SESSION 10

Programming Languages for Objects

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 text. 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 the abstract class *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, since characters, like any other kind of data, have to be coded as binary numbers to be stored or processed by a computer. 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 one way this is done in Section 11.5.

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, even when working with the standard ASCII character set.

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 an abstract 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 a checked exception, 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. (Checked exceptions and mandatory exception handling were covered in Subsection 8.3.3.)

The *InputStream* class also defines methods for reading multiple 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 bits 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 the analogous 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);
```

In fact, the parameter to the constructor can also be an *OutputStream* or a *File*, and the constructor will build a *PrintWriter* that can write to that output destination. (Files are covered in the next section.) When you output data to the *PrintWriter* `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 destination. 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 formatted 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 2.4.1, and `out.printf` has the same functionality.
- `out.flush()` -- ensures that characters that have been written with the above methods are actually sent to the output destination. In some cases, notably when writing to a file or to the network, it might be necessary to call this method to force the output to actually appear at the destination.

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*.

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 ultimately 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.

There are various ways for characters to be encoded as binary data. A particular encoding is known as a charset or character set. Charsets have standardized names such as "UTF-16," "UTF-8," and "ISO-8859-1." In UTF-16, characters are encoded as 16-bit UNICODE values; this is the character set that is used internally by Java. UTF-8 is a way of encoding UNICODE characters using 8 bits for common ASCII characters and longer codes for other characters. ISO-8859-1, also known as "Latin-1," is an 8-bit encoding that includes ASCII characters as well as certain accented characters that are used in several European languages. *Readers* and *Writers* use the default charset for the computer on which they are running, unless you specify a different one. This can be done, for example, in a constructor such as

```
Writer charSink = new OutputStreamWriter( byteSink, "ISO-8859-
1" );
```

Certainly, the existence of a variety of charset encodings has made text processing more complicated -- unfortunate for us English-speakers but essential for people who use non-Western character sets. Ordinarily, you don't have to worry about this, but it's a good idea to be aware that different charsets exist in case you run into textual data encoded in a non-default way.

11.1.4 Reading Text

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*. (The *Scanner* class, introduced briefly in Subsection 2.4.6 and covered in more detail below, comes pretty close.) There is one basic case that is easily handled by the standard class *BufferedReader*, which 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 ) {
      processOneLineOfInput( line );
      line = in.readLine();
   }
}
catch (IOException e) {
}
```

This code segment reads and processes lines from standard input until 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`.

Note that the main purpose of *BufferedReader* is not simply to make it easier to read lines of text. Some I/O devices work most efficiently if data is read or written in large chunks, instead of as individual bytes or characters. A *BuferedReader* reads a chunk of data, and stores it in internal memory. The internal memory is known as a buffer. When you read from the BufferedReader, it will take data from the buffer if possible, and it will only go back to its input source for more data when the buffer is emptied. There is also a *BufferedWriter* class, and there are buffered stream classes for byte streams as well.

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 input source at a time 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 character 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; those objects can then be used to read data from several files or other input sources 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 is 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 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.
- `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 void close()` -- Closes the input stream. This should be done when finished reading from the stream. (*TextReader* implements the *AutoCloseable* interface and so can be used as a "resource" in a `try..catch` statement, as discussed at the end of Subsection 8.3.2.)

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 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. (My *TextIO* and *TextReader* classes are not complete solutions either.) 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, tabs, and end-of-line markers, but you can change a *Scanner's* delimiters if you need to. 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 -- read 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. With so many input classes available -- *BufferedReader*, *TextReader*, *Scanner* -- you might have trouble deciding which one to use! In general, I would recommend using a *Scanner* unless you have some particular reason for preferring the *TextIO*-style input routines of *TextReader*. *BufferedReader* can be used as a lightweight alternative when all that you want to do is read entire lines of text from the input source.

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 the actual type of the object that was read.

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. These primitive type values can be interspersed with objects in the data.

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 Section 11.5 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 GUI component classes and many other classes in Swing and in the AWT. One of the programming examples in Section 11.3 uses object IO.

One warning about using *ObjectOutputStreams*: These streams are optimized to avoid writing the same object more than once. When an object is encountered for a second time, only a reference to the first occurrence is written. Unfortunately, if the object has been modified in the meantime, the new data will not be written. Because of this, *ObjectOutputStreams* are meant mainly for use with "immutable" objects that can't be changed after they are created. (*Strings* are an example of this.) However, if you do need to write mutable objects to an *ObjectOutputStream*, and if it is possible that you will write the same object more than once, you can ensure that the full, correct version of the object can be written by calling the stream's `reset()` method before writing the object to the stream.

