

File Input and Output

Supplement

2

Contents

Preliminaries

- Why Files?

- Streams

- The Kinds of Files

- File Names

Text Files

- Creating a Text File

- Reading a Text File

- Changing Existing Data in a Text File

- Defining a Method to Open a Stream

Binary Files

- Creating a Binary File of Primitive Data

- Reading a Binary File of Primitive Data

- Strings in a Binary File

- Object Serialization

Prerequisite

- Prelude

- Designing Classes

- Java Interlude 2

- Exceptions

- Supplement 1 (online)

- Java Basics

Input to a program can come from a keyboard or a mouse, and its output can be displayed on a screen. These forms of input and output occur in real time and are temporal. When the program ends, the input and output vanish.

However, program input can be taken from a file, and its output can be sent to a file. A **file** is simply data on a particular storage medium, such as a disk. In this supplement, we explain how you can read input from a file and send output to another file, and why you would want to do so.

Preliminaries

We begin by looking at some generalities about files.

Why Files?

S2.1 You own files such as Java programs, songs, photos, and so on. Each of these files was created by a program. You might have run the program yourself—such as when you used a text editor to write a Java program—or obtained the result of someone else’s program in the form of a song or picture. In any event, you clearly do not want the output of the program to disappear when program execution stops. You want the data to last, to be persistent. By “persistent,” we mean “to last beyond program execution.” The contents of a file last until a program changes them or they become damaged. We want to use our files—that is, we want to run our programs, listen to our music, and watch our videos. The reasons for creating files should be obvious.

Would you ever use a file as input to a program? You have done so if you have ever edited a photo or revised an essay you wrote yesterday. Java programs—like the ones we have written—can read data from an input file rather than from the keyboard. Thus, files provide a convenient way to deal with large data sets.



Note: Reasons why a program creates a file

- A program creates and saves a file so it can be used over and over by other programs.
- A program creates a file for its own use as temporary storage for numerous intermediate results. The program writes data into the file and later reads the data, but does not save the file.

Streams

S2.2 In Java, all input and output of data—including reading a file or writing a file—involves streams. A **stream** is an object that represents a flow of data. The data is a collection of eight-bit bytes representing numbers, characters, music, and so on. A stream either

- Sends data from your program to a destination, such as a file or the screen—in which case it is called an **output stream**—or
- Takes data from a source, such as a file or the keyboard, and delivers it to your program—in which case it is called an **input stream**

For example, the object `System.out` is an output stream that moves data from a program to a display. If an output stream is connected to a file, data will move from the program to the file. That is, the program will write the file. Likewise, `System.in` is an input stream that moves data from the keyboard to a program. If an input stream is connected to a file, the program will read data from the file.



Note: Input and output are done from the perspective of the program. Thus, “input” means that data moves *into* your program from an input device such as a disk or keyboard. The word “output” means that data moves *out of* your program to an output device such as a disk or the screen.



Note: Streams in Java are objects of certain classes in the Java Class Library. In particular, these classes are in the package `java.io`. Thus, to use these classes, you must import them from `java.io`.

Before you can read or write a file, you must connect the file to an appropriate stream and associate it with your Java program. You accomplish these steps, or **open** the file, when you create the stream by invoking the constructor of the stream's class. After you are finished with the file you must **close** it, or disconnect it from the stream, and hence from your program, by calling a particular stream method named `close`. We will examine the details of these steps shortly.

The Kinds of Files

S2.3 All files are written as representations of ones and zeros, that is, binary digits, or bits. Java, however, treats files as either text files or binary files. A **text file** represents a collection of characters. The streams associated with text files provide methods that interpret the file's binary contents as characters. The files that contain your Java programs are most likely text files. A text editor can read a text file and make it appear to you as a sequence of characters. Any file other than a text file is called a **binary file**. For example, your song and picture files are binary files.

A Java program can create or read text files and binary files. The steps for processing—that is, writing or reading—a text file are analogous to those for a binary file. The kind of file determines which stream classes we use to perform the input or output.



Programming Tip: Choosing the kind of file

Your Java program should use a text file if you will use a text editor to

- Create files that the program will read
- Read or edit files that the program will create

If you will not use a text editor to create or read a file, consider using a binary file, as such files typically require less disk space than text files.

File Names

S2.4 Although Java does not specify the characters that can make up a file name, your operating system does. Typically, you use letters, digits, and a dot in the name of a data file, ending it with a suffix, such as `.txt`. This suffix is simply a convention, not a Java rule, although your operating system can give it meaning. This book uses the suffix `.txt` to indicate a text file.

Within a Java program, the name of a file is a string. Although our examples will use a `String` constant as a file name, our programs could have asked the user for the file's name and stored it in a `String` variable.

Text Files

We begin by exploring text files.

Creating a Text File

S2.5 The contents of a text file. A text file contains a sequence of characters in which each character is represented by the system's default encoding. Java uses the Unicode character set, which includes many letters in natural languages that are quite different from English. The Unicode representation of each character requires two bytes. Many text editors, operating systems, and programming languages other than Java use the ASCII character set. The ASCII character set is a subset of the Unicode character set and contains the characters normally used for English and typical Java programs. The representation of each character in ASCII requires one byte.

A typical text file is organized as lines, each ending with a special end-of-line character. The lines in a text file are analogous to the lines you see when a program displays its output. In reality, however, the file is a sequence of data. For this reason, we say that a text file offers **sequential access** to its contents. That is, before you can read the n^{th} line of the file, you start at the file's beginning and read through its first $n - 1$ lines.

S2.6 Opening a text file for output. The standard class `PrintWriter`, which is in the package `java.io` of the Java Class Library, has the familiar methods `print` and `println`. As you will see, these methods behave like `System.out.print` and `System.out.println`. Thus, we will use this class to create an output stream for a text file.

Before you can write to a text file, you must open it by writing a statement like

```
PrintWriter toFile = new PrintWriter(fileName);
```

The call to `PrintWriter`'s constructor creates an output stream and connects it to the file named by the `String` variable `fileName`. The **stream variable** `toFile` references this stream. Once the stream is created, your program always refers to the file by using the stream variable instead of its actual file name.

When you connect a file to an output stream in this way, your program always starts with an empty text file. If the file named by `fileName` did not exist before the constructor was called, a new, empty file is created and given that name. However, if the file named by `fileName` already exists, its old contents will be lost.

