Companion Chapter 2

File I/O in Java

What You Will Learn in This Chapter

Working with file input and output (I/O) is a skill that most developers will need at some point in their careers. For information systems professionals learning to develop in the Java language, file I/O skills are very important. Files present a great way to store data in a more permanent fashion, provide data backups, and enable data portability between Java applications. The Java language provides several easy-to-use classes to help developers get up to speed quickly in Java's handling of file I/O.

Specifically, this chapter will help you do the following:

1. Learn about the classes needed to write to and read from plain-text data files
2. Learn the basic concepts every programmer should consider when working with file I/O
3. Learn how the `Scanner` class can be used for more than just console-based user input
4. Learn how exception handling relates to file I/O and how it enhances the stability of your application
5. Learn how binary data files differ from plain text, their benefits, and the classes used to interact with them
6. Learn how to write primitive data types out to and read their values from a binary data file
7. Learn how Java allows for complex data-type objects to be written to and retrieved from a binary data file
8. Learn how the `Serializable` interface impacts the use of class objects in binary data files

Opening Scenario

Your project manager begins the latest meeting by stating, "The client's data is all over the place. We can consolidate and export it into a clean format, but then it'll be your job to import it into the new system!" You have joined in on several meetings with your colleagues who are handling the discovery and mapping of the data needs of the mom-and-pop grocery store client. Their assessment is correct: the client has data in spreadsheets and third-party desktop database files, and some of it is in text files typed manually! Both you and several of your fellow team members know that demonstrating the prototype system with data from their actual day-to-day operations will help "sell" the development efforts. With so much else going on, you quickly realize that typing in a lot of their data manually will not be effective. You'll need to write code that imports this data: "Data quality will be a concern, but we'll try to standardize and correct as many problems as possible before sending it over to you," you tell them.

The client had also purchased several analytics software packages that will be, for now, separate from the system your team is building. The grocery inventory system will need the ability to export data into files that can be sent over to the third-party system. Your project manager has already specified your role in this: "I'll have the user interface team design the forms and the part of the system that will detail the export operation. I just need you to have a good grasp on the code that will make it possible." Having worked with file input/output (I/O) operations before in other languages, you are familiar with some of the basic concepts. Working with file I/O in Java will be new to you. Once again you dive into your studies, intent on being ready to implement this functionality for the team . . .

CC2.1 Basics of File I/O in Java

Learning how Java handles file input/output (I/O) activities begins with a discussion of the `File` class.[*] Conveniently, Java handles file access in a very object-oriented way. Instance objects of the `File` class effectively act as object wrappers, containing information about the files on your hard disk. `File` class instance objects are used by the other Java file I/O classes that perform the actual reading and writing of data from files, so it helps to understand how this class works.

Basics to keep in mind: In general, you need to know three things about a file before you can work with it (and these apply no matter the programming language you work in, not just Java):

- The name of the file
- The location of the file
- The contents and structure of the data stored within the file

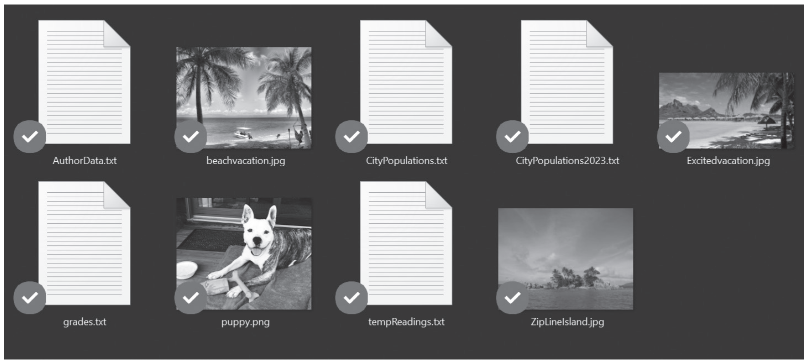


Figure CC2.1. Files Can Contain a Variety of Types, Quantities, and Structures of Data

Sources: "Tropical Island—Seychelles" by tiarescott is licensed under CC BY 2.0, https://www.flickr.com/photos/80403443@N00/33515808; "Wheeler Island One of the 20 Tropical Island Just Offshore from Mission Beach" by Paul from www.Castaways.com.au is licensed under CC BY 2.0, https://www.flickr.com/photos/54113234@N05/5031478376.

Instance objects of the `File` class can store information on the name and location of the file itself, useful because other Java file I/O classes know how to read and use this information from `File` objects. The structure and contents of a file are determined by you, the developer. Generally, you will have to or can discover this information based on the project you are working on or the problem you are trying to solve. For example, a file may contain a list of students and their grades, the populations of various cities, or information on time, date, location, and temperature readings for a weather forecasting office. One problem developers often face is knowing the quantity of data in a file. You may know ahead of time that a file contains student names and their grades but not know *how many* students are listed in the file. Your code's logic must account for any unknowns like this and stay flexible in these situations.

The location of a file can be represented in two ways:

- **Absolute path:** This is where you use the fully qualified location of a particular file on a computer system. For example, a text file containing student grades may have an absolute file path of "`C:\data\grades\Spring2023\Class0001.txt`" in the Windows file system or "`/Users/ezellj/Library/data/grades/Spring2023/Class0001.txt`" in Mac OS X and other similar Unix OS file systems. In these two cases, the "`C:\`" and the very first "`/`" for Unix-flavored systems are called the **root elements** of the file system. This helps Java know that you are using an absolute file path to locate a file.
    - *Note:* Because the Windows file system uses the backslash ("\") to separate out the folders in a file path, in a Java application, you will need to **escape** the backslash with another backslash (see chapter 3) when using an absolute path. An example of this would be "`C:\\data\\grades\\Spring2023\\Class0001.txt`."
- **Relative path:** More commonly used, this allows the developer to programmatically reference files that are located relative to the program that is executing. For example, in a Java project, "`data/Class0001.txt`" would refer to a file called "`Class0001.txt`" that is saved in a "`data`" folder that is local to the current executing directory of your program. This is helpful in cases where your application might create a directory and then choose that directory to write to / read from / store all external files it might use. Take particular note that the relative file path did not start with a root element. Leaving the root element off helps Java

[*] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/File.html.

know this file path should be processed in a relative manner (relative to where the Java program is executing). The Oracle documentation for the `File` class mentions that the file I/O classes we will explore in this chapter assume a relative file path unless an absolute one is given.

Using the `File` class: You can test out the differences between the relative and absolute file paths easily to fully understand their difference. Many of the examples throughout this chapter can be run in a standard Java `main()` class file, and it would be helpful to create one here at the start.

In order to use the Java file I/O classes, you need to first add the appropriate import to your `main()` class file. Add the following two, one for Java file I/O and the other so that the `Scanner` class can be used later in this chapter:

Code Snippet CC2.1

```
import java.util.*; // to use Scanner
import java.io.*;   // for Java File I/O
```

Instantiate a new object of the `File` class like so (code added to the `main()` method):

```
// in main()
File nameFile = new File("nameFile.txt");
```

Figure CC2.2 shows both the logical and the physical contents of the Java project folder in Apache NetBeans after running the program with this one line of code. This project structure and contents will be similar to other IDEs such as Eclipse. Notice that no file called "`nameFile.txt`" currently exists. The `File` class **does not actually create the file**; it is merely an instance object that stores information about where the file is currently or is planned to be in the future.