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 can be read from a file using an object belonging to the class *FileReader*, which is a subclass of *Reader*.

Similarly, data can be 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.

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. 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
}
```

To use a *Scanner* to read from the file, you can construct the scanner in a similar way. However, it is more common to construct it from an object of type *File* (to be covered below):

```
Scanner in;

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

Once you have a *Scanner* or *TextReader* for reading from a file, you can get data from the file using exactly the same methods that work with any *Scanner* or *TextReader*. When you read from a file using either of these, exceptions can occur. Since the exceptions in this case are not checked exceptions, you are not forced to enclose your input commands in a `try..catch` statement, but it is usually a good idea to do it anyway.

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
}
```

However, as with *Scanner*, it is more common to use a constructor that takes a *File* as parameter; this will automatically wrap the File in a FileWriter before creating the PrintWriter:

```
PrintWriter result;

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

You can even use just a *String* as the parameter to the constructor, and it will be interpreted as a file name (but you should remember that a *String* in the *Scanner* constructor does not name a file; instead the file will read characters from the string itself).

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.

When you are finished with a *PrintWriter*, you should call its `flush()` method, such as "`result.flush()`", to make sure that all the output has been set to its destination. If you forget to do this, you might find that some of the data that you have written to a file has not actually shown up in the file.

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 *PrintWriter*, *TextReader*, or *Scanner*. 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, *PrintWriter*, *TextReader*, and *Scanner* 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 it is better not to depend on this.

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 real numbers. The input file is read using a *Scanner*. 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 is guaranteed to be executed, no matter what. See Subsection 8.3.2.)

```
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));

            result.flush();  // Make sure data is actually sent
to the file.

            if (result.checkError())
                System.out.println("Some error occurred while
writing the file.");
            else
                System.out.println("Done!");

        }
        catch (IOException e) {
            // Some problem reading the data from the input
file.
```

```
            // (Note that PrintWriter doesn't throw exceptions
on output errors.)
            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 class ReverseFileWithTextReader
```

A version of this program that uses a *Scanner* instead of a *TextReader* can be found in *ReverseFileWithScanner.java*. Note that the *Scanner* version does not need the final `try..catch` from the *TextReader* version, since the *Scanner* method for reading data doesn't throw an *IOException*. Instead, the program will simply stop reading data from the file if it encounters anything other than a number in the input.

As mentioned at the end of Subsection 8.3.2, the pattern of creating or opening a "resource," using it, and then closing the resource is a very common one, and the pattern is supported by the syntax of the `try..catch` statement. Files are resources in this sense, as are *Scanner*, *TextReader*, and all of Java's I/O streams. All of these things define `close()` methods, and it is good form to close them when you are finished using them. Since they all implement the *AutoCloseable* interface, they are all resources in the sense required by `try..catch`. A `try..catch` statement can be used to automatically close a resource when the `try` statement ends, which eliminates the need to close it by hand in a `finally` clause. This assumes that you will open the resource and use it in the same `try..catch`.

As an example, the sample program *ReverseFileWithResources.java* is another version of the example we have been looking at. In this case, `try..catch` statements using the resource pattern are used to read the data from a file and to write the data to a file. My original program opened a file in one `try` statement and used it in another `try` statement. The resource pattern requires that it all be done in one `try`, which requires some reorganization of the code (and can sometimes make it harder to determine the exact cause of an exception). Here is the `try..catch` statement from the sample program that opens the input file, reads from it, and closes it automatically.

```
try( TextReader data = new TextReader(new
FileReader("data.dat")) ) {
    // Read numbers, adding them to the ArrayList.
    while ( data.eof() == false ) {  // Read until end-of-file.
        double inputNumber = data.getlnDouble();
        numbers.add( inputNumber );
    }
```

```
}
catch (FileNotFoundException e) {
        // Can only be caused by the TextReader constructor
    System.out.println("Can't open input file data.dat!");
    System.out.println("Error: " + e);
    return;  // Return from main(), since an error has
occurred.
             //  (Otherwise, the program would try to do the
output!)
}
catch (IOException e) {
        // Can occur when the TextReader tries to read a
number.
    System.out.println("Error while reading from file: " + e);
    return;  // Return from main(), since an error has
occurred.
}
```

The resource, `data` is constructed on the first line. The syntax requires a declaration of the resource, with an initial value, in parentheses after the word "try." It's possible to have several resource declarations, separated by semicolons. They will be closed in the order opposite to the order in which they are declared.

