Chapter 5
File Input/Output

We now turn our attention to reading and writing text files. Java has libraries that make this pretty straightforward.

File I/O

File I/O (input/output) means reading and writing the contents of files. We can view the contents of a file as if it contains text (in ASCII, Unicode, etc) or also as if it contains binary data such as integers, floats, and other types of non-textual data.

We'll show a couple of versions of reading and writing content, so just keep in mind that the preferred versions would be:

- read textual data with a Scanner
- write textual data with a PrintWriter
- read and write binary files with ObjectInputStream and ObjectOutputStream.

Reading Text with a Scanner.

We have already been using Scanner objects to read from the keyboard via System.in. It turns out we can attach a Scanner to many different things!

```
import java.util.Scanner;     //before the class
import java.io.File;    //before the class
...
Scanner sc1 = new Scanner (System.in);
Scanner sc2 = new Scanner ("really long\nString\n\t\tthat I want to pick apart\n");
Scanner sc3 = new Scanner (new File("local.txt")); // needs exception handling… see below
```

In each case, we are then able to just use the Scanner methods all in the same fashion. We can call nextInt(), nextDouble(), nextLine(), and so on regardless of where the inputs are coming from. We get to use this one nice interface for everything!

When opening a file, we should close the file when we're done with it: `sc3.close();`

The only ingredient missing for us is to be able to handle exceptions that might occur when opening a file. For certain kinds of exceptions, Java will not allow us to write code that might allow those types of exceptions to silently escape. If we tried the declaration of Scanner sc3 above, we'd see the following compiler error message:

```
Error: /path/to/file.ClassName.java:8: unreported exception
java.io.FileNotFoundException; must be caught or declared to be thrown
```

We will have a more detailed look at exceptions later on; for now, we will need to use our exception handling (try-catch blocks) successfully to just read a file:

```
import java.util.*; // Scanner
import java.io.*;   // File
public class FIO {

  public static void main(String[] args){
    Scanner sc3 = null;
    try {
      sc3 = new Scanner (new File("help.txt"));
    }
    catch (FileNotFoundException e){
      //print actual error messages to System.err.
      System.err.println("file wasn't found! Choosing to quit now.");
      //return 0 for 'normal' exit; non-0 for crashy exit
      //(also resets DrJava's "interactions" pane).
      System.exit(0);
    }
    // assuming we got here, use the Scanner as normal.
    String s = sc3.next();
    System.out.println("s='"+s+"'");
    sc3.close();
  }
}
```

Notes

- We are printing error messages to System.err.
- We are choosing to end the program with our catch-block (via System.exit(0)), but your own program might want to loop back and try another file somehow, or use a default file, or whatever makes the most sense for your situation.
- We closed the Scanner when we were done with it.

Your Turn!

- Write code to ask the user for a text file; repeatedly ask for it until you get one that exists. Then print out its contents all in uppercase.
- Find an old class definition lying around that had you perform significant user interaction. Change the code to attach the Scanner to a particular file, and then try putting your user responses into a file all at once. See if you can run your program successfully in this wholly automated way!
    - You could write a program that accepts a file containing a single int (how many items are coming), followed by that many ints, and then print the sum of them.

Overall, using the Scanner class looks much more inviting. We'll try echoing lines in all-caps.

```
    public static void main(String[] args) throws IOException {
      Scanner sc = new Scanner (new FileReader(foo.txt"));

      String currentLine;
      while (sc.hasNextLine()) {
          currentLine = sc.nextLine();
          System.out.println(currentLine.toUpperCase());
      }
      sc.close();
    }
```

The Last Newline Question

The only complication we have is if the last line of text doesn't contain a newline character. Consider a file that contained the following in it: "a \n b \n c \n d e f". There are three words on the last line (d e f), but our Scanner's nextLine method will return false at the last line, because there is no newline character.

We can fix this by just checking whether we have either any next token (using the hasNext() method, looking for non-whitespace) or a newline (using the hasNextLine() method):

```
if (sc.hasNext() || sc.hasNextLine() ) { … }
```

Now, it doesn't matter if we have empty lines at the end (caught by hasNextLine), or an incomplete line at the end (no newline character, caught by the hasNext method).

Note: although creating the Scanner as Scanner sc = new Scanner (new File("foo.txt")); also works, it makes it harder to identify non-existing files. The original way we showed also allows for closing the underlying file stream (because FileReader implements Closeable and File doesn't) , which lets us manage our resources better.

Your Turn!

- Use a Scanner to open a file that contains an integer first, indicating how many words are in the file; then, create an array of Strings of that length, and read in all the words using the next() method.
    - Be sure to test this on a file that has stuff on the last line, but no newline character (so, the cursor is next to a word but can't go down to the next blank line).
- Mix-and-match: use command line arguments to get the file name; does this change your usage?
- Where we've had "foo.txt", you can actually have a proper directory path, such as "foo/bar.txt". Give it a try.
- Question: what would you say is the difference between using a Scanner on System.in, using a Scanner on a File object, and using command-line arguments?
    - How might you reasonably use two or more means of input at once in a project?