S2.7 `PrintWriter`'s constructor can throw a checked exception—`FileNotFoundException`—while attempting to open a file. For this reason, its invocation must appear within either a `try` block that is followed by an appropriate `catch` block or a method whose header lists this exception in a `throws` clause. For example, we could write the following statements to open the text file `data.txt` for output:

```
String fileName = "data.txt";
PrintWriter toFile = null;
try
{
    toFile = new PrintWriter(fileName);
}
catch (FileNotFoundException e)
{
    System.out.println("PrintWriter error opening the file " + fileName);
    System.out.println(e.getMessage());
    < Possibly other statements to deal with this exception. >
}
```

Notice that we declared the variable `toFile` outside of the `try` block so that `toFile` is available outside of this block. Remember, anything declared in a block—even a `try` block—is local to the block. Declaring `toFile` in this way enables us to use it to write to the file, as you will see.

A `FileNotFoundException` does not necessarily mean that the file was not found. After all, if you are creating a new file, it doesn't already exist. In that case, an exception is thrown if the file could not be created because, for example, the file name is already used as a folder (directory) name.



Note: A `FileNotFoundException` will occur if a file cannot be opened for output, either because it does not exist and cannot be created or because it is inaccessible.

S2.8 Writing a text file. The methods `println` and `print` of the class `PrintWriter` work the same for writing to a text file as the respective methods `System.out.println` and `System.out.print` work for writing to the screen. Thus, when a program writes a value to a text file, the number of characters written is the same as if it had written the value to the screen. For example, writing the `int` value 12345 to a text file places five characters in the file. In general, writing an integer of type `int` places between 1 and 11 characters in a text file.

The following statements write four lines to the text file we created in the previous segment:

```
for (int counter = 1; counter <= 4; counter++)
    toFile.println("Line " + counter);
```

Notice that we use the stream variable `toFile` when calling `println`, not the actual name of the file. Additionally, we did not place the call to `println` within a `try` block, as `println` does not throw any checked exceptions. The same is true of `print`.



Note: `System.out` is an object of the standard class `PrintStream`. Because both `PrintStream` and `PrintWriter` define the same `print` and `println` methods, objects of `PrintWriter` have the same `print` and `println` methods as `System.out`. However, `System.out` directs its output stream to a different destination than `PrintWriter` objects do.

S2.9 Buffering. Instead of sending output to a file immediately, `PrintWriter` waits to send a larger packet of data. Thus, the output from `println`, for example, is not sent to the output file right away. Instead, it is saved and placed into a portion of memory called a **buffer**, along with the output from other invocations of `print` and `println`. When the buffer is too full to accept more output, its contents are written to the file. Thus, the output from several `print` and `println` statements is written at the same time, instead of each time one statement executes. This technique is called **buffering**, and it saves execution time.

S2.10 Closing a text file. When we are finished using a file, we must disconnect it from the stream. Every stream class, including `PrintWriter`, has a method named `close` to accomplish this task. Thus, to close the file associated with the stream variable `toFile`, we write

```
toFile.close();
```

When closing a file, the system writes any data still left in the buffer to the file and releases any resources it used to connect the stream to the file. Note that `close` will not throw an exception.



Programming Tip: If you do not close a file, Java will close it for you, but only if your program ends normally. To avoid any possible loss of data or damage to a file, you should close it as soon as possible after you are finished using it.

S2.11 Flushing an output text file. You can force any pending output that is currently in a buffer to be written to its destination file by calling `PrintWriter`'s method `flush`, as follows:

```
toFile.flush();
```

The method `close` automatically calls the method `flush`, so for most simple applications, you do not need to call `flush` explicitly. However, if you continue to program, you will eventually encounter situations in which you will have to use `flush`.

S2.12 Example: Creating a text file of user data. Let's create a text file of data that a user enters at the keyboard. We'll provide this operation as a static method within a class that reads and writes text files. If the method accepts as arguments the file name and number of lines that the user will enter, its header could be as follows:



```
public static boolean createTextFile(String fileName, int howMany)
```

The only checked exception that can occur is during the opening of the file, so our method can return a boolean value to indicate whether the operation succeeds.

Listing S2-1 shows the definition of this method within a class `TextFileOperations`.

LISTING S2-1 The static method `createTextFile` in the class `TextFileOperations`

```

1  import java.io.FileNotFoundException;
2  import java.io.PrintWriter;
3  import java.util.Scanner;
4  public class TextFileOperations
5  {
6      /** Writes a given number of lines to the named text file.
7       * @param fileName The file name as a string.
8       * @param howMany The positive number of lines to be written.
9       * @return True if the operation is successful. */
10     public static boolean createTextFile(String fileName, int howMany)
11     {
12         boolean fileOpened = true;
13         PrintWriter toFile = null;
14         try
15         {
16             toFile = new PrintWriter(fileName);
17         }
18         catch (FileNotFoundException e)
19         {
20             fileOpened = false; // Error opening the file
21         }
22
23         if (fileOpened)
24         {
25             Scanner keyboard = new Scanner(System.in);
26             System.out.println("Enter " + howMany + " lines of data:");
27             for (int counter = 1; counter <= howMany; counter++)
28             {
29                 System.out.print("Line " + counter + ": ");
30                 String line = keyboard.nextLine();
31                 toFile.println(line);
32             } // end for
33
34             toFile.close();
35         } // end if
36
37         return fileOpened;
38     } // end createTextFile
39 } // end TextFileOperations

```

S2.13 Appending to a text file: Getting ready. When you add a line to the end of an existing text file, you append it to the file. Since we actually will write a new line to the file, we want to use the class `PrintWriter`, as we did to create the file. When we consult the documentation for `PrintWriter`,

however, we find no constructor that will open an existing file and append data to it. As we mentioned before, when you open a file for output, you ordinarily lose any existing data.

Although `PrintWriter` does not have an appropriate constructor, it is still the class we want to use to write the file. What we need is another class to help us open the file in the way we want. Such a class is `FileWriter`, which is in the package `java.io` of the Java Class Library. The appropriate `FileWriter` constructor is declared as follows:

```
public FileWriter(String fileName, boolean append)
```

This constructor opens for output the text file named by the `String` variable `fileName`. If the argument `append` is true, any data written to the file will be appended to its end. Otherwise, if `append` is false, any existing data in the file is lost.

