Attributes and text can be processed without directly manipulating the corresponding nodes, so we will be concerned almost entirely with element nodes.

The sample program *XMLDemo.java* lets you experiment with XML and the DOM. It has a text area where you can enter an XML document. Initially, the input area contains the sample XML document from this section. When you click a button named "Parse XML Input", the program will attempt to read the XML from the input box and build a DOM representation of that document. If the input is not legal XML, an error message is displayed. If it is legal, the program will traverse the DOM representation and display a list of elements, attributes, and textual content that it encounteres. (The program uses a few techniques that I won't discuss here.)

In Java, the DOM representation of an XML document file can be created with just two statements. If `selectedFile` is a variable of type *File* that represents the XML file, then

```
DocumentBuilder docReader
                = DocumentBuilderFactory.newInstance().newDocumentBuilder();
xmldoc = docReader.parse(selectedFile);
```

will open the file, read its contents, and build the DOM representation. The classes *DocumentBuilder* and *DocumentBuilderFactory* are both defined in the package `javax.xml.parsers`. The method `docReader.parse()` does the actual work. It will throw an exception if it can't read the file or if the file does not contain a legal XML document. If it succeeds, then the value returned by `docReader.parse()` is an object that represents the entire XML document. (This is a very complex task! It has been coded once and for all into a method that can be used very easily in any Java program. We see the benefit of using a standardized syntax.)

The structure of the DOM data structure is defined in the package `org.w3c.dom`, which contains several data types that represent an XML document as a whole and the individual nodes in a document. The "org.w3c" in the name refers to the World Wide Web Consortium, W3C, which is the standards organization for the Web. DOM, like XML, is a general standard, not just a Java standard. The data types that we need here are *Document*, *Node*, *Element*, and *NodeList*. (They are defined as `interfaces` rather than `classes`, but that fact is not relevant here.) We can use methods that are defined in these data types to access the data in the DOM representation of an XML document.

An object of type *Document* represents an entire XML document. The return value of `docReader.parse()`—`xmldoc` in the above example—is of type *Document*. We will only need one method from this class: If `xmldoc` is of type *Document*, then

```
xmldoc.getDocumentElement()
```

returns a value of type *Element* that represents the root element of the document. (Recall that this is the top-level element that contains all the other elements.) In the sample XML document from earlier in this section, the root element consists of the tag `<simplepaint version="1.0">`, the end-tag `</simplepaint>`, and everything in between. The elements that are nested inside the root element are represented by their own nodes, which are said to be ***children*** of the root node. An object of type *Element* contains several useful methods. If `element` is of type *Element*, then we have:

- `element.getTagName()` — returns a *String* containing the name that is used in the element's tag. For example, the name of a `<curve>` element is the string "curve".
- `element.getAttribute(attrName)` — if `attrName` is the name of an attribute in the element, then this method returns the value of that attribute. For the element, `<point x="83" y="42"/>`, `element.getAttribute("x")` would return the string "83". Note that the return value is always a *String*, even if the attribute is supposed to represent a numerical value. If the element has no attribute with the specified name, then the return value is an empty string.
- `element.getTextContent()` — returns a *String* containing all the textual content that is contained in the element. Note that this includes text that is contained inside other elements that are nested inside the element.
- `element.getChildNodes()` — returns a value of type *NodeList* that contains all the *Nodes* that are children of the element. The list includes nodes representing other elements and textual content that are directly nested in the element (as well as some other types of node that I don't care about here). The `getChildNodes()` method makes it possible to traverse the entire DOM data structure by starting with the root element, looking at children of the root element, children of the children, and so on. (There is a similar method that returns the attributes of the element, but I won't be using it here.)
- `element.getElementsByTagName(tagName)` — returns a *NodeList* that contains all the nodes representing all elements that are nested inside `element` and which have the given tag name. Note that this includes elements that are nested to any level, not just elements that are directly contained inside `element`. The `getElementsByTagName()` method allows you to reach into the document and pull out specific data that you are interested in.

An object of type *NodeList* represents a list of *Nodes*. It does not use the API defined for lists in the Java Collection Framework. Instead, a value, `nodeList`, of type *NodeList* has two methods: `nodeList.getLength()` returns the number of nodes in the list, and `nodeList.item(i)` returns the node at position `i`, where the positions are numbered `0`, `1`, ..., `nodeList.getLength() - 1`. Note that the return value of `nodeList.get()` is of type *Node*, and it might have to be type-cast to a more specific node type before it is used.

