**Day 36 Character IO and Interfaces**

**NOTE:**To use the classes discussed in this lecture,   
 import java.io.\*;   
at the top of your program.

**Files**Files are represented in java with an object of the File class. Such an object may represent an actual file that is stored in some folder, or it may represent a ‘potential’ file and we use the File object to find out about the potential File. For example, suppose we ask the user for the name of a file to read:  
  
 Scanner in = new Scanner(System.in);  
 . . .  
 System.out.print("Enter the name of the file: ");  
 String fileName = in.next(); in.nextLine();

To find out if the file exists, and is readable, we write the code  
  
 File inFile = new File(fileName);  
 if(inFile.exists() && inFile.canRead()) {  
 // read the file here  
 }

If we are going to write a file, we do not need to do these checks. If the file does not exist, Java will create it for us. If the file **does** exist, Java will erase it and start us off with a new file.

**Reading a Text or Character File**In either event, reading or writing, we need **to** **open** the file, **read** it (or **write** it), and then **close** it. The simple way to read a file is:  
 FileReader fRead = new FileReader(inFile);  
 Scanner fIn = new Scanner(fRead);  
and then use fIn.nextInt(), fIn.nextDouble(), fIn.next(), fIn.nextLine() … etc … as we have done for reading from the keyboard. These lines turn the File object into a FileReader object and then into a Scanner.  
  
This can be done shorter with:  
 Scanner fIn = new Scanner( new FileReader( inFile ) );  
to get from the fileName to the Scanner.  
  
When we are done with the file, we close it as:  
 fIn.close();

**Writing a Text or Character File**To do the other side … writing to a file … the code might look like:  
 Scanner in = new Scanner(System.,in);  
 . . .  
 System.out.print("Enter the name of the file: ");  
 String fileName = in.next(); in.nextLine();  
 // Make a File object  
 File outFile = new File(fileName);  
 PrintWriter fOut = new PrintWriter( new FileWriter( outFile ) );  
 and then use fOut.print(), fOut.println(), or fOut.printf().  
  
When we are done with the file, we close it:  
 fOut.close();

**In Class Exercise**Write a class, with a main program, that expects two command line arguments. The first is the name of an input file and the second is the name of an output file.   
 // first, provide code to be sure there ARE two command line arguments  
 . . .  
 String fNameIn = args[0];  
 String fNameOut = args[1];  
Your program should then open the file named fNameIn. It should read this file one line at a time and write each line to the file named fNameOut, after changing the line to all uppercase. Remember to  
 open BOTH files … one for reading with Scanner and one for writing with PrintWriter.  
 loop until there is no more input (look up the hasNextLine() method of Scanner)  
 read one line from the input file  
 change it to all upper case (look up the toUpperCase() method of String)  
 write the changed line to the output file  
 end loop  
 close BOTH files

**IMPORTANT**There are lots of things that can go wrong with I/O and if any occur they cause Java to “throw an exception”. Exceptions are next … but for now write the main program as:  
 public static void main(String[] args) **throws Exception** { … }  
and Java will not bother you about these. The throws Exception is considered poor form and we will abandon it when we learn about Exceptions.

In JGrasp create a text file to use as your input file. Put several lines of text in it (you can copy and paste from a web page) with some upper case letters, some lower case letters, and some non-letters.

**Other IO Classes**There are many other classes to read text files. Actually, FileReader is one, but it only allows you to read 1 character at a time from the file. Furthermore, the character it reads is returned as the integer representation of the character. Similarly, FileWriter will write data to a file, but again only 1 character at a time, and the character must be given as an integer. Scanner and PrintWriter are more useful.  
  
There are also BufferedReader and BufferedWriter classes. The BufferedReader class reads ahead to fill a small storage area in memory (the buffer) with characters. When you ask it to read, it actually takes the characters out of the buffer. It is more efficient to read a bunch of characters into memory at once and then take them one at a time from the buffer. BufferedReader also has a readLine() method that reads one line of text from the file and returns it as a String. When the file has all been read, readLine() returns the special value null.  
  
BufferWriter saves up characters that you write to it until there are too many to fit in the small storage area (the buffer). When the buffer gets full, all of it is written to the file at once … again, this is more efficient. With output like this, you can force the buffer to be written even when it is not yet full with  
 bWriter.flush();

**Catching Exceptions**Go back to your main program from the in class exercise. Remove the **throws Exception** from the main method and try to compile. Now you get errors like …

UpperCase.java:23: error: unreported exception FileNotFoundException; must be caught or declared to be thrown  
 Scanner fIn = new Scanner(new FileReader(new File(fNameIn)));  
 ^  
UpperCase.java:24: error: unreported exception IOException; must be caught or declared to be thrown  
 PrintWriter fOut = new PrintWriter(new FileWriter(new File(fNameOut)));

Java is complaining that when you try to open a file, it might not exist (FileNotFoundException). In that case, it will ‘throw’ a FileNotFoundException and you have to handle it (or catch it as we say in Java). To catch exceptions, you must surround the code that might cause a problem with a try-catch block, such as:

try {  
 Scanner fIn = new Scanner(new FileReader(new File(fNameIn)));  
 PrintWriter fOut =   
 new PrintWriter(new FileWriter(new File(fNameOut)));  
 }  
 catch(FileNotFoundException fnfe) {  
 System.out.println(fnfe);  
 System.exit(1);  
 }  
  
The ‘try’ simply tells Java what code to watch for errors. The ‘catch’ says what to do when a FileNotFoundException occurs inside this code … in this case, print an error message and terminate the program.