Conveniently, `PrintWriter` has a constructor that accepts a `FileWriter` object as its argument. Thus, we can use `FileWriter` to open the text file so that it will accept additional output, and then use `PrintWriter` to provide methods such as `print` and `println` to write the output. Note that `FileWriter` does not have such methods.

S2.14 Appending to a text file: Writing new lines. Most of `FileWriter`'s constructors, including the one given in the previous segment, can throw an `IOException` if they cannot open the designated file. The reasons for such an occurrence are the same as we encountered before: Either no such file exists and one cannot be created; `fileName` names an existing text file, but it cannot be opened; or `fileName` is actually the name of a folder instead of a text file.

Since the constructors of both `FileWriter` and `PrintWriter` can throw an exception, we invoke them within a try block. For example, to append another line to the existing text file `CollegeFile.txt`, we could open it as follows:

```
try
{
    FileWriter fw = new FileWriter(fileName, true); // IOException?
    toFile = new PrintWriter(fw);                  // FileNotFoundException?
}
catch (FileNotFoundException e)
{
    System.out.println("PrintWriter error opening the file " + fileName);
    System.out.println(e.getMessage());
    System.exit(0);
}
catch (IOException e)
{
    System.out.println("FileWriter error opening the file " + fileName);
    System.out.println(e.getMessage());
    System.exit(0);
}
```

We now can add one or more lines of data to the end of the file by using statements of the form

```
toFile.println(. . .);
```



Note: Why do we need `PrintWriter`? Isn't `FileWriter` enough?

Although the class `FileWriter` has the constructor we need to append data to a file, it provides only basic support for text files. In particular, `FileWriter`'s methods can write only character arrays and strings to a file. The class `PrintWriter` lacks the necessary constructor, but it has useful methods such as `println` to write formatted representations of objects to a file. Using both classes provides an appropriate constructor and convenient methods for appending data to a file.

Reading a Text File

You know how to use the class `Scanner` to read data from the keyboard. We can use this same class to read data from a text file. Let's see how. Realize, however, that `Scanner` is not a stream class.

S2.15 Opening a text file for input. Previously, we invoked `PrintWriter`'s constructor to open a text file for output. In an analogous way, we might invoke `Scanner`'s constructor to open a text file for input. You might guess that we would accomplish this step by using a statement such as

```
Scanner fromFile = new Scanner(fileName); // Does NOT open a text file
```

where `fileName` is a `String` variable representing the file's name. Unfortunately, this statement has nothing to do with any text file! Instead, the `Scanner` object `fromFile` extracts portions of the string `fileName`, as Supplement 1 (online) describes, beginning with Segment S1.81.



Note: A Scanner object is not a stream

We will use `Scanner` to get data from a text file, but `Scanner` is not a stream class. In fact, it belongs to the package `java.util`, not `java.io`. According to the constructor we use to create it, a `Scanner` object can process a string

- Entered at the keyboard or
- Read from a text file or
- Given to its constructor as an argument

The `Scanner` objects we will use to read a text file actually contain a stream object that reads the file. The `Scanner` methods then translate, or parse, the input string, convert the data to the desired data type, and pass it to our program.

`Scanner` has several constructors, one of which takes an instance of the standard class `File`. This latter class, which is in the package `java.io`, represents a file in a system-independent and abstract way. Moreover, one of `File`'s constructors accepts a string that is the file's name as its argument. Thus, to access the file whose name is referenced by the `String` variable `fileName`, we would write

```
Scanner fileData = new Scanner(new File(fileName));
```

This constructor opens the file for input and creates a private input stream. Even though the `Scanner` object `fileData` is not a stream object, its `Scanner` methods can read from the file using the private input stream.

Unlike other constructors of `Scanner` that we use when either reading from the keyboard or processing a string, this new constructor can throw a `FileNotFoundException`. Thus, its invocation must appear within either a `try` block that is followed by an appropriate `catch` block or a method whose header lists this exception in a `throws` clause. For example, the following statements ultimately open the text file named `data.txt` for input:

```
String fileName = "data.txt";
Scanner fileData = null;
try
{
    // Can throw FileNotFoundException
    fileData = new Scanner(new File(fileName));
}
catch (FileNotFoundException e)
{
    System.out.println("Scanner error opening the file " + fileName);
    System.out.println(e.getMessage());
    < Possibly other statements that react to this exception. >
}
```


**Note: The class File**

Segment S2.15 introduced the standard class `File`, whose constructor accepts as its argument a string that is a file's name. Because `Scanner` has no constructor that accepts a file name as an argument, but `File` does, we can open a text file for input by using a statement such as

```
Scanner fileData = new Scanner(new File(fileName));
```

This `Scanner` constructor accepts a `File` object as its argument and instantiates an input stream connected to the file. Although this stream belongs to the `Scanner` object `fileData` and is hidden from us, we can use `fileData` to read from the file.

The class `File` is useful in other ways as well. For example, you can

- Test whether a file exists, using the method `exists`
- Test whether a file exists and can be read, using the method `canRead`
- Test whether a file exists and can be written, using the method `canWrite`
- Change the name of a file, using the method `renameTo`
- Delete a file, using the method `delete`

The next segment will use `File` to perform some of these tasks. For more details, you should consult the online documentation for this class in the Java Class Library.



Note: The `Scanner` constructor, whose parameter is a `File` object, will throw a `FileNotFoundException` if it cannot open a file for input because the file either does not exist or is inaccessible.

**Note: Input from a text file**

Opening a text file for input enables you to read data from it sequentially, starting from the file's beginning. Note that other standard classes exist that let you open a text file directly, rather than indirectly by using `Scanner`. However, `Scanner` offers you a convenient way to read from a text file.

S2.16 Reading a text file. All of `Scanner`'s methods are available to you when reading a text file. If you do not know the format of the data in the file, you can use the `Scanner` method `nextLine` to read it line by line. For example, the following statements read and display the lines in the existing text file `data.txt`:

```
while (fileData.hasNextLine())
{
    String line = fileData.nextLine();
    System.out.println(line);
} // end while
```

If `nextLine` were to read beyond the end of a file, it would throw the runtime exception `NoSuchElementException`. We use `Scanner`'s method `hasNextLine` to prevent `nextLine` from reading beyond the end of the file. Thus, the `while` loop ends when the end of the file is reached.