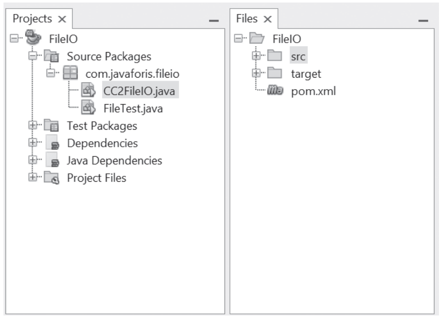


Figure CC2.2. Logical and Physical File Views in Apache NetBeans

Interestingly, the `File` class can also be used to create a directory. Consider the following:

Code Snippet CC2.2

```
// in main()
File nameFile = new File("nameFile.txt");
File dataDir = new File("./dataDir");

dataDir.mkdir();
```

Notice the relative file path prefixed to the "dataDir" name in the `File` instance object. When using a relative file path, prefixes can be used:

- **Current directory:** You can reference the current working/executing directory by using the prefix `./`.
- **Directory above the current:** If you need to reference something in the directory **above the one you are currently executing your application in**, then use the prefix `../`.
    - For example, you can chain these prefixes together to reference directories as high above the current as you need. Two above the current would be `../../`.

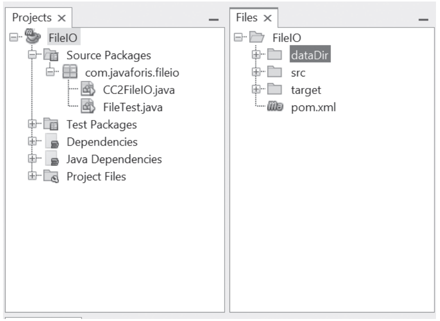


Figure CC2.3. Directory Created by Invoking the `.mkDir()` Method on Our File Object

After running this code, figure CC2.3 shows the file contents of our project. Notice that in Apache NetBeans, the "Projects" view shows the **logical** view of the Java project, while the "Files" view shows the **physical** contents of the files in the Java project. Other IDEs like Eclipse will show similar views.

Though the Oracle documentation gives a complete listing of the methods available for the File class, table CC2.1 lists some of the more important methods to be aware of and a brief description of each (besides the overloaded constructor):

Table CC2.1. Select Methods of the File Class

Return type and method name	Brief description
`boolean .canRead()`	Returns `true` if the file (at the file path) can be read from
`boolean .canWrite()`	Returns `true` if the file (at the file path) can be written to
`boolean .delete()`	Returns `true` if the file or the directory (which must be empty) was successfully deleted
`boolean .exists()`	Returns `true` if the file (at the file path) exists
`String .getAbsolutePath()`	Returns a `String` that contains the full, absolute path of a file indicated by this `File` object
`long .lastModified()`	Returns back the milliseconds-since-1/1/1970 timestamp of when the file was last modified
`String[] .list()`	Returns a `String[]` array of all the files and directories located within the directory indicated by this `File` instance object
`boolean .mkdir()`	Returns `true` if the directory represented by this `File` object was successfully created

If your application expects another program or process to create a file at a certain location before using it, you can use the `.exists()` method to test for the existence of the file and `.canRead()` to ensure your application has the appropriate level of permissions to access that file location.

I/O exceptions and writing to a text file: To further understand how the relative file path works in a Java project, first you can try creating and writing out to a plain-text file. The `PrintWriter` class can be used to easily write out to a file.* The class has several overloaded versions of its constructor, with three in particular that will be discussed here:

- `PrintWriter(File f)`: This version takes a reference to a `File` class instance object, similar to what we created earlier.
- `PrintWriter(String s)`: You can include a `String` or a `String` literal directly as a parameter of the `PrintWriter` constructor, specifying the file path within the `String` value.
- `PrintWriter(OutputStream o)`: Later in this chapter, the subclasses of the `OutputStream` class will be examined. These come in handy for both **appending data** to a file and writing out **binary data to a `.dat` file**.

The `PrintWriter` class instance objects can have the `.print()`, `.println()`, and `.printf()` methods invoked upon them as you write out to a file. Also, `PrintWriter` will create a file when it writes if the file **does not already exist** and will overwrite a file if it does. The use of an `OutputStream` instance object in the `PrintWriter` constructor (as you will see later) allows appending to a file instead of destroying and overwriting it with new data.

Consider the following code:

Code Snippet CC2.3

```
// in main()
File nameFile = new File("myName.txt");
PrintWriter fileOut = new PrintWriter(nameFile);

fileOut.print("Your Name");
fileOut.close(); // Always close your file connections!
```

* https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/PrintWriter.html.

If your IDE can show compiler and syntax errors as you type, then you will see that the invocation of the constructor is underlined with an error:

```
unreported exception FileNotFoundException; must be caught or declared to be thrown
```

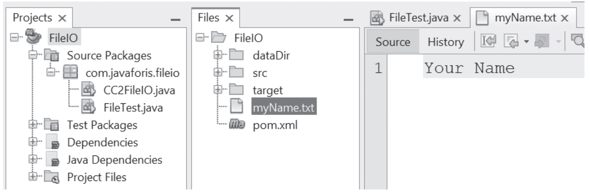
Not good. Recall the discussion in chapter 10 where some Java code has a higher risk of an error occurring during execution than others. Code that interacts with some *external* resources like a database connection, network communication, multithreading, or, in this chapter's case, file I/O could encounter an error, since your code has no direct control over those external resources. In these cases, there are **checked exceptions** that your code must explicitly handle to prevent errors in your application. In this case, you need to surround any file I/O activities in a `try…catch` statement to handle any I/O exceptions that occur. `IOException` class objects or any of their subclasses (`FileNotFoundException` is one of these subclasses) can be "caught," or superclass `Exception` objects can be caught in a wide net as well. The earlier code can be modified to handle checked exceptions:

Code Snippet CC2.4

```
// in main()
// try…catch statement wrapping the File I/O code
try
{
    File nameFile = new File("myName.txt");
    PrintWriter fileOut = new PrintWriter(nameFile);

    fileOut.print("Your Name");
    fileOut.close(); // Always close your file connections!
}
catch (IOException ioex)
{
    System.out.println(ioex.toString());
}
```

Better! When this code is run, the new file should appear in your project as shown in figure CC2.4. Double-clicking the file will open it in the editor of your IDE, and you should see the String "Your Name" in the file in plain text. Notice the invocation of `.close()` at the end of the `try` block's logic: **always close file connections when you are done**. This way, you prevent your application from causing errors for the operating system or other applications that may need to access a specific file.

Modify this example further, and add the use of the `.println()` method instead of `.print()`:



```
Your Name
```

Figure CC2.4. Plain-Text File Created in Project Directory and Open in the Editor Widow

Code Snippet CC2.5

```
// in main()
try
{
    File nameFile = new File("myName.txt");
    PrintWriter fileOut = new PrintWriter(nameFile);

    fileOut.println("Your Name");
    fileOut.println("My Name");
    fileOut.close(); // Always close your file connections!
}
catch (IOException ioex)
{
    System.out.println(ioex.toString());
}
```

This will produce the following file contents:

```
Your Name
My Name
```

Notice that both `String` values are still written out whole, but a new-line character (not visible) is added to the end of each line due to the call to `.println()`, which outputs to the file the same way as it does when we output to the console.