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.
- `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." The corresponding path on Windows is `..\examples\data.dat`.

When working on the command line, it's safe to say 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. You can then use the path name, for example, in a constructor for a *File* or a *PrintWriter*. 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 a directory. The second can be the name of the file in that directory or a relative path from that 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 output stream.
- `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`. The method `file.listFiles()` is similar, except that it returns an array of *File* instead of an array of *String*

Here, for example, is a program that will list the names of all the files in a directory specified by the user. In this example, I have used a *Scanner* 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)`.

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 6.7.2. File dialog boxes are similar to those, but are just 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 return 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.
            .
         }
         catch (Exception e) {
            JOptionPane.showMessageDialog(this,
                "Sorry, but an error occurred while trying to read
the data:\n" + e);
         }
         finally {
            in.close();
         }
      }
```

One fine point here is that the variable `fileDialog` is an instance variable of type *JFileChooser*, not a local variable. 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. Otherwise, the selected file could be carried over from the previous time the file dialog was used.

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("output.txt"); // (default
output 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 {
      out = new PrintWriter( selectedFile );
   }
   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. (Does not
throw exceptions.)
      .
     out.flush();
     out.close();
     if (out.checkError())   // (need to check for errors in
PrintWriter)
        throw new IOException("Error occurred while trying to
write file.");
   }
   catch (Exception e) {
      JOptionPane.showMessageDialog(this,
          "Sorry, but an error occurred while trying to write
the data:\n" + e);
```

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. Subsection 4.3.6 included a program for copying text files using *TextIO*. The example in this section will work for any file.

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 byte streams *InputStream* and *OutputStream* to operate on the file. 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 4.3.6.) 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.

The program *CopyFile.java* 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
```

It is not terribly efficient to copy one byte at a time. Efficiency could be improved by using alternative versions of the `read()` and `write()` methods that read and write multiple bytes (see the API for details). Alternatively, the input and output streams could be wrapped in objects of type *BufferedInputStream* and *BufferedOutputStream* which automatically read from and write data to files in larger blocks, which is more efficient than reading and writing individual bytes.

(There is also a sample program *CopyFileAsResources.java* that does the same thing as `CopyFile` but uses the resource pattern in a `try..catch` statement to make sure that the streams are closed in all cases.)

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" );
        // A file named .phone_book_data in the user's home
directory.

if ( ! dataFile.exists() ) {
   System.out.println("No phone book data file found.  A new one");
   System.out.println("will be created, if you add any entries.");
   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.flush();
        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, 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*. 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();
    Scanner in;
    try {
        in = new Scanner( selectedFile );
    }
    catch (FileNotFoundException e) {
        JOptionPane.showMessageDialog(this,
             "Sorry, but an error occurred while trying to open the
file:\n" + e);
        return;
    }
    try {
        StringBuilder input = new StringBuilder();
        while (in.hasNextLine()) {
            String lineFromFile = in.nextLine();
            if (lineFromFile == null)
                break;  // End-of-file has been reached.
            input.append(lineFromFile);
            input.append('\n');
            if (input.length() > 10000)
                throw new IOException("Input file is too large for
this program.");
        }
        text.setText(input.toString());
        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);
    }
    finally {
        in.close();
    }
}
```

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. (*TrivialEdit* is defined as a subclass of *JFrame*.)

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 7.3.3. (You might want to run it now to remind yourself of what it can do.) 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 `pt.x` and `pt.y` of type int that represent the pixel 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.
}
```

Then, a list of type `ArrayList<CurveData>` is used to hold data for all of the curves that the user has drawn. Note that in the new version of the program, the *CurveData* 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. This 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()` method from `SimplePaintWithFiles`, with the changes from the generic `writeFile()` method shown in red:

```
/**
 * 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.flush();
               out.close();
               editFile = selectedFile;
               setTitle("SimplePaint: " + editFile.getName());
            }
            catch (Exception e) {
               JOptionPane.showMessageDialog(this,
                  "Sorry, but an error occurred while trying to write the
         file:\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 last line depends on the fact that the *CurveData* class implements the *Serializable* interface. (So does the first; the *Color* class, like many of Java's standard classes, implements *Serializable*.)