S2.17 Example. Listing S2-2 adds the static method `displayFile` to our previous class `TextFileOperations`, as shown in Listing S2-1. Notice the previous code fragments in the context of this method. After the text file is read, we use the `Scanner` method `close` to indirectly close the file. Also, notice that the `try` block here opens, reads, and closes the file, whereas the `try` block in Listing S2-1 only opens the file. However, that code requires an `if` statement to skip further file processing if the file cannot be opened.



LISTING S2-2 The static method `displayFile` in the class `TextFileOperations`

```

1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.io.PrintWriter;
4  import java.util.Scanner;
5  /**
6   * A class of static methods that create and display a text file of
7   * user-supplied data.
8   */
9  public class TextFileOperations
10 {
11     < The method createTextFile, as given in Listing S2-1, appears here. >
12
13     /** Displays all lines in the named text file.
14      * @param fileName The file name as a string.
15      * @return True if the operation is successful. */
16     public static boolean displayFile(String fileName)
17     {
18         boolean fileOpened = true;
19         try
20         {
21             Scanner fileData = new Scanner(new File(fileName));
22             System.out.println("The file " + fileName +
23                               " contains the following lines:");
24             while (fileData.hasNextLine())
25             {
26                 String line = fileData.nextLine();
27                 System.out.println(line);
28             } // end while
29             fileData.close();
30         }
31         catch (FileNotFoundException e)
32         {
33             fileOpened = false; // Error opening the file
34         }
35         return fileOpened;
36     } // end displayFile
37 } // end TextFileOperations

```

**Note: Other methods in Scanner**

The previous example simply reads and displays entire lines of a text file. However, `Scanner` has other convenient methods that read data in various formats. Segment S1.33 of Supplement 1 (online) mentioned `nextInt` and `nextDouble` in conjunction with reading from the keyboard. You can use these and other similar methods to read data from a text file as well. Note that none of `Scanner`'s methods throw checked exceptions, and so they need not appear within a `try` block. However, `Scanner`'s constructor can throw a checked exception.



Programming Tip: When you read a value at the end of a line using any `Scanner` method other than `nextLine`, the character or characters marking the end of the line remain as the next to be read. Only `nextLine` reads past these characters. So regardless of whether you read from the keyboard or from a text file using `Scanner`, you must read beyond these ending characters—by using `nextLine`—before trying to read the next string.

**Note: Reading a file more than once**

Reading a file can be time consuming, particularly when it is quite large. Therefore, you should not read a file more than once, if at all possible. Some algorithms, however, do require you to process data more than once. Doing so is not a problem when the data is in memory, but accommodating the data from a large file in an array, for example, might be impossible. In those cases, you might want to read the file several times.

After the file is read completely, you close it by closing the `Scanner` object. You then open it again so that you can read the file from its beginning.

**Note: Sorting the data in a file**

Searching a data set for a particular entry is a common task. As Chapter 19 shows, searching sorted data can be faster than searching unsorted data. Chapters 15 and 16 discuss sorting the entries in an array. When your data is in a file, however, it might not fit entirely into an array: The file is simply too large for the computer's memory to accommodate it.

The merge sort, which Chapter 16 presents, can be adapted to sort the data in a file. Although the details of this modification are beyond our scope, its basic idea is not hard to understand. The sort reads a portion of the file into an array, sorts the array, and writes the sorted data to another file. This process is repeated for successive portions of the original file until all data has been processed. In the next phase, pairs of sorted blocks of data are read from the new file and merged into a larger block, which is then written to the original file. This merging continues until the second file is processed and is then repeated on the original file, and so on until all the data has been sorted. Note that merging involves small portions of sorted data at a time, so that only a small amount of memory is needed to sort a large file.

Changing Existing Data in a Text File

S2.18 Once we have created a text file, some of its data might need to be updated or corrected. Although we can add lines at the end of an existing text file, we cannot add them anywhere else. Moreover, we cannot delete lines, and in general, we cannot modify lines. However, as we read a text file, we can copy lines unchanged to another text file, skip lines that we no longer want to keep in the file, and write additional or modified lines to the new file. Changing the data in a file, therefore, is like reading one file and writing another. Most of the logic in a program that makes the changes must decide where and what changes are needed.

When the new file is complete, we can delete the old file and—if we like—give the new file the name of the old file. Although we can use the operating system to rename or delete a file after the program has run, our Java program can perform these tasks for us by using the class `File`. For example, to change the name of the existing file `MyData.txt`, we could write statements such as

```
File originalFile = new File("MyData.txt");
File newFile = new File("MyUpdatedData.txt");
originalFile.renameTo(newFile);
```

The `File` method `renameTo` changes the name of the file and returns a boolean value to indicate whether the change was successful. Although the previous example ignores this returned value, we could have written something like

```
if (originalFile.renameTo(newFile))
    System.out.println("Name change successful.");
else
    System.out.println("Attempted name change unsuccessful.");
```

Note that `renameTo` has a `File` object as its argument instead of a string containing the new name.

The `File` method `delete` deletes an existing file and returns either `true` or `false` to indicate whether the operation was successful. For example, to delete the previous file, we could write

```
originalFile.delete();
```



Programming Tip: Avoid a silent program

Without the `println` statements in the previous segment, the user would not know whether the name of the file was changed. Worse than this code is an entire program that processes a file without any visible output. Such a program is called a **silent program** and can bewilder its user. Did the program actually do anything? Always provide at least one message to the user to indicate the program's status.

Defining a Method to Open a Stream

S2.19 Imagine that we want to write a method that opens a file. We will open a text file for output, but the idea is also applicable to opening it for input and to a binary file. Our method has a `String` parameter that represents the file name. The client of this method could either read the file name from the user or use a literal for the file name. The method that follows creates an output stream, connects it to the given file, and returns the stream to the client.

```
public static PrintWriter openOutputTextFile(String fileName)
    throws FileNotFoundException, IOException
{
    PrintWriter toFile = new PrintWriter(fileName);
    return toFile;
} // end openOutputTextFile
```

We could invoke this method as follows:

```
PrintWriter toFile = null;
try
{
    PrintWriter toFile = openOutputTextFile("data.txt");
}
< Appropriate catch blocks are here >
```

and go on to use `toFile` to write to the file.