Writing to Text Files

Writing to a text file is very easy – we have already been using System.out.print, System.out.println, and System.out.printf to write to the terminal, and we will now use a **PrintWriter** object to print/println/printf directly to a named file. We still must deal with exceptions just like when reading a file, though. Here's a complete example:

```
import java.util.*; // Scanner
import java.io.*;   // File, PrintWriter

public class FIO {

  public static void main(String[] args){

    PrintWriter pw = null;
    try {
      pw = new PrintWriter(new File("help.txt"));
    }
    catch (FileNotFoundException e) {
      System.err.print("couldn't open file for writing!");
      System.exit(0);
    }
    pw.print("partial line ");
    pw.println("complete line");
    pw.printf("with %s\n","substitutions");
    pw.close(); // always!
  }
}
```

Notes

- We can use print and println in the same familiar way as we have with PrintStream (System.out is a PrintStream).
- It even works for our own types, like a Square object, in getting the String representation via the toString implementation.
- We **always** close the PrintWriter when we're done! This is when your file actually gets written, even though all the previous code was figuring out what will be written.
- Nothing about file-writing means we can't still get input via a Scanner anywhere, or even still write to the terminal via System.out.

Your Turn!

- Try writing a method that accepts an integer n as a parameter and a String filename for a filename, writes n to a file, and then that many integers, into the named file (separated by some whitespace). → this kind of file can be read by a previous Your-Turn task! ☺ Just write any numbers you want – all 7's, increasing/decreasing numbers, anything.
- Write a complete program that (1) asks the user for a filename and a number to use for n; (2) calls your file-writing method; (3) calls your file-reading method and prints out the result to the terminal.
- In a method that accepts an array of ints as a parameter, create two separate PrintWriter objects, attached to two files, named evens.txt and odds.txt. Using that array as your source of numbers, print the even numbers on separates lines in the file evens.txt and print the odd numbers on separate lines in the file odds.txt.

Historical Diversion:
BufferedReader and BufferedWriter.

Prior to the Scanner class's existence, the way to perform textual reading and writing was through the BufferedReader and BufferedWriter classes. You might come across it in legacy code, so it is presentd here. But please use Scanner and PrintWriter when you've got the choice.

Textual Input with BufferedReader.

As a simple approach, let's begin with an example that allows us to read lines of text from a file, and prints them to the terminal in all-caps.

```
import java.io.*; // gets BufferedReader, FileNotFoundException, IOException, FileReader…

public class Lab7 {

   public static void main(String[] args) throws IOException {
         String theFileName = "foo.txt";
         BufferedReader br = new BufferedReader(new FileReader(theFileName));
         String currentLine = br.readLine();
         while (currentLine != null) {
            System.out.println( currentLine.toUpperCase() );
            currentLine = br.readLine();
         }
         br.close();
   }
}
```

The important parts of this example are:
(1) we see how to create a BufferedReader object based on a file name
(2) we are seeing the invocation necessary to call the readLine method
(3) we can check if the end of the file has been reached by checking if the result of reading a line is null.

Here are the littler details of the example: we create a BufferedReader object, which represents a stream of characters that can be read using the readLine() method. We supplied our buffered reader the stream of characters in the form of a new FileReader, whose constructor needed a String containing the file name.

As a brief look at Java prior to the introduction of the FileReader class, we would create a BufferedReader like this:

```
BufferedReader oldWay = new BufferedReader(
                                    new InputStreamReader(
                                          new FileInputStream(new File("foo.txt"))));
```

Hideous long-lined code! At least the usage afterwards would just be the same as above, calling readLine and checking that the result isn't null.

There aren't many other useful methods in BufferedReader: we can read a single character with the read() method, and some other ways to mark our current place in the stream and jump back to it, but we don't have any of the convenience that we saw with the Scanner class.

Textual Output with BufferedWriter

File output is achieved in similar fashion. This first time around, we will use a BufferedWriter object. We will do even better with a PrintWriter next. But you just might come across code that still does things this first way we show, so let's take a quick look at it.

```
public static void main(String[] args) throws IOException {
        BufferedWriter bw = new BufferedWriter ( new FileWriter("out.txt"));
        bw.write("hello!");
        bw.write(" more stuff!\n yeah\n that's write\n");  //spelling intended.
        bw.write("the end.!no newline.");
        bw.close();     //always!
}
```

Notes:

- We feed the contents we want written to the file a String at a time, through the write method. We manually add newlines ("\n") as necessary.
- We **always** close the file when we're done!
- because the write method expects a String, if we pass other types to it, it unfortunately won't automatically use toString definitions to get the String representations – it will record their actual byte representation, which would then look like various seemingly-random characters once those bytes are used as lookups into the ASCII/Unicode tables. Manually call toString (or even wrap up as a Wrapper object to do so on primitive types) for non-String types.