Reading in from a text file: Luckily, a familiar class can be used to easily read in from a plain-text file: the `Scanner` class! All of the `.next____()` methods available in Scanner will work for reading in data from a file. These `Scanner` methods will work the same way as they do for console input also (see chapter 0). The `File` instance object reference is provided instead of the reference to the `System.in` member.

Further modify the code by using Scanner to read back in the data just written out to the file (in **bold**):

Code Snippet CC2.6

```
// in main()
try
{
    File nameFile = new File("myName.txt");
    PrintWriter fileOut = new PrintWriter(nameFile);

    fileOut.println("Your Name");
    fileOut.println("My Name");
    fileOut.close(); // Always close your file connections!

    Scanner fileIn = new Scanner(nameFile);
    System.out.println(fileIn.nextLine());
    System.out.println(fileIn.nextLine());
    fileIn.close();
}
catch (IOException ioex)
{
    System.out.println(ioex.toString());
}
```

Executing this code will produce the following on the console, exactly matching what was stored in the text file:

```
Your Name
My Name
```

What if you accidentally try to read in more data than is present in the text file? You know that there are only two names present in the data file. Consider the following alteration of the earlier code (alteration in **bold**):

Code Snippet CC2.7

```
    Scanner fileIn = new Scanner(nameFile);
    // . . .
    System.out.println(fileIn.nextLine());
    System.out.println(fileIn.nextLine());
    System.out.println(fileIn.nextLine());
    fileIn.close();
    // . . .
```

This extra attempt to read in from the file assumes that there are three names. As you will see later, this will often cause an end-of-file exception to be thrown. In this case, the `Scanner` method `.nextLine()` will throw a `NoSuchElementException` to indicate there was no more data to be read in:

```
Your Name
My Name
Exception in thread "main" java.util.NoSuchElementException: No line found
   at java.base/java.util.Scanner.nextLine(Scanner.java:1651)
   at com.javaforis.fileio.FileTest.main(FileTest.java:23)
Command execution failed.
```

The first two file reads still work, though the third causes the exception to be thrown. You can use exception chaining to catch any exceptions not under the `IOException` family tree and keep the application from crashing:

Code Snippet CC2.8

```
    // . . .earlier try block logic
}
catch (IOException ioex)
{
    System.out.println(ioex.toString());
}
catch (NoSuchElementException eofex)
{
    System.out.println("[End of File Reached!]");
}
catch (Exception ex)
{
    System.out.println(ex.toString());
}
```

The error occurs but now is caught and handled, and the program can continue executing additional logic. Your understanding of the file's contents and structure can help prevent errors like this.

Example usage—student grades: An additional example will help you understand the usage and behavior of file I/O in Java better. Note that **first** this example will model a situation where you know ahead of time how many records are contained within a data file. **This is done first for learning purposes but is not a real-world expectation!** Following this, the example will be modified for the much more realistic scenario where you do **not** know ahead of time how many records an incoming data file may contain.

Student grades—record quantity known ahead of time: Create a Java `main()` class application called `StudentData.java` that does the following:

- Writes twenty student grades out to a text file called `studentdata.txt`. Each record will contain a test name for a student ("Student 1," "Student 2," etc.) and a random exam grade between 40 and 100.
- The program will read in the twenty values from `studentdata.txt` back and calculate the minimum, maximum, and average grades. It will print these statistics along with the name of the student that achieved the minimum and maximum, respectively.

The first task is to set up the application for the write out to the file. This code will look like the following:

Code Snippet CC2.9

```
// in main()
String studentMinName = "";
String studentMaxName = "";
double minGrade = 100.0;
double maxGrade = 0.0;
double gradeAverage = 0.0;
int numOfLoops = 20;

try
{
    File studentData = new File("studentdata.txt");
    PrintWriter fileOut = new PrintWriter(studentData);

    // Write out 20 random grades to the file
    for (int i = 1; i <= numOfLoops; i++)
    {
      fileOut.print("Student " + i + " ");
      fileOut.printf("%.1f\n",40 + (Math.random() * 61)); // 40 to 100
    }

    fileOut.close();

    // File read in code will go here later

}
catch (IOException ioex)
{
    System.out.println(ioex.toString());
}
catch (NoSuchElementException eofex)
{
    System.out.println("[End of File Reached!]");
}
catch (Exception ex)
{
    System.out.println(ex.toString());
}
```

Notice that the writing of each record to the file happens in two lines: First the name "Student" is printed to the file with the current loop iterator value added along with a space. In the next print, the `.printf()` method of the `PrintWriter` class is used to format the randomly generated score to one decimal place before writing it out to the file. Finally, an escaped new-line character is used in the format specifier (chapter 3) to ensure the next record is printed to the next line in the file. Executing this code will generate the contents of the file similar to this (yours may differ, since the grades are randomly generated):

```
Student 1 81.8
Student 2 80.6
Student 3 60.2
Student 4 82.8
Student 5 49.6
. . . some records omitted for brevity
Student 19 60.5
Student 20 78.2
```

The challenge here (and one you might face with real-world raw data) is the "name" of the student partially consists of a number separated from the rest of the name by a space. If you remember, the `Scanner` methods generally (except for `.nextLine()`) ignore blank spaces and new-line characters.

Since you want to find the minimum, maximum, and average values from this file, the pseudocode for this logic and this know-record-quantity-ahead-of-time situation will look like this:

```
while loop count <= 20
   Read in student name
   Read in grade
      gradeAverage += grade  // sum
   if grade < current minimum
      current minimum = grade
      minimum grade student name = student name
   if grade > current maximum
      current maximum = grade
      maximum grade student name = student name
end of loop
gradeAverage /= number of loops
print statistics
```

This is pretty good pseudocode, as it will probably be a line-for-line translation into actual Java code. One implementation may look like this (add this code where the comment note was placed in the first half):

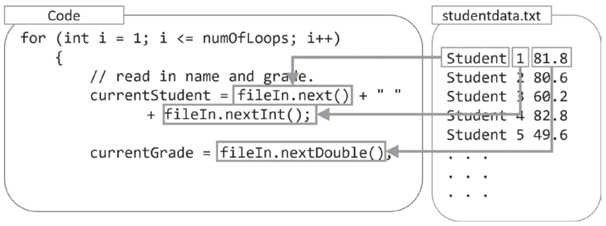
Code Snippet CC2.10

```java
// . . .
// File read in code
Scanner fileIn = new Scanner(studentData);
String currentStudent = "";
double currentGrade = 0.0;
for (int i = 1; i <= numOfLoops; i++)
{
    // read in name and grade.
    currentStudent = fileIn.next() + " " + fileIn.nextInt();
    currentGrade = fileIn.nextDouble();

    // Build the sum
    gradeAverage += currentGrade;

    // Compare to minimum and maximum.
    if (currentGrade <= minGrade)
    {
        minGrade = currentGrade;
        studentMinName = currentStudent;
    }

    if (currentGrade >= maxGrade)
    {
        maxGrade = currentGrade;
        studentMaxName = currentStudent;
    }
} // end of for loop

// calculate average
gradeAverage /= numOfLoops;

// print statistics
System.out.println("Student with highest grade of "
        + maxGrade + " was " + studentMaxName);
System.out.println("Student with lowest grade of "
        + minGrade + " was " + studentMinName);
System.out.printf("Average Grade Was: %.2f\n", gradeAverage);
```