What if we had written the method as a `void` method, so that instead of returning an output stream, it had the stream as a parameter? The following method looks reasonable, but it has a problem:

```
// This method does not do what we want it to do.
public static void openFile(String fileName, PrintWriter stream)
    throws FileNotFoundException, IOException
{
    stream = new PrintWriter(fileName);
} // end openFile
```

Let's consider, for example, the following statements that invoke the method:

```
PrintWriter toFile = null;
try
{
    openFile("data.txt", toFile);
}
}
```

After this code is executed, the value of `toFile` is still `null`. The file that was opened by the method `openFile` went away when the method ended. The problem has to do with how Java handles arguments of a class type. These arguments are passed to the method as a memory address that cannot be changed. The object at the memory address normally can be changed, but the memory address itself cannot be changed. Thus, you cannot change the value of `toFile`.

This observation applies only to arguments of methods. If the stream variable is either a data field or declared locally within the body of the method, you can open a file and connect it to the stream and this problem will not occur. Once a stream is connected to a file, however, you can pass the stream variable as an argument to a method, and the method can change the file.

Binary Files

S2.20 Text files and binary files share some similarities. For example, before you can read or write any file, you open it, and when you are finished, you close it. The standard classes used for text files, however, differ from those we will use here for binary files, with one exception—the class `File`. This class, which was introduced in Segment S2.15, can be used with binary files as well as text files.

Anything that you can write to a text file can be written to a binary file. Binary files store a value of a primitive type in the same format and the same number of bytes as it is stored in the computer's primary memory. For example, every `int` value occupies four bytes, and every `double` value requires eight bytes. This is one reason why a binary file can be more efficient in its use of time and space than a text file.

The most commonly used stream classes for processing binary files are `DataInputStream` and `DataOutputStream`. Each class has methods to read or write data one byte at a time. These streams can also convert numbers and characters to bytes that can be stored in a binary file. They allow your program to write data to or read data from a binary file as if the data were not just bytes but either strings or items of any of Java's primitive data types. If you do not need to access your files via an editor, the easiest and most efficient way to read and write data to files is to use `DataInputStream` and `DataOutputStream` with a binary file.

Binary files also offer features not possible with a text file. For example, you can write entire objects to a binary file, without knowledge of their data, and read them back again. In contrast, writing an object to a text file requires you to write each of its data components. This task becomes even more complicated when a data field is itself an object.

While some binary files are read only via sequential access, just like text files, certain binary files offer **random access**. That is, you can access their data without having to first read all the data preceding it, as you would with a sequential-access file. In other words, you can access the data at position n much as you can access the data in an array at index n . However, we will not cover random-access files in this book.

In this supplement, we will use the suffix `.bin` when naming a sequential-access binary file.



Note: You can use a Java program to create a binary file on one computer and have it read by a Java program on another computer. Normally, you cannot use a text editor to read binary files.

Creating a Binary File of Primitive Data

Would you ever create or use a binary file of primitive data? Yes; files of statistical or scientific data, such as the hourly outdoor temperatures for a year in a certain location, are common. Let's begin by writing primitive data to a sequential-access binary file. When a binary file will contain only data of a primitive type, you typically will use the standard class `DataOutputStream` to create it.

S2.21 Opening a binary file for output. To write primitive data to a binary file, we'll use a stream of the class `DataOutputStream`. Like most of the classes we use for files, this one belongs to the package `java.io` of the Java Class Library. To open a binary file for output, we invoke the constructor of `DataOutputStream`. Like some of the classes we encountered earlier for text files, `DataOutputStream` has no constructor that accepts a file name as an argument. However, it does have a constructor whose argument can be an object of the class `FileOutputStream`, and

a constructor of `FileOutputStream` does accept a file name as an argument. Thus, if the name of a binary file is in the `String` variable `fileName`, we can open the file for output by writing statements such as

```
FileOutputStream fos = new FileOutputStream(fileName);
DataOutputStream toFile = new DataOutputStream(fos);
```

These statements create an output stream and connect it to the named file. The stream variable `toFile` references this stream. As is true of text files, when you open a binary file for output, your program always starts with an empty file. If the named file does not already exist, a new, empty file is created. However, if the named file already exists, its old contents will be lost.

S2.22 Example. `FileOutputStream`'s constructor will throw a `FileNotFoundException` if the file either does not exist and cannot be created or exists but cannot be opened. Since this exception is a checked exception, the constructor's invocation must appear within either a try block that is followed by an appropriate catch block or a method whose header lists this exception in a throws clause. Note that `DataOutputStream`'s constructor does not throw any exceptions. For example, we could write the following statements to open the binary file `data.bin` for output:



```
String fileName = "data.bin";
DataOutputStream toFile = null;
try
{
    FileOutputStream fos = new FileOutputStream(fileName);
    toFile = new DataOutputStream(fos);
}
catch (FileNotFoundException e)
{
    System.out.println("Cannot find, create, or open the file " + fileName);
    System.out.println(e.getMessage());
    System.exit(0);
}
```

We declare the variable `toFile` outside of the try block so that `toFile` is available outside of the try block.

S2.23 Using the class `File` when opening a binary file. Earlier, we introduced the class `File`. Recall that you can use methods from this class to test whether a file exists, can be read, or can be written. We can use `File` to make the same tests of a binary file. If a particular binary file exists and can be written, for example, we can use the `File` object that we created for the tests as the argument for a constructor of `FileOutputStream`. Thus, we might modify the previous statements as follows:

```
< Declarations of and assignments to fileName and toFile >
boolean okToWrite = true;
try
{
    File aFile = new File(fileName);
    if (aFile.canWrite())
    {
        FileOutputStream fos = new FileOutputStream(aFile);
        toFile = new DataOutputStream(fos);
    }
    else
        okToWrite = false;
}
< catch block is here >
< Statements that test okToWrite and act accordingly >
```


S2.24 Writing primitive data to a binary file. `DataOutputStream` has an output method for each primitive data type, as follows:

- `writeByte`—Writes the low-order 8 bits of its `int` argument
- `writeChar`—Writes the low-order 16 bits of its `int` argument as a Unicode character
- `writeShort`—Writes the low-order 16-bit integer of its `int` argument
- `writeInt`—Writes the 32-bit integer in its `int` argument
- `writeLong`—Writes the 64-bit integer in its `long` argument
- `writeFloat`—Writes the 32-bit real value in its `float` argument
- `writeDouble`—Writes the 64-bit real value in its `double` argument
- `writeBoolean`—Writes the boolean value in its `boolean` argument

Each method is a void method, has one parameter, and can throw an `IOException` in case an error occurs while writing a value to the file. Note that each of the methods `writeByte`, `writeChar`, `writeShort`, and `writeInt` has an `int` parameter, even though the first three write a smaller value to the file.