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. 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 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"); // Name and 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; // default value
            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;  // default value
                    curve.symmetric = false;    // default value
                    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 11.5, we will look at one solution to the data representation problem that has become increasingly common.

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 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 -- or even running on the same computer. 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." (URLs can actually refer to resources from other sources besides the web; after all, they are "universal".)

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, plus 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.

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 *BufferedReader*, 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/png", 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 "png". 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.  Note: If
        getContentType() method were called before getting the
input
        stream, it is possible for contentType to be null only
because
        no connection can be made.  The getInputStream() method
will
        throw an error if no connection can be made. */
```

```
        String contentType = connection.getContentType();
        System.out.println("Stream opened with content type: " +
contentType);
        System.out.println();
        if (contentType == null || contentType.startsWith("text") ==
false)
              throw new IOException("URL does not seem to refer to a
text file.");
        System.out.println("Fetching context from " + urlString + "
...");
        System.out.println();

        /* 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);
        }
        in.close();

} // end readTextFromURL()
```

A complete program that uses this subroutine can be found in the file *FetchURL.java*. When you run the program, you can specify the URL on the command line; if not, you will be prompted to enter the URL. For this program, a URL can begin with "http://" for a URL that refers to a resource on the web, with "file://" for a URL that refers to a file on your computer, or with "ftp://" for a URL that uses the "File Transfer Protocol." If it does not start with any of these, then "http://" is added to the start of the URL. Try the program with URL `math.hws.edu/javanotes` to fetch the front page of this textbook on the web. Try it with some bad inputs to see the various errors that can occur.

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 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. We'll look at how it works in the next chapter.

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. 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 `64.89.144.135`, 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 same line 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. (The slashes at the start of each address are not part of the actual address.) The other numbers in the output are IPv6 addresses.

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 actually has a port number in combination with an IP address. A port number is just a 16-bit positive 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 in Java

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. 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. (A parameter value of 0 in this method tells the server socket to listen on any available port.)

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 program -- or more exactly, the thread -- that called the method can't do anything else. If there are other threads in the same program, they 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 that you want it to continue to accept connections as long as the program is running. 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 the
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 as 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 can be specified as a command-line argument. For example, if the server is running on a computer named math.hws.edu, then you could run the client with the command "`java DateClient math.hws.edu`". If a computer is not specified on the command line, then the user is prompted to enter one. Here is the complete client program:

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

/**
 * This program opens a connection to a computer specified
 * as the first command-line argument.  If no command-line
 * argument is given, it prompts the user for a computer
 * to connect to.  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, DataServer, 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 {
            Scanner stdin = new Scanner(System.in);
            System.out.print("Enter computer name or IP address:
");
            hostName = stdin.nextLine();
        }

        /* 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 request 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 would 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 web 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 `outgoing.println()` to send a line of data to the client, the server program calls `outgoing.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; the client is similar:

```
import java.net.*;
import java.util.Scanner;
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.

        Scanner 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 Scanner(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.nextLine();
                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 a client and a server as part of setting up a 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. Note that if you run *CLChatClient* without specifying a computer on the command line, then you will be asked to type in the name or IP address of the server computer.

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 are also alternatives to DTDs, such 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 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.

11.5.1 Basic XML Syntax

If you know HTML, the language for writing web pages, then XML will look familiar. An XML document looks a lot like an HTML document. 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'/>`, with a "`/`" before the final "`>`". This 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 author of a well-formed XML document gets to choose the tag names and attribute names, and meaningful names can be chosen to descibe the data to a human reader. (For a valid 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.

11.5.2 Working With the DOM

The sample XML file shown above was designed to store information about simple drawings made by the user. The drawings are ones that could be made using the sample program *SimplePaint2.java* from Subsection 7.3.3. We'll look at another version of that program that can save the user's drawing using an XML format for the data file. The new version is *SimplePaintWithXML.java* 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`. This produces an XML file with the same structure as the example shown above:

```
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. This could be difficult to do by hand. 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 well-formed 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 encounters. 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*. Unfortunately, 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

```
<background red='255' green='153' blue='51'/>
```

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();
```

You can find the complete source code in *SimplePaintWithXML.java*.