If both halves of the code are run together (where the data being written and processed are those listed earlier), then the output that occurs at the end of execution is as follows:

```
Student with highest grade of 98.6 was Student 11
Student with lowest grade of 44.2 was Student 12
Average Grade Was: 66.84
```

Figure CC2.5 visualizes how the first few lines of the `for` loop read in the data from the studentdata.txt data file. The blank spaces between each "token" of data are ignored. The `Scanner` method `.next()` disregards blank spaces and captures data as a `String`. Both `.nextInt()` and `.nextDouble()` attempt to treat the next token of data to be read as the data types inherent to these methods (`int` and `double`, respectively). Since these three methods are invoked in this order, each time a loop iterator occurs, the next three tokens of data will be read from the file in the same order.



```
for (int i = 1; i <= numOfLoops; i++)
    {
        // read in name and grade.
        currentStudent = fileIn.next() + " "
                + fileIn.nextInt();

        currentGrade = fileIn.nextDouble();
```

```
Student 1 81.8
Student 2 80.6
Student 3 60.2
Student 4 82.8
Student 5 49.6
. . .
. . .
. . .
```

Figure CC2.5. Tokens of Data Read from `studentdata.txt` Using `Scanner` Methods

Student grades—record quantity not known ahead of time: What if you encounter a situation where we are not sure how many records are found within the file? This is the **most common real-world situation!** The code demonstrated so far will work then too. Consider the following, where you can change one line of code in grade generation logic (in **bold**):

```
int numOfLoops = (int)(Math.random() * 50);
```

Now this code will generate a random quantity of student grade records between zero and fifty. For example, the following are the file contents on another run:

```
Student 1 80.3
Student 2 94.3
Student 3 84.4
.. some records omitted for brevity
Student 10 46.2
Student 11 87.3
```

The following is the output statistics:

```
Student with highest grade of 95.3 was Student 6
Student with lowest grade of 41.0 was Student 9
Average Grade Was: 73.33
```

Notice that careful use of the `Scanner` class's "next" methods allows you to read in plain-text data of any type. `PrintWriter`'s implementation of the three print methods, including `.printf()`, is highly useful and allows for precise control over the output written to the file.

SUMMARY POINTS

- Java has several classes for handling both plain-text and binary-formatted I/O activities.
- The `File` class allows the developer to create basic file wrapper objects that contain the file name and location.
- It is always a good idea to try to know as much about the structure and contents of a data file as possible to make the I/O programming task easier.
- The absolute path of a file is the fully qualified location of the file on a disk or network location. The path typically starts with a file-system root element indicated.
- The relative path describes the location of a file in relation to the location where the Java application executes.
- The `PrintWriter` class provides a convenient and easy syntax for writing plain-text data out to a file. Its constructor can use a reference to a `File` instance object or a `String` containing a file path.
- The `Scanner` class provides a convenient and easy syntax for reading plain-text data in from a file in a manner similar to capturing console-based user input.
- File I/O in most any language, including Java, involves exception and error handling to some extent due to the external nature of files and the operating system's file system.

QUICK PROBLEMS

1. **Coding:** Write a small program that writes ten numbers out to a file. Have the program read those numbers back in and report the largest to the console.
2. **Think:** What is the benefit of being able to store data in plain-text files?
3. **Coding:** Write a small program that stores five names in a `String[]` array. Randomly choose from those names ten times, and write them out to a plain-text file.

CC2.2 Working with Binary File Data

Binary is the language of technology. Whether you know it or not, you are communicating with your friends, family, faculty, employer, and more in binary as you use technology. When the binary data reaches you or your recipient, it is translated, or **encoded**, into a plain-text character format that is human readable. This translation happens also in the context of the target recipient's operating system, language, culture, and so on. Instead of trying to manually format data according to the target recipient's language and context, data is stored in a binary format, and the translation is performed by their platform/language/technology. With today's technology, this adds a negligible but noticeable level of processing and resource usage. This makes data encoded into binary highly **portable**. Indeed, this is why Java classes, when compiled, are stored in binary: Java is a run-anywhere language, and the formatting of Java classes into binary makes this portability possible. Each platform can interpret the binary data in a manner that works best for it.

Figure CC2.6. Binary Data Is a Highly Portable, Often Encoding-Free Data Format

Java provides several classes to allow the developer to write out and read in binary data from files. This can be useful, as many applications expect data input to occur in binary and, the reverse, may only write out data in a binary format that your application will need to use. Figure CC2.7 shows the inheritance relationship of some of these binary data classes that will be the focus of this section.

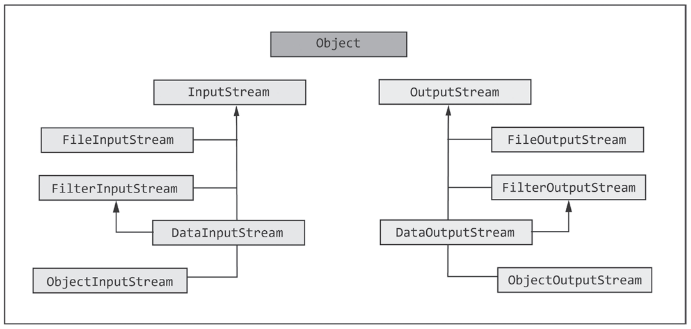


Figure CC2.7. Inheritance Hierarchy of Selected Binary Input/Output Classes

`InputStream` and `OutputStream` serve as the base classes for the two branches, respectively.[*,†] These two base classes define most of the core methods that allow their subclasses to read and write from binary files. `InputStream` and `OutputStream` themselves do not work directly with files, but they lay the groundwork for binary I/O from/to any stream of data. The `FileInputStream` and `FileOutputStream` subclasses specifically target a file for binary I/O.[‡,§] These classes can only convert integers and bytes back and forth into and out of files. The other classes, such as `DataInput[Output]Stream` and `ObjectInput[Output]Stream`, will use `FileInput[Output]Stream` as the "wrapper" for information on and access to a file, similar to how the `File` class works. This class hierarchy is also a great demonstration of the beauty of class inheritance: each subclass extends the abilities of its superclass into a more specific, targeted context.

File I/O with binary data: The most basic of the three classes, `FileInput[Output]Stream`, specifically handles connections to files. The constructor for these two classes can take a `File` instance object, similar to `PrintWriter` and `Scanner` discussed earlier in this chapter, or a `String` that contains the file path and name of the target file. Importantly, the `FileInput[Output]Stream` class has several overloaded versions that allow the developer to specify a `boolean append` parameter so that files can be added to instead of overwritten in output activities. Unlike the `File` class, though, `FileOutputStream` has the ability to both create the file and write binary data to it. Consider the following example in a new Java `main()` class application (the same import statements from earlier in the chapter still apply):

[*] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/InputStream.html.
[†] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/OutputStream.html.
[‡] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/FileInputStream.html.
[§] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/FileOutputStream.html.