S2.25 Closing a binary file. You use the method `close` to close a binary file after writing it. Unlike the `close` method for a text file, this method can throw an `IOException` if an error occurs while closing the file.

S2.26 Appending to an existing binary file. To add more data to the end of the data already in an existing binary file—that is, to append data to the file—you would revise the statements given in Segment S2.21 that open the file by calling `FileOutputStream`'s constructor as follows:

```
FileOutputStream fos = new FileOutputStream(fileName, true);
```

The second parameter of the constructor is a boolean value that indicates whether to append data to an existing file. If the named file does not exist already, Java will create an empty file of that name and write the output to this empty file. However, if the file does exist, additional output to the file will be placed after the old contents of the file.

S2.27 Example: Creating a binary file of random integers. Let's define a class of static methods that deal with binary files. One of these methods can create a file of random integers. Suppose that the client of our method passes it the name of the file and the desired number of random values the file should contain. The method's header then can have the following form:



```
public static returnType createRandomIntegerFile(String fileName, int howMany)
```

As we noted previously in this supplement, we must concern ourselves with checked exceptions when we create this file. We must either handle the exceptions within the body of the method or add a `throws` clause to the method's header. The choice we make will affect the return type of our method.



Design Decision: Should a method handle exceptions or pass them on to its client?

If we pass exceptions on to a method's client, the method will be simpler to write and can be a void method. However, its client will have to worry about the exceptions. Instead, we could handle the exceptions ourselves but tell the client what has happened by returning a code. The resulting method definition will be more involved, but its use will be convenient.

Listing S2-3 shows the definition of the static method `createRandomIntegerFile` within a class `BinaryFileOperations`. Notice how we close the file within a `finally` block, which Java Interlude 3 introduces. Its use not only ensures that the file is closed but also allows us to have two separate catch blocks for an `IOException`. In this way, we can tell whether an `IOException` is caused by the method `writeInt` or the method `close`.

LISTING S2-3 The static method `createRandomIntegerFile` in the class `BinaryFileOperations`

```

1  import java.io.DataOutputStream;
2  import java.io.FileNotFoundException;
3  import java.io.FileOutputStream;
4  import java.io.IOException;
5  import java.util.Random;
6
7  public class BinaryFileOperations
8  {
9      /** Writes a given number of random integers to the named binary file.
10         @param fileName The file name as a string.
11         @param howMany The positive number of integers to be written.
12         @return An integer code indicating the outcome of the operation:
13         0 = Success; > 0 = Error: opening (1), writing (2), or closing (3) the file.
14     */
15     public static int createRandomIntegerFile(String fileName, int howMany)
16     {
17         int resultCode = 0;
18         Random generator = new Random();
19         DataOutputStream toFile = null;
20         try
21         {
22             FileOutputStream fos = new FileOutputStream(fileName);
23             toFile = new DataOutputStream(fos);
24
25             for (int counter = 0; counter < howMany; counter++)
26             {
27                 toFile.writeInt(generator.nextInt());
28             } // end for
29         }
30         catch (FileNotFoundException e)
31         {
32             resultCode = 1;    // Error opening file
33         }
34         catch (IOException e)
35         {
36             resultCode = 2;    // Error writing file
37         }
38         finally
39         {
40             try
41             {
42                 if (toFile != null)
43                     toFile.close();
44             }
45             catch (IOException e)
46             {
47                 resultCode = 3; // Error closing file
48             }
49             return resultCode;
50         }
51     } // end createRandomIntegerFile
52 } // end BinaryFileOperations


```

Reading a Binary File of Primitive Data

S2.28 Once we have created a binary file of primitive values by using an output stream of the class `DataOutputStream`, we will use the class `DataInputStream` to read the file. The details of opening and closing this file for input are analogous to the ones we saw earlier for output. For each method in `DataOutputStream` that we used to write a primitive value to the file, `DataInputStream` has an analogous method to read the value from the file. Those methods are as follows:

- `readByte`—Reads and returns the next byte in the file as a `byte` value
- `readChar`—Reads and returns the next two bytes in the file as a `char` value
- `readShort`—Reads and returns the next two bytes in the file as a `short` value
- `readInt`—Reads and returns the next four bytes in the file as an `int` value
- `readLong`—Reads and returns the next eight bytes in the file as a `long` value
- `readFloat`—Reads and returns the next four bytes in the file as a `float` value
- `readDouble`—Reads and returns the next eight bytes in the file as a `double` value
- `readBoolean`—Reads and returns the next `boolean` value in the file

Each of these methods can throw either an `EOFException` when the end of the file is encountered or an `IOException` if an error occurs during the read operation. These checked exceptions must be handled.

S2.29  **Example: Reading a binary file of integers.** Listing S2-4 shows the definition of the static method `displayBinaryFile` within the class `BinaryFileOperations`. The method is similar to `createRandomIntegerFile`, as given in Listing S2-3, with respect to its return value and how it opens the file, closes the file, and handles exceptions. This method has only one parameter, the file name as a string, and displays all of the integers in the file. Notice that `createRandomIntegerFile` did not write a sentinel after it wrote the last integer to the file.