A problem here is that the Scanner and PrintWriter are local to the try block and go away when we exit the try block. If we do not want this to happen, we should write this as:

Scanner fIn = null;  
 PrintWriter fOut = null;  
 try {  
 fIn = new Scanner(new FileReader(new File(fNameIn)));  
 fOut =   
 new PrintWriter(new FileWriter(new File(fNameOut)));  
 }  
 catch(FileNotFoundException fnfe) {  
 System.out.println(fnfe);  
 System.exit(1);  
 }

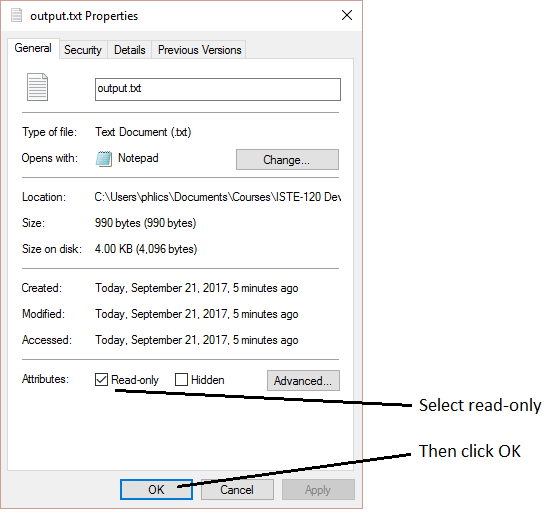
Now, while we still use fIn and fOut in the try block, they are declared in the outer scope and still exist when we exit the try block. Fix the in class example so that exceptions are caught. Try it out, by using an input file that does not exist.

**More On Exceptions**In the above example, another exception can occur … an IOException. This is a pretty general exception and can occur, for example, if you try to open a file for output in a directory in which you do not have write permission. Many other cases of this ilk exist as well.

To handle this, you need to allow for more than one exception to occur in the try block that we created. This is done by having more than one catch at the end of the try block … one for each possible exception. As in:

Scanner fIn = null;  
 PrintWriter fOut = null;  
 try {  
 fIn = new Scanner(new FileReader(new File(fNameIn)));  
 fOut =   
 new PrintWriter(new FileWriter(new File(fNameOut)));  
 }  
 catch(FileNotFoundException fnfe) {  
 System.out.println(fnfe);  
 System.exit(1);  
 }  
 catch(IOException ioe) {  
 System.out.println(ioe);  
 System.exit(1);  
 }

Add code similar to the above to your program for the in class exercise and see if it works. Try it with errors as well. For example:

* Run the program with the first command line argument being the name of a file that does NOT exist in the folder that the program is in. What happens (what error message is printed)?
* Now, run the program with an existing input file, so that the output file is created. Then, in the file explorer, right click on the output file and select Properties from the menu that appears. In the dialog that pops up, check the Read-only box and choose OK. Now you cannot write to the output file.
  + Try to write to the output file by rerunning the program. What happens (what error message is printed)?

If you look closely at the javadocs for IOException and FileNotFoundException you will see that IOException is a subclass of the Exception class and the FileNotFoundException is a subclass of the IOException class. Go back to your program. Delete the catch block for FileNotFoundException and IOException and replace them with one for just Exception. Then try the two tests above again. What are the results??

Since a FileNotFoundException **is-an** Exception and an IOException **is-an** Exception, this one catch block catches both. So it is all you need … unless you want to do something different for a FileNotFoundException from what you want to do for an IOException. In that case, two catch blocks would still be necessary.

When listing catch blocks for several different exceptions, list them starting with the most specialized sub class first (the class furthest from Exception) and with the most general class last. This way, when an exception occurs, it will be caught by the most specialized catch block and the most general catch block will catch everything else.

**Creating Your Own Exceptions**  
If you want to create your own exceptions, you can. For example, suppose you are doing input validation and when an error occurs, you want an InputValidationException to be thrown. In this case you need a new class, called InputValidationException, that extends Exception, as in:  
  
public class InputValidationException extends Exception {  
 public InputValidationException(String \_message) {  
 super(\_message);  
 }  
}

When your code detects an input problem, it should now:  
 throw new InputValidationException("Any message you like.");

**In Class Exercise**To try this out, write the InputValidationException class as show above and get it to compile. Then, write a class TestInputValidationException. This should have a main method which should read in a letter grade, and print out its numerical equivalent (A = 4, B = 3, C = 2, D = 1, F = 0). Do this in a loop until the user enters a Q for quit. If anything other than A, B, C, D, F, or Q is entered, throw an InputValidationException with an appropriate message. Place the entire loop in a try block with a catch block for an InputValidationException. If one occurs, print a message (as above) and terminate the program.