Code Snippet CC2.11

```
try
{
    // Create a binary file in .dat format
    FileOutputStream binaryOut = new FileOutputStream("data.dat");

    // Write out 10 int values
    for (int i=0;i<10;i++)
    {
        binaryOut.write(i);
    }

    // Close the connection
    binaryOut.close();

    // Input code will go here. . .
}
catch (IOException ioex)
{
    // Handle an IO exception
}
catch (Exception ex)
{
    // Handle a general exception
}
```

Notice that the `FileOutputStream` constructor can use either a `File` instance object or a `String` with a file path. In this case, a `String` with a relative file path is used. When this code executes, a binary file with the "`.dat`" extension is created (a common extension for binary data files). Since a relative file path is used, the file is created in the Java project directory as shown in figure CC2.8. Figure CC2.8 also shows what is visible when the binary file is opened both in your IDE's text editor (figure CC2.8 shows this for Apache NetBeans, but it will look similar in other IDEs such as Eclipse) and in the operating system when the raw binary data is viewed. Notice that each software attempts to display the binary data in a format that makes sense to it. Though you cannot read it, the binary data is readable by other Java programs.

The `FileInputClass` can convert the raw binary data back into `int` values so it can be displayed or used:

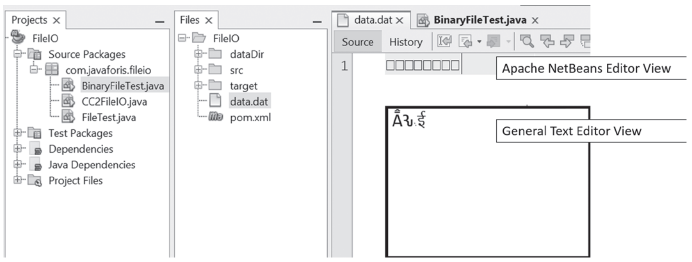


Figure CC2.8. Viewing the Raw Binary Data in an IDE and a General Text Editor

Code Snippet CC2.12

```
// . . .
// Input code will go here. . .
FileInputStream binaryIn = new FileInputStream("data.dat");

for (int i=0;i<10;i++)
{
    System.out.print(binaryIn.read() + " ");
}

binaryIn.close();
// . . .
```

When executed, this code will print the following to the console:

```
0 1 2 3 4 5 6 7 8 9
```

The "`data.dat`" file that already exists can be appended to each time it is opened and written to. By adding the following `true` to the `FileOutputStream` constructor in the write-out / read-in example, appending will occur with the file each time the application is executed:

Code Snippet CC2.13

```
// Create a binary file in .dat format
FileOutputStream binaryOut = new FileOutputStream("data.dat", true);
```

To see this in action, you will need to modify the input loop in the following way:

Code Snippet CC2.14

```
int readInValue = 0;
while (true)
{
    readInValue = (int)binaryIn.read();
    if (readInValue == -1)
    {
        binaryIn.close();
        break;
    }
    System.out.print(readInValue + " ");
}
```

Several things have changed here: Now the value from the binary file is read directly into an `int` variable. The `.read()` method of the `FileInputStream` class will return a "`-1`" when the end of the file has been reached. If a `-1` is returned, the connection is closed and while loop ended. The output from this after three runs will look like the following:

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
```

Another example is the storing of text data in byte form within a file:

Code Snippet CC2.15

```
try
{
    // Create a binary file in .dat format
    FileOutputStream binaryOut =
            new FileOutputStream("text.dat");

    // Write out some text
    String authorName = "Your Author";
    byte[] byteArray = authorName.getBytes();
    binaryOut.write(byteArray);

    // Close the connection
    binaryOut.close();

    // . . .
    // Input code will go here. . .
    FileInputStream binaryIn =
            new FileInputStream("text.dat");

    int readInValue = 0;
    while (true)
    {
        readInValue = (int)binaryIn.read();
        if (readInValue == -1)
        {
            binaryIn.close();
            break;
        }
        System.out.print((char)readInValue);
    }
}
catch (IOException ioex)
{
    // Handle an IO exception
}
catch (Exception ex)
{
    // Handle a general exception
}
```

Executing this code will output the following to the console:

```
Your Author
```

Binary file I/O with primitive data types: The `DataOutputStream` and `DataInputStream` classes allow you to work with other primitive data types without having to manually perform the conversion into `byte` values as was done earlier.[*,†] The `DataInputStream` class also has methods that can explicitly throw the `EOFException` that allows us to watch for the end of the file better than checking for a certain value. The `DataOutput[Input]Stream` classes have various methods—the common ones of which are detailed in table CC2.2—that work with primitive data types.

The methods ending in "UTF" deal with conversion to and from a portable Unicode character format and help convert `String` values into binary. You can use these methods easily. The constructors for both of these classes require an instance object of either `Input[Output]Stream` (the base classes) or one of their subclasses. For example, the `DataInputStream` constructor will accept a reference to a `FileInputStream` instance object, which itself stores information on the location and name of the binary file to be interacted with.

Earlier in the chapter, you generated random grades for randomly named students. The same can done in a binary output file:

Table CC2.2. I/O Methods of the `DataInputStream` and `DataOutputStream` Classes

Data type	DataInputStream	DataOutputStream
int	.readInt()	.writeInt(int i)
char	.readChar()	.writeChar(char c)
boolean	.readBoolean()	.writeBoolean(boolean b)
long	.readLong()	.writeLong(long l)
short	.readShort()	.writeShort(short s)
double	.readDouble()	.writeDouble(double d)
float	.readFloat()	.writeFloat(float f)
String	.readUTF()	.writeUTF(String str)

Code Snippet CC2.16

```
int numOfStudents = 20;

try
{
    FileOutputStream binaryOut =
            new FileOutputStream("studentdata.dat");
    DataOutputStream studentOut =
            new DataOutputStream(binaryOut);

    for (int i=1; i<numOfStudents; i++)
    {
        studentOut.writeUTF("Student");
        studentOut.writeInt(i);
        studentOut.writeDouble((40 + (Math.random() * 61)));
```

[*] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/DataOutputStream.html.

[†] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/DataInputStream.html.