Your Turn!

- write your name, favorite number, and favorite color to a file using a BufferedWriter.

Reading/Writing Binary Data

We can also read and write files that contain binary data: the actual bits representing, say, an int, double, or Square, rather than human-readable ASCII characters. The difference between a text file and binary file is the same idea as the difference between "5" and 5; they have different bit representations and will only make sense if interpreted the correct way.

We could use DataInputStream and DataOutputStream if we only had primitive values, but we can also use ObjectInputStream and ObjectOutputStream for a mixture of objects and primitives, so we will jump directly to the Object* versions.

Let's start with the following **two** pieces of code:

- example code using an ObjectOutputStream for writing, and ObjectInputStream for reading:

```
public static void main(String[] args) throws IOException,ClassNotFoundException {

    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream ("test.bin"));
    oos.writeInt(5);
    oos.writeDouble(Math.PI);
    oos.writeObject(new Square(17));
    oos.close();

    ObjectInputStream ois = new ObjectInputStream (new FileInputStream ("test.bin"));
    int x = ois.readInt();
    double pi = ois.readDouble();
    Square sq = (Square) ois.readObject();      //note we had to cast to our object type
    System.out.println("x="+x+"\npi="+pi+"\nsquare: "+sq+"\n");
    ois.close(); //always!
}
```

- a modification to our Square class so that it can be written:

```
import java.io.Serializable;
public class Square implements Serializable {
        …
```

Notice that we can write primitive types with writeInt, writeDouble, writeBoolean, and so on. For all objects, regardless of the class, we need them to implement java.io.Serializable, and then we use the writeObject method of ObjectOutputStream to write it.

Notes

- We are writing the actual bit representations to a file; try opening the test.bin file above with your favorite text editor. Why do we not see the written contents in human-readable form? Because "5"≠5, and we are interpreting the bit-pattern for 5 (which takes 4 bytes = 32 bits) as a character encoding, reading a byte at a time and interpreting them as ASCII codes.

- We always close the file! Whether we are reading or writing it, we close it when we're done. Yes, if the program ends, closing may be taken care of for you; still, you should get in the habit of closing

them on your own.

- Only Serializable things can be written/read to binary files. The Serializable interface has zero methods, though, so if there's even the slightest chance that a class might be used in this fashion, go ahead and implement it.

- Of course the file-writing and file-reading don't have to be adjacent blocks of code! Now you can write programs where you store objects for 'long-term' storage between program runs, and you can read the information back in. For instance, the high scores table in a game could be stored in a binary file by writing the objects to the file, and they could be read back in the next time the program runs.

- **One Step Further:** If your object contains references to other objects, Java will automatically write those other objects into the stream so that they are also read in automatically. Even better, suppose obj1 and obj2 both have references to obj3. If you write obj1 (causing obj3 to be written too) and then write obj2,(with a reference to obj3 also), and then later on read them both, they will still have references to the same third object! When you write obj2, Java realizes that it's already written the object to which obj2 is referring, and it will store a reference to that already-written object seamlessly. This only works within one output stream, so if you are creating multiple ObjectOutputStreams and writing an object via both, they will be separate copies and are no longer linked together.
    - Always use just one ObjectOutputStream object to write one file.

- Sadly, appending more objects to a file isn't directly allowed (perhaps due to seeking for shared sub-objects?). But you can read in the objects, and then write them as well as your new objects with a single ObjectOutputStream.

Your Turn!

- Make a couple classes that are Serializable. It would be even better if one of them contains fields of the other (like a Sphere having a Coordinate). Reusing classes you have lying around is just fine here; the point is to make them Serializable so we can use them next.

- Create an ObjectOutputStream, create a few objects, and write them to the file (using writeObject(..)).
    - Write some primitive types into the file, too (using writeInt(..), writeFloat(..), etc).
    - close the file.

- Create an ObjectInputStream, and read in the objects. You'll **have** to do so in the exact same order as you wrote them; we cannot escape the ordering.
    - You'll have to cast each object to the type you're expecting. This could cause an exception if you cast to the wrong type.
    - close the file.

- Use (maybe just print) the objects again, showing to yourself that you've successfully read in objects/primitives from the file.

- Problem: Suppose you want to make your class Serializable, but it has a field of a non-serializable class. How could you get around the situation, so that you can still have a serializable version for

your class? Direct coding solutions, or reading about the issue online, think of how you can use the Java ideas you've already learned (perhaps quite recently, hint-hint) to solve this issue.