Knowing just this much, you can do the most common types of processing of DOM representations. Let's look at a few code fragments. Suppose that in the course of processing a document you come across an *Element* node that represents the element

```
&lt;background red='255' green='153' blue='51'/&gt;
```

This element might be encountered either while traversing the document with `getChildNodes()` or in the result of a call to `getElementsByTagName("background")`. Our goal is to reconstruct the data structure represented by the document, and this element represents part of that data. In this case, the element represents a color, and the red, green, and blue components are given

by the attributes of the element. If `element` is a variable that refers to the node, the color can be obtained by saying:

```
int r = Integer.parseInt( element.getAttribute("red") );
int g = Integer.parseInt( element.getAttribute("green") );
int b = Integer.parseInt( element.getAttribute("blue") );
Color bgColor = new Color(r,g,b);
```

Suppose now that `element` refers to the node that represents the element

```
<symmetric>true</symmetric>
```

In this case, the element represents the value of a **boolean** variable, and the value is encoded in the textual content of the element. We can recover the value from the element with:

```
String bool = element.getTextContent();
boolean symmetric;
if (bool.equals("true"))
   symmetric = true;
else
   symmetric = false;
```

Next, consider an example that uses a *NodeList*. Suppose we encounter an element that represents a list of *Points*:

```
<pointlist>
   <point x='17' y='42'/>
   <point x='23' y='8'/>
   <point x='109' y='342'/>
   <point x='18' y='270'/>
</pointlist>
```

Suppose that `element` refers to the node that represents the `<pointlist>` element. Our goal is to build the list of type `ArrayList<Point>` that is represented by the element. We can do this by traversing the *NodeList* that contains the child nodes of `element`:

```
ArrayList<Point> points = new ArrayList<Point>();
NodeList children = element.getChildNodes();
for (int i = 0; i < children.getLength(); i++) {
   Node child = children.item(i);   // One of the child nodes of element.
   if ( child instanceof Element ) {
      Element pointElement = (Element)child;  // One of the <point> elements.
      int x = Integer.parseInt( pointElement.getAttribute("x") );
      int y = Integer.parseInt( pointElement.getAttribute("y") );
      Point pt = new Point(x,y); // Create the Point represented by pointElement.
      points.add(pt);            // Add the point to the list of points.
   }
}
```

All the nested `<point>` elements are children of the `<pointlist>` element. The `if` statement in this code fragment is necessary because an element can have other children in addition to its nested elements. In this example, we only want to process the children that are elements.

All these techniques can be employed to write the file input method for the sample program *SimplePaintWithXML.java*. When building the data structure represented by an XML file, my approach is to start with a default data structure and then to modify and add to it as I traverse the DOM representation of the file. It's not a trivial process, but I hope that you can follow it:

```
Color newBackground = Color.WHITE;
ArrayList<CurveData> newCurves = new ArrayList<CurveData>();

Element rootElement = xmldoc.getDocumentElement();

if ( ! rootElement.getNodeName().equals("simplepaint") )
   throw new Exception("File is not a SimplePaint file.");
String version = rootElement.getAttribute("version");
try {
   double versionNumber = Double.parseDouble(version);
   if (versionNumber > 1.0)
      throw new Exception("File requires a newer version of SimplePaint.");
}
catch (NumberFormatException e) {
}

NodeList nodes = rootElement.getChildNodes();

for (int i = 0; i < nodes.getLength(); i++) {
   if (nodes.item(i) instanceof Element) {
      Element element = (Element)nodes.item(i);
      if (element.getTagName().equals("background")) { // Read background color.
         int r = Integer.parseInt(element.getAttribute("red"));
         int g = Integer.parseInt(element.getAttribute("green"));
         int b = Integer.parseInt(element.getAttribute("blue"));
         newBackground = new Color(r,g,b);
      }
      else if (element.getTagName().equals("curve")) { // Read data for a curve.
         CurveData curve = new CurveData();
         curve.color = Color.BLACK;
         curve.points = new ArrayList<Point>();
         newCurves.add(curve);  // Add this curve to the new list of curves.
         NodeList curveNodes = element.getChildNodes();
         for (int j = 0; j < curveNodes.getLength(); j++) {
            if (curveNodes.item(j) instanceof Element) {
               Element curveElement = (Element)curveNodes.item(j);
               if (curveElement.getTagName().equals("color")) {
                  int r = Integer.parseInt(curveElement.getAttribute("red"));
                  int g = Integer.parseInt(curveElement.getAttribute("green"));
                  int b = Integer.parseInt(curveElement.getAttribute("blue"));
                  curve.color = new Color(r,g,b);
               }
               else if (curveElement.getTagName().equals("point")) {
                  int x = Integer.parseInt(curveElement.getAttribute("x"));
                  int y = Integer.parseInt(curveElement.getAttribute("y"));
                  curve.points.add(new Point(x,y));
               }
               else if (curveElement.getTagName().equals("symmetric")) {
                  String content = curveElement.getTextContent();
                  if (content.equals("true"))
                     curve.symmetric = true;
               }
            }
         }
      }
```

```
   }
}
curves = newCurves;  // Change picture in window to show the data from file.
setBackground(newBackground);
repaint();
```