```
    }

    studentOut.close();
    binaryOut.close();

    FileInputStream binaryIn = new FileInputStream("studentdata.dat");
    DataInputStream studentIn = new DataInputStream(binaryIn);

    for (int i=1; i<numOfStudents; i++)
    {
        System.out.print(studentIn.readUTF() + " " + studentIn.readInt());
        System.out.printf("\t%.2f\n", studentIn.readDouble());
    }

    studentIn.close();
    binaryIn.close();

}
catch (IOException ioex)
{
    // Handle an IO exception
}
catch (Exception ex)
{
    // Handle a general exception
}
```

Executing this code produces the following output:

```
Student 1     49.47
Student 2     54.17
. . .some records omitted for brevity
Student 18     64.48
Student 19     75.54
```

Notice that in the section of the code, the file is written out so that blank spaces, tabbing, or new-line characters are not a concern. Java handles the binary encoding of this data and knows where each token of data ends and the next begins. The use of the various `.read___()` methods helps handle this for the developer as well.

How would you modify this in the more realistic scenario where the number of records of data in the file is not known ahead of time? The `.read___()` methods of `DataInputStream` will throw an `EOFException` that will catch when the end of the data file has been reached in our code. You can modify our example like so:

Code Snippet CC2.17

```
int numOfStudents = 20;

// Use a Try-With-Resources
// Autocloses file connections
try (
    FileOutputStream binaryOut =
            new FileOutputStream("studentdata.dat");
    DataOutputStream studentOut =
            new DataOutputStream(binaryOut);
    FileInputStream binaryIn =
            new FileInputStream("studentdata.dat");
    DataInputStream studentIn =
            new DataInputStream(binaryIn);
    )
{

    for (int i=1; i<numOfStudents; i++)
    {
        studentOut.writeUTF("Student");
        studentOut.writeInt(i);
        studentOut.writeDouble(
                (40 + (Math.random() * 61)));
    }

    studentOut.close();
    binaryOut.close();

    while (true)
    {
        System.out.print(
                studentIn.readUTF() + " " + studentIn.readInt());
        System.out.printf(
                "\t%.2f\n", studentIn.readDouble());
    }
}
catch (EOFException eofex)
{
    System.out.println("[End of Student Data File Reached!]");
}
catch (IOException ioex)
{
    // Handle an IO exception
}
catch (Exception ex)
{
    // Handle a general exception
}
```

In this example, you convert the `try…catch` to a try-with-resources block, where the file resources you want to close are declared and initialized in the header of the `try` itself. This technique will automatically close the file resources when done (notice the calls to `.close()` are gone in the new code as well). The end-of-file exception catch has been added, and a message prints letting the user know that all data has been read:

```
Student 1     84.48
Student 2     99.87
Student 3     67.73
. . .some output omitted for brevity
Student 17     92.02
Student 18     47.35
Student 19     55.76
[End of Student Data File Reached!]
```

Binary file I/O with complex data types: `ObjectOutputStream` and `ObjectInputStream` classes are very useful: they allow a developer to save actual complex data-type objects to a binary file.[*,†] This can come in handy and prevent the need for a lot of tedious logic that might have to extract data from objects and write them out one by one. First, consider this example of saving an array and the contents of an `ArrayList` (companion chapter 5) to a binary file using the `DataOutputStream` class:

[*] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/ObjectOutputStream.html.
[†] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/ObjectInputStream.html.

Code Snippet CC2.18

```
try
{
    int[] numArray = {3,4,2,4,6,3,0,8,9};
    ArrayList<Integer> intArray = new ArrayList<>();
    intArray.add(5);
    intArray.add(9);
    intArray.add(11);

    FileOutputStream fileOut = new FileOutputStream("arrays.dat");
    DataOutputStream arraysOut = new DataOutputStream(fileOut);

    for (int num: numArray)
        arraysOut.writeInt(num);

    for (Integer num: intArray)
        arraysOut.writeInt(num);

    fileOut.close();
    arraysOut.close();

    // Read in array code here
}
catch (EOFException eofex)
{
    System.out.println("[End of Student Data File Reached!]");
}
catch (IOException ioex)
{
    // Handle an IO exception
}
catch (Exception ex)
{
    // Handle a general exception
}
```

Two `for` loops are defined to traverse each array and write out their `int` and `Integer` values. If the regular `int[]` array had any empty locations (with default value `0` in them), you would have to consider whether or not the `0` values should be written out as well.

Now the real problem: If this application (or another Java application) needs to read these back in, and the number of elements that should belong to each array is unknown, how do you tell where one array ends and the other begins? This is a real head-scratcher! It would be great if there was a way to save the arrays into a file directly *and* preserve their array structure so that the start and finish of each were also preserved. The `ObjectOutput[Input]Classes` let you do just that. Modify this example like so (changes in **bold**):

Code Snippet CC2.19

```
try
{
    int[] numArray = {3,4,2,4,6,3,0,8,9};
    ArrayList<Integer> intArray = new ArrayList<>();
    intArray.add(5);
    intArray.add(9);
    intArray.add(11);

    FileOutputStream fileOut = new FileOutputStream("arrays.dat");
    ObjectOutputStream arraysOut = new ObjectOutputStream(fileOut);

    arraysOut.writeObject(numArray);
    arraysOut.writeObject(intArray);

    fileOut.close();
    arraysOut.close();

    // Read in array code here
    int[] fileNumArray;
    ArrayList<Integer> fileIntArray;


    FileInputStream fileIn = new FileInputStream("arrays.dat");
    ObjectInputStream arraysIn = new ObjectInputStream(fileIn);


    fileNumArray = (int[])arraysIn.readObject();
    fileIntArray = (ArrayList<Integer>)arraysIn.readObject();


    arraysIn.close();
    fileIn.close();


    for (int num: fileNumArray)
        System.out.print(num + " ");


    System.out.println();


    for (Integer num: fileIntArray)
        System.out.print(num + " ");


    System.out.println();

}
catch (EOFException eofex)
{
    System.out.println("[End of Student Data File Reached!]");
}
catch (IOException ioex)
{
    // Handle an IO exception
}
catch (Exception ex)
{
    // Handle a general exception
}
```

When this code runs, the following will print to the console:

```
3 4 2 4 6 3 0 8 9
5 9 11
```

The `ObjectInput[Output]Stream` classes have available all the classes used with the `DataInput[Output]Stream` class with one key addition:

Code Snippet CC2.20

```
void .writeObject(Object o)          // ObjectOutputStream
Object .readObject()                 // ObjectInputStream
```

Recall that the `Object` class is the top-level definition for all classes in the Java library. The writing out and reading of objects, up-converted to Object, ensures that these I/O classes can be used with object data types that may not even exist yet in addition to all those that currently do. Notice in the code that a cast operator is used to convert the read in `Object` data type back to the array data type desired. Notice as well that you did not need to know anything about the size or contents: all that structure was saved into the binary file and preserved.

Java file I/O and user-defined classes: You can use these I/O classes with user-defined classes (ones we create) as well. For example, consider class `Person` defined in chapter 9. Each instance object of `Person` will store several pieces of data on each individual, and these are of various data types. Import the `Person` class from chapter 9 into your current project (or see the code listing there to type it in if you have not created it), and consider the following code that creates a small array of `Person` objects:

Code Snippet CC2.21

```
    ArrayList<Person> roster = new ArrayList<>();
    Person temp = new Person("Blue",70,23,"Suzie",150000);
    roster.add(temp);
    temp = new Person("Green",71,22,"Brianna",151000);
    roster.add(temp);
```

Instead of traversing the `ArrayList<E>` and accessing the data fields (or accessor methods) for each `Person` object, you can simply write the entire `ArrayList<E>` object out to the file! The following code accomplishes this (*Note:* output prints for the other `Exception catch` statements have been added):

Code Snippet CC2.22

```
try
{
    ArrayList<Person> roster = new ArrayList<>();
    Person temp = new Person("Blue",70,23,"Suzie");
    roster.add(temp);
    temp = new Person("Green",71,22,"Brianna");
    roster.add(temp);

    FileOutputStream fileOut =
            new FileOutputStream("personroster.dat");
    ObjectOutputStream personOut =
            new ObjectOutputStream(fileOut);

    personOut.writeObject(roster);

    fileOut.close();
    personOut.close();

    FileInputStream fileIn =
            new FileInputStream("personroster.dat");
    ObjectInputStream personIn =
            new ObjectInputStream(fileIn);

    ArrayList<Person> newRoster =
            (ArrayList<Person>)personIn.readObject();

    for (Person p: newRoster)
        System.out.println(p.toString());

}
catch (EOFException eofex)
```

```
{
    System.out.println("[End of Student Data File Reached!]");
}
catch (IOException ioex)
{
    System.out.println(ioex.toString());
}
catch (Exception ex)
{
    System.out.println(ex.toString());
}
```

When this code is executed, an error occurs!