LISTING S2-4 The static method `displayBinaryFile` in the class `BinaryFileOperations`

```

1  import java.io.DataInputStream;
2  import java.io.DataOutputStream;
3  import java.io.EOFException;
4  import java.io.FileInputStream;
5  import java.io.FileNotFoundException;
6  import java.io.FileOutputStream;
7  import java.io.IOException;
8  import java.util.Random;
9  /**
10     A class of methods that create and display a binary file of random integers.
11     @author Frank M. Carrano
12  */
13  public class BinaryFileOperations
14  {
15      < The method createRandomIntegerFile, as given in Listing S2-3, appears here. >
16
17      /** Displays all integers in the named binary file.
18          @param fileName The file name as a string.
19          @return An integer code indicating the outcome of the operation.
20              0 = Success; > 0 = Error opening (1), reading (2), closing (3) the file.
21      */
22      public static int displayBinaryFile(String fileName)
23      {
24          int resultCode = 0;
25          DataInputStream fromFile = null;

```

```

26     try
27     {
28         FileInputStream fis = new FileInputStream(fileName);
29         fromFile = new DataInputStream(fis);
30
31         while (true)
32         {
33             int number = fromFile.readInt();
34             System.out.println(number);
35         } // end while
36     }
37     catch (FileNotFoundException e)
38     {
39         resultCode = 1;    // Error opening file
40     }
41     catch (EOFException e)
42     {
43         // Normal occurrence since entire file is read; ignore exception
44     }
45     catch (IOException e)
46     {
47         resultCode = 2;    // Error reading file
48     }
49     finally
50     {
51         try
52         {
53             if (fromFile != null)
54                 fromFile.close();
55         }
56         catch (IOException e)
57         {
58             resultCode = 3; // Error closing file
59         }
60         return resultCode;
61     }
62 } // end displayBinaryFile
63 } // end BinaryFileOperations

```

When you read a binary file of primitive values, you can detect the end of the file without checking for a sentinel at its end. You do so by catching the `EOFException` that the read methods throw when the end of the file is reached. You treat the exception as an expected occurrence, not as a mistake.

For example, the method `displayBinaryFile` contains an infinite loop to read all of the integers in the file. The entire loop—highlighted in the listing—is within a `try` block, so we can handle the exceptions. `DataInputStream`'s method `readInt` can throw either an `EOFException` when it attempts to read beyond the end of the file or an `IOException` if an error occurs during a read operation. Since the end of the file is an expected and normal occurrence in this example, one of the catch blocks after the previous `try` block is

```

catch (EOFException e)
{
    // Normal occurrence since the entire file is read; ignore exception
}

```

Thus, we catch but ignore an `EOFException`.

Strings in a Binary File

Although strings are objects, you can use `DataOutputStream` to write one to a binary file as a sequence of characters and `DataInputStream` to read these characters from the file as a string.

S2.30 Writing strings to a binary file. `DataOutputStream` has another method, `writeUTF`, that writes a string to a file as a sequence of characters in a machine-independent way:

- `writeUTF`—Writes the characters in its `String` argument using an encoding called **UTF**, or **Unicode Transformation Format**

Like the other methods of `DataOutputStream` that we introduced previously, `writeUTF` is a void method, has one parameter, and can throw an `IOException`.



Note: What is UTF?

Recall that Unicode is a representation of characters by unique integers. However, Unicode does not specify how many bytes one should use to store each integer in memory. UTF makes this specification. Several UTF encoding schemes exist. **UTF-8** represents each character as a sequence of between one and four bytes. UTF-8 uses one byte for each ASCII character, which is sufficient for the characters in the English language. Unicode, however, is more universal and provides many more characters. For such characters, UTF-8 uses two, three, or four bytes each. The method `writeUTF` uses a slight variation of UTF-8, known as modified UTF-8. Thus, the method `writeUTF` can represent all Unicode characters, but it saves file space when ASCII characters are written.

UTF-16 uses either two or four bytes to represent a character. Java uses UTF-16, but a char value occupies only two bytes, and the method `writeChar` writes two bytes to a file. For strings and char arrays, characters requiring four bytes each are represented as pairs of char values.

UTF-32 uses four bytes for each character. Although a fixed number of bytes per character is convenient, UTF-32 is space inefficient.

S2.31 Reading strings from a binary file. To read a value previously written to the file by the method `writeUTF`, we use the following method of `DataInputStream`:

- `readUTF`—Reads and returns the next UTF string in the file as a `String` object

This method can throw one of two possible checked exceptions: a `UTFDataFormatException` if the bytes read are not a UTF string, or an `IOException` if an error occurs during the read operation.



Programming Tip: A binary file can contain data of differing types, and you can use `DataInputStream` to read such a file. You must be careful, however, to match the data with the appropriate read methods. If a data item in the file is not of the type expected by the reading method, the result is likely to be wrong. For example, if your program writes an integer using `writeInt`, any program that reads that integer should read it using `readInt`. If you instead use `readDouble`, for example, your program will misbehave. This and subsequent read operations will most likely not match the file and will be incorrect.



Programming Tip: Binary files and text files encode their data in different ways. A stream that expects to read a binary file, such as a stream in the class `DataInputStream`, will have problems reading a text file. If you attempt to read a text file with a stream in the class `DataInputStream`, your program either will read “garbage values” or will encounter some other error condition.

**Programming Tip: Check for the end of a file**

When reading from a file, your program should check for the end of the file and do something appropriate when it reaches it. If your program tries to read beyond the end of a file, it may enter an infinite loop or end abnormally. Even if you think your program will not read past the end of the file, you should provide for this eventuality just in case things do not go exactly as you planned.

**Programming Tip: Ways to check for the end of a file**

Here are some possible ways to test for the end of a file:

- Catch an `EOFException`, if the read method that you use throws one. When reading from a binary file, the methods in `DataInputStream` throw an `EOFException` when they try to read beyond the end of a file.
- Test for a special value—the sentinel—if one has been written at the end of the file. Your program then can stop reading when it reads the sentinel value. For example, you could use a negative integer as a sentinel value at the end of a file of nonnegative integers. Using a sentinel value, however, restricts your data to values other than the sentinel.

**Note: Path names**

When passing a file name as an argument to a constructor of classes like `File`, you can use a simple file name. Java assumes that the file is in the same directory (folder) as the one that contains the program. You also can use a full or relative path name. A full path name gives not only the name of the file, but also tells what directory (folder) the file is in. A relative path name gives the path to the file starting in the directory that contains your program. Paths depend on your operating system rather than the Java language.