* * *

XML has developed into an extremely important technology, and some applications of it are very complex. But there is a core of simple ideas that can be easily applied in Java. Knowing just the basics, you can make good use of XML in your own Java programs.

Exercises for Chapter 11

1. The sample program *DirectoryList.java*, given as an example in Subsection 11.2.2, will print a list of files in a directory specified by the user. But some of the files in that directory might themselves be directories. And the subdirectories can themselves contain directories. And so on. Write a modified version of `DirectoryList` that will list all the files in a directory and all its subdirectories, to any level of nesting. You will need a **recursive** subroutine to do the listing. The subroutine should have a parameter of type *File*. You will need the constructor from the *File* class that has the form

```
public File( File dir, String fileName )
   // Constructs the File object representing a file
   // named fileName in the directory specified by dir.
```

2. Write a program that will count the number of lines in each file that is specified on the command line. Assume that the files are text files. Note that multiple files can be specified, as in:

```
java  LineCounts  file1.txt  file2.txt  file3.txt
```

Write each file name, along with the number of lines in that file, to standard output. If an error occurs while trying to read from one of the files, you should print an error message for that file, but you should still process all the remaining files. Do not use *TextIO* to process the files; use a *FileReader* to access each file.

3. For this exercise, you will write a network server program. The program is a simple file server that makes a collection of files available for transmission to clients. When the server starts up, it needs to know the name of the directory that contains the collection of files. This information can be provided as a command-line argument. You can assume that the directory contains only regular files (that is, it does not contain any sub-directories). You can also assume that all the files are text files.

When a client connects to the server, the server first reads a one-line command from the client. The command can be the string "`index`". In this case, the server responds by sending a list of names of all the files that are available on the server. Or the command can be of the form "`get filename`", where `filename` is a file name. The server checks whether the requested file actually exists. If so, it first sends the word "ok" as a message to the client. Then it sends the contents of the file and closes the connection. Otherwise, it sends the word "error" to the client and closes the connection.

Ideally, your server should start a separate thread to handle each connection request. However, if you don't want to deal with threads you can just call a subroutine to handle the request. See the `DirectoryList` example in Subsection 11.2.2 for help with the problem of getting the list of files in the directory.

4. Write a client program for the server from Exercise 11.3. Design a user interface that will let the user do at least two things: (1) Get a list of files that are available on the server and display the list on standard output; and (2) Get a copy of a specified file from the server and save it to a local file (on the computer where the client is running).

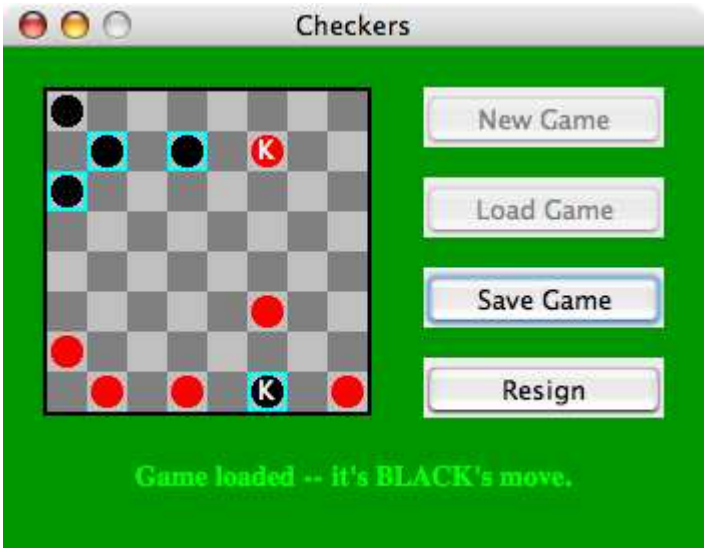
5. The sample program *PhoneDirectoryFileDemo.java*, from Subsection 11.3.2, stores name/number pairs for a simple phone book in a text file in the user's home directory.