```
java.io.NotSerializableException: com.javaforis.fileio.Person
```

For the instance objects of complex data types and classes to be eligible for storage in a binary file, the definitions of those classes (the CDF) must specify that the class implements the `Serializable` interface (see companion chapter 4 for more information on interfaces in Java). The `Serializable` interface has no abstract methods that must be overridden. A class that implements it will have its basic structure summarized when its instance objects (and their data) are written out to a binary file. The class must be symbolically resolvable too (through either import or inclusion in your project) to read them in from a binary file.

To get the code working, the header for the imported `Person` class will need to be modified:

Code Snippet CC2.23

```
import java.io.*; // Contains definition for Serializable

public class Person implements Serializable {
// . . .
```

The program will now execute and print the following to the output console:

```
Name: Suzie, age: 23
Name: Brianna, age: 22
```

Writing out one array that stores references to multiple complex data-type objects is a lot easier than tediously traversing the data fields for each and writing them out one by one. Java's I/O classes handle the structure of the objects and the structure of their data for you. Simply specify if any of your custom, user-defined classes implement the `Serializable` interface, and their instance objects will be writable (and readable) from binary files. Handy for transporting complex objects to Java applications across platforms.

SUMMARY POINTS

- Binary data is much more portable than plain text, as it allows receiving platforms and software to choose how to encode it for view or usage.
- Java divides its binary I/O classes into two difference hierarchies: input- and output-focused classes.
- The `InputStream` class is the base class for all input-focused binary I/O classes, and the `OutputStream` class is the base for all output-focused binary I/O classes.
- The `FileOutputStream` class acts in a similar fashion to the base nature of the File class for plain-text I/O. `FileOutputStream` handles the low-level encoding into binary and writing directly to the binary files. `FileInputStream` can read bytes in from a binary file.
- The `DataOutputStream` class allows the writing out of the Java primitive data types to a binary file via a `FileOutputStream` object. `DataInputStream` uses a `FileInputStream` object to read bytes in from a binary file and convert them back to the appropriate primitive data types.
- For classes and complex data types that have implemented the `Serializable` interface, `ObjectOutputStream` can store complex objects directly in binary data files. `ObjectInputStream` can read them back out.

QUICK PROBLEMS

1. **Coding:** Write a small program that writes the multiplication table out to a binary file. Store the "# x #" (where the # represents zero through nine) along with the answer on each line. Have the program read all the records back in and display them on the console.
2. **Think:** What benefits are there to writing a complex object out to a binary file?
3. **Coding:** Store the results of the multiplication table in a two-dimensional array. Write this array out to a binary file and then read it back in and display the results to the console, with the appropriate multipliers along the top row and left column.

Summary

In this chapter, you have explored the very basics of working with file I/O in Java. Several other I/O classes and topics have been excluded for brevity, but you are invited to research how to use them. For example, the `FileWriter` class provides a convenient set of overloaded constructors to allow easy appending of data to plain-text data files, similar to the constructors of the `FileOutputStream` class.[*] The `BufferedInputStream` and `BufferedOutputStream` allow you to write out to the computer's memory first and then write to disk, often increasing application efficiency and reducing disk usage in large, scaled applications.[†,‡] Now that you have a basic grasp of how to work with both plain-text and binary data files, you can accomplish a great deal of common, information-intensive tasks: enabling data save and data backup in applications, creating data files for other applications to process, publishing the programmatic analysis of data to an open scientific commons, and so on. There is a lot you can now accomplish with file I/O in Java!

Practice Problems

Terminology

Match the following terms from the chapter with their most appropriate definition:

1. I/O	a. View of the actual location, arrangement, and directory structure of files in the file system.
2. Files	b. A character, symbol, or group of either that precedes a relative file path, further indicating the nature of the relative location indicated.
3. `File` class	c. Data format that is highly portable between platforms and easily translatable by various platforms and software.
4. Absolute path	d. Java class that provides basic binary output capabilities, writing out `byte` and `int` values. Serves as the "file wrapper" for the other binary I/O classes.
5. Relative path	e. A relative file path prefix that indicates the location in a directory immediately superior to the currently executing directory. These prefixes can be chained to indicate a directory quantity superior to the current.
6. Root element	f. A form of the `try…catch` statement where external resource links are defined in the try header and connections to these external resources are closed automatically when the program terminates.
7. Prefix	g. Java class that provides basic, plain-text output to files.
8. `./`	h. Java interface that allows instance objects of classes to be stored in binary files.

[*] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/FileWriter.html.
[†] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/BufferedInputStream.html.
[‡] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/BufferedOutputStream.html.

9. `../`	i. Java class that provides the ability to write both primitive data types and complex data-type objects out to files in binary format.
10. Physical file view	j. Base class of all Java classes by either direct or indirect inheritance.
11. Logical file view	k. File path that specifies the name and location of a file that is related to the location where an accessing program is executing.
12. `PrintWriter` class	l. Java class that serves as the base class for all output-focused I/O classes.
13. Exception handling	m. Constructor parameter option for the `FileOutputStream` class enabling the adding of data to a preexisting file's contents rather than overwriting them.
14. Try-with-resources block	n. Structures that are external to an application, usually located on a hard drive, portable drive, or networked/cloud location. Offers a more long-term means of data storage.
15. Binary file data	o. The first, or base, location of a file path.
16. FileOutputStream	p. File path that fully specifies the location of a file.
17. DataOutputStream	q. A relative file path prefix that indicates the location starts in the local, executing directory.
18. ObjectOutputStream	r. A view of the relationship between files and directories that may not be physically structured or closely colocated.
19. OutputStream	s. Technique where common errors that could occur during program execution are "caught" and "handled" to ensure application stability.
20. Object	t. Acronym for input/output. Usually refers to the file, data, or network activities (among others) of an application or system.
21. Append to file	u. Java class that provides the ability to write primitive data types out to files in a binary format.
22. Serializable	v. Java class that provides a basic "wrapper" storing information on files such as name and file path.

Find the Error

In the following **single-code listing**, carefully examine the code given, and determine the error(s)/issue(s) with each. Keep in mind, the error(s) could be syntax (code) or logic (intended outcome) based or both!

For the following problem, assume all necessary imports have been made and that the code is located within an application's `main()` method. Also, assume the application is reading from a **binary** file called "`weatherdata.dat`" that contains weather observations, with an **unknown quantity of records** for one year with data in the following format:

DayOfMeasurement   MonthOfMeasurement   MaxTemperature   MaxWindSpeed   Humidity%

The application has been tasked with the following requirements:

- Report the average max temperature for each month.
- Report the average max windspeed for each month.
- Report the average humidity for each month.

The file is **already located in the application's local directory**. A (plain-text) example of the data would look like the following:

```
5       1     92.2      3.4     35
6       1     91.0      2.2     69
. . .
123     4     56.1     14.5     15
124     4     58.8      1.1     80
. . .