Note that Java provides an interface `Path` and a class `Paths` whose instances are like path names, but cannot be passed to `File`'s constructors. However, a method in `Paths` returns a `Path` object whose method `toFile` returns an instance of `File` that represents the specified path.

Object Serialization

S2.32 You have seen how to write primitive values and strings to a file, and how to read them again. How would you write and read objects other than strings? You could, of course, write an object's data fields to a file and invent some way to reconstruct the object when you read the file. When you consider that a data field could be another object that itself has an object as a data field, completing this task sounds formidable.

Fortunately, Java provides a way—called **object serialization**—to represent an object as a sequence of bytes that can be written to a binary file. This process will occur automatically for any object that belongs to a class that implements the interface `Serializable`. This interface, which is in the package `java.io`, is empty, so you have no additional methods to implement. Adding only the words `implements Serializable` to the class's definition is enough.

For example, we could begin a class `Student` as follows:

```
import java.io.Serializable;
public class Student implements Serializable
{
    . . .
}
```

The `Serializable` interface tells the compiler that `Student` objects can be serialized. Since `Serializable` appears only once in the definition of the class `Student`, you could conveniently omit the `import` statement and begin the class as follows:

```
public class Student implements java.io.Serializable
```

To serialize an object and write it to a binary file, you use the method `writeObject` from the class `ObjectOutputStream`. To read a serialized object from a binary file, you use the method `readObject` from the class `ObjectInputStream`.

S2.33 Example. To serialize the `Student` object `aStudent` and write it to a binary file, we would write the following statements within one or more try blocks:



```
FileOutputStream fos = new FileOutputStream(fileName);
ObjectOutputStream toFile = new ObjectOutputStream(fos);
. . .
toFile.writeObject(aStudent);
```

Any objects that are data fields of `aStudent` must also belong to a class that implements `Serializable`. Such objects are serialized when `aStudent` is serialized. Many classes in the Java Class Library—including `String`—implement `Serializable`.

To read the `Student` object from the binary file, we would write

```
FileInputStream fis = new FileInputStream(fileName);
ObjectInputStream fromFile = new ObjectInputStream(fis);
. . .
Student joe = (Student)fromFile.readObject();
```

within one or more try blocks.

S2.34 We will call a class serializable if all of the following are true:

- It implements the interface `Serializable`
- Its data fields are either primitive values or objects of a serializable class
- Its direct superclass, if any, is serializable and defines a default constructor

Any subclass of a serializable class is serializable.

We will call an object serializable if it belongs to a class that is either serializable or a subclass of a serializable class.



Note: Object serialization

As we mentioned, object serialization is the process Java uses to represent an entire object as a sequence of bytes. Any serialized object receives a serial number as it is written to a binary file. If later we write the same object to the file, only its serial number is written. This approach saves space on the file, as the object's data is not written again. Moreover, when the first instance of the object is read from the file, both its data and serial number are read and a new object is created within the program. However, when a duplicate serial number is read, Java creates a reference to the object—that is, an alias—instead of a duplicate object.

If an object is not serializable, you cannot use `writeObject` to write it to a binary file. Instead, you must write each of its data fields to the file.



Security Note: Should all objects be serialized to save file space?

The designers of Java did not think so. Some objects simply should not be saved in a file. Perhaps more importantly, serialized objects are easy to access, and for security reasons, easy access is not always a good idea. Thus, the interface `Serializable` gives programmers the option of allowing or preventing the serialization of objects of their classes.



Note: Strings are serializable objects

We know that strings are objects of the class `String`. Since `String` implements the interface `Serializable`, we can use the method `writeObject` instead of `writeUTF` to write strings to a binary file.

S2.35 Arrays. Java treats arrays as objects, and they are serializable. You can use `writeObject` to write an array to a binary file, and you can use `readObject` to read it from the file. For example, suppose that `group` is an array of `Student` objects. If `toFile` is an instance of `ObjectOutputStream` that is associated with a binary file, we can write the array to that file by executing the statement

```
toFile.writeObject(group);
```

After creating the file, we can read the array by using the statement

```
Student[] myArray = (Student[])fromFile.readObject();
```

where `fromFile` is an instance of `ObjectInputStream` that is associated with the file that we just created.



Note: The classes `ObjectOutputStream` and `ObjectInputStream` have methods to write and read primitive values when working with a binary file. You can consult the online documentation for the Java Class Library to learn about these methods.



Note: Checked exceptions thrown by a selection of classes and methods in the Java Class Library (All are in the package `java.io`, except for `ClassNotFoundException`)

| Class and Methods | Exceptions | Class and Methods | Exceptions |
|---|---|---|---|
| <code>DataInputStream</code> Constructor <code>close</code> <code>readInt</code> , other <code>read</code> methods | None <code>IOException</code> <code>EOFException</code> , <code>IOException</code> | <code>ObjectInputStream</code> Constructor <code>close</code> <code>readObject</code> | <code>IOException</code> , <code>StreamCorrupted-Exception</code> <code>IOException</code> <code>java.lang. -</code> <code>ClassNotFoundException</code> , <code>InvalidClassException</code> , <code>OptionalDataException</code> , <code>StreamCorrupted-Exception</code> , <code>IOException</code> |
| <code>DataOutputStream</code> Constructor <code>close</code> , <code>flush</code> , <code>writeInt</code> , other <code>write</code> methods | None <code>IOException</code> | <code>ObjectOutputStream</code> Constructor, <code>close</code> <code>writeObject</code> | <code>IOException</code> <code>InvalidClass Exception</code> , <code>IOException</code> , <code>NotSerializable-Exception</code> |
| <code>File</code> Constructor, <code>canRead</code> , <code>canWrite</code> , <code>delete</code> , <code>exists</code> , <code>getName</code> , <code>getPath</code> , <code>length</code> | None | <code>PrintWriter</code> Constructor <code>close</code> , <code>flush</code> <code>print</code> , <code>println</code> | <code>FileNotFoundException</code> <code>IOException</code> None |
| <code>FileInputStream</code> Constructor <code>close</code> | <code>FileNotFoundException</code> <code>IOException</code> | <code>Scanner</code> Constructor Other methods | <code>FileNotFoundException</code> None |
| <code>FileOutputStream</code> Constructor <code>close</code> | <code>FileNotFoundException</code> <code>IOException</code> | | |