Modify that program so that is uses an XML format for the data. The only significant changes that you will have to make are to the parts of the program that read and write the data file. Use the DOM to read the data, as discussed in Subsection 11.6.3. You can use the XML format illustrated in the following sample phone directory file:

```
<?xml version="1.0"?>
<phone_directory>
  <entry name='barney' number='890-1203'/>
  <entry name='fred' number='555-9923'/>
</phone_directory>
```

(This is just an easy exercise in simple XML processing; as before, the program in this exercise is not meant to be a useful phone directory program.)

6. The sample program *Checkers.java* from Subsection 7.5.3 lets two players play checkers. It would be nice if, in the middle of a game, the state of the game could be saved to a file. Later, the file could be read back into the file to restore the game and allow the players to continue. Add the ability to save and load files to the checkers program. Design a simple text-based format for the files. Here is a picture of my solution to this exercise, just after a file has been loaded into the program:



Note: The original checkers program could be run as either an applet or a stand-alone application. Since the new version uses files, however, it can only be run as an application. An applet running in a web browser is not allowed to access files.

It's a little tricky to completely restore the state of a game. The program has a variable `board` of type *CheckersData* that stores the current contents of the board, and it has a variable `currentPlayer` of type **int** that indicates whether Red or Black is currently moving. This data must be stored in the file when a file is saved. When a file is read into the program, you should read the data into two local variables `newBoard` of type *CheckersData* and `newCurrentPlayer` of type **int**. Once you have successfully read all the data from the file, you can use the following code to set up the program state correctly. This code assumes that you have introduced two new variables `saveButton` and `loadButton` of type *JButton* to represent the "Save Game" and "Load Game" buttons:

```
board = newBoard;  // Set up game with data read from file.
currentPlayer = newCurrentPlayer;
legalMoves = board.getLegalMoves(currentPlayer);
selectedRow = -1;
gameInProgress = true;
newGameButton.setEnabled(false);
loadButton.setEnabled(false);
saveButton.setEnabled(true);
resignButton.setEnabled(true);
if (currentPlayer == CheckersData.RED)
   message.setText("Game loaded -- it's RED's move.");
else
   message.setText("Game loaded -- it's BLACK's move.");
repaint();
```

(Note, by the way, that I used a *TextReader* to read the data from the file into my program. *TextReader* is a non-standard class introduced in Subsection 11.1.4 and defined in the file *TextReader.java*. How to read the data in a file depends, of course, on the format that you have chosen for the data.)

Quiz on Chapter 11

1. In Java, input/output is done using streams. Streams are an *abstraction.* Explain what this means and why it is important.

2. Java has two types of streams: character streams and byte streams. Why? What is the difference between the two types of streams?

3. What is a *file?* Why are files necessary?

4. What is the point of the following statement?

```
out = new PrintWriter( new FileWriter("data.dat") );
```

Why would you need a statement that involves two different stream classes, *PrintWriter* and *FileWriter*?

5. The package `java.io` includes a class named *URL*. What does an object of type *URL* represent, and how is it used?

6. Explain what is meant by the *client / server* model of network communication.

7. What is a *socket*?

8. What is a *ServerSocket* and how is it used?

9. Network server programs are often *multithreaded.* Explain what this means and why it is true.

10. What is meant by an *element* in an XML document?

11. What is it about XML that makes it suitable for representing almost any type of data?

12. Write a complete program that will display the first ten lines from a text file. The lines should be written to standard output, `System.out`. The file name is given as the command-line argument `args[0]`. You can assume that the file contains at least ten lines. Don't bother to make the program robust. Do not use *TextIO* to process the file; use a *FileReader* to access the file.