File weatherData = new File("../weatherdata.dat");
```

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

while (true)
{
    fileIn.nextInt();
    fileIn.nextInt();
    fileIn.next(int);
    fileIn.nextDouble();
    fileIn.nextInt();
}

int day = fileIn.nextInt();
int month = fileIn.nextInt();
for (int i=0;i<=12;i++)
{
    monthAverage =
            (int maxTemperature = fileIn.readDouble()) / 12;
    monthAverage =
            (double maxWindSpeed = fileIn.readDouble()) / 365;
}

DataOutputStream fileOut =
        new DataOutputStream(
                new FileOutputStream("results.dat"));
fileOut.writeDouble(monthAverage);
fileOut.writeDouble(monthWindSpeedAverage);
fileOut.close();

System.out.println(
        "Average Temperature: " fileOut.readDouble());
System.out.println(
        "Average Windspeed: " + fileOut.readDouble());
System.out.println(
        "Average Humidity: " + fileOut.readDouble());

try
{
    fileOut.close();
}
catch (Exception ex)
{
    fileOut.close();
}
```

Think about It

1. What is the main difference between a file with plain-text data and one with binary-formatted data?
2. What are three things you should know about a data file before working with it programmatically?
3. What is the difference between an absolute file path and a relative file path?
4. Describe the use of the root element in a file path. Is it needed for absolute file paths, relative file paths, or both?
5. Why is escaping characters sometimes necessary in a file path String value?
6. What is the main purpose of the `File` class?
7. What are two different directory references for a relative file path? How do you use them?
8. Why do data files created by your code not show up in an IDE's logical view of a Java project but instead in its physical file view?
9. What are some of the more common methods of the File class, and what do they do?

10. Describe the primary usage of the `PrintWriter` class. Is it used more for binary data or plain-text data?
11. Describe how the Scanner class can be used for more than just console input.
12. Why must most file I/O code in Java and other programming languages watch for and handle exceptions?
13. What happens when you try to view the contents of a binary file manually?
14. Describe why storing data in a binary format can provide benefits to programmers.
15. Describe the inheritance hierarchy of the Java binary I/O classes.
16. Which of the Java binary I/O classes can write primitive data types like `double`, `float`, and `short` to a binary file?
17. What is the best way to read in a `String` value from a binary file? Is there more than one way?
18. Why must there be a type conversion when working with complex data-type objects and binary files? When must this type conversion occur?
19. What must be adjusted in a class to enable its instance objects to be written to a binary data file?
20. What are several ways to handle reaching the end of a data file during read activities?

Short Syntax Problems

1. Write a small program that accepts inputs from the console of a student's name (first and second), GPA, and major. In a loop, write this record out to a plain-text file ten times. Make sure each record is on its own line.
2. Write a small program that accepts inputs from the console of a student's name (first and second), GPA, and major. In a loop, write out each entered record to a binary file. Use the appropriate binary I/O class so that if the application is exited and then executed again, new records are appended to the same file.
3. Find a long block of plain text from a website or file of your choice. Copy and paste it into a plain-text file, and save it into your project as a local file. Write a small program that will read in each word separately, then count the vowels. The program will count all the vowels in every word in the file and then at the end report the count for each (include the English character *y* in your count).
4. Create a small program that fills a size `10` int array with random numbers between `0` and `100`. Append the entire array into a binary file each time you run the application. Write a second small program that will read in all the arrays present in the same binary file and print their contents to the console.
5. Write a small program that will accept inputs for grocery store inventory items from the console and write them out to a binary file. The information will include the item name, price, quantity in stock, and general description. The application will loop until a menu option is chosen to quit.
   a. **Added challenge:** Use the `InventoryItem` class from chapter 8.

Full Problems

1. Write a Java program called `StockMarket.java` that will allow a user to perform the following tasks:
   a. The user can enter the information for a business's stock (i.e., security). The information will include the stock's ticker symbol, the business name, and the stock's most recent price at the close of trading day:
      i. Have the application write this out to a file called `stockdata.txt.`
      ii. Include a way for the user to loop, entering new stocks until they are finished.
   b. The user can then enter their portfolio:
      i. Prompt the user to choose from one of the stocks listed in stockdata.txt.
      ii. Have them enter a quantity purchased.
      iii. Have them enter a purchase price (which could differ from the most recent closing price).
      iv. Store this information in a file called `userportfolio.dat` (notice the `.dat` extension: use the appropriate code to write/read from a binary file!).
      v. Include a way for the user to loop and enter multiple owned stocks, giving them a way to finish when done.
   c. Finally, report on the user's gains or losses in their portfolio:
      i. For each stock in the user's portfolio, take the stock's most recent price, and subtract from it

the purchase price. Multiply this by the owned quantity. This will be the user's unrealized gains/losses per stock. Print this information for each stock in the user's portfolio.

   ii. Report the total unrealized gain/loss for the entire portfolio before allowing the application to close.

2. Implement a fully functional version of the weather observation problem detailed in the "Find the Error" section earlier in this chapter. Take the following steps:
   a. First, write a small program that will generate random data for the MaxTemperature, MaxWindSpeed, and Humidity (within reasonable value ranges). Generate a random number of observations between three hundred and one thousand days' worth. Increment the month every thirty days.
   b. Second, write an error-free version of the application that will read in the data as structured from the data file and report the required statistics:
      i. Average MaxTemperature for each month
      ii. Average MaxWindSpeed for each month
      iii. Average Humidity for each month

      Ensure that the month number is reported along with each average.
3. Write a program that will prompt a user for their full name (first and last) and a quantity of time they would like a loop to print their name to the console. Then in a plain-text file, your application should write out all the appropriate syntax for an actual Java program that will loop and print the user's name to the console. <u>Your program will be writing the code for a program.</u> Write out the class header, the main method header, and the code. Adjust the code dynamically so that the entered username and the loop quantity are part of the `for` loop you write to the plain-text file. Be sure to give your plain-text file the extension of ".java."
   a. **Added challenge:** When done, open the file in your IDE. Can you compile and run it?
4. Modify the `ManageInventory.java` application written earlier in this chapter in the following ways:
   a. Instead of the `ArrayList<E>` containing `String` arrays, use the InventoryItem class from either chapter 8 or chapter 11 instead. Have the `ArrayList<E>` contain InventoryItem class instance objects. You'll need to modify the import, export, edit, and new entry logic to accommodate for this.
   b. Add a "Profit Report" menu option where, for each `InventoryItem`, the potential profit is reported if all the on-hand quantities of that item were sold for the sale price (profit = what it sold for – what was paid to purchase it by the store).