Chapter 4 - Files

- Random Numbers with `Random` class
  - Differences with `Math.random()` method
- File input and output
  - `PrintWriter` - Write Data to a File
  - Exceptions
  - `FileWriter` - Append Data to a File
  - Saving a file to a specific location
  - Reading Data from a File with Scanner
  - Try-with-resources

Generating Random Numbers with the Random Class

Some applications, such as games and simulations, require the use of randomly generated numbers. The Java API has a class, Random, for this purpose.

To use the Random class, use the following import statement and create an instance of the class.

```
import java.util.Random;
Random randomNumbers = new Random();
```

There are several methods for generating random number types using an instance of Random:

```
// Returns the next random number as a double.  The number will be within the
range of 0.0 and 1.0.
randomNumbers.nextDouble();
//Returns the next random number as a float.  The number will be within the
range of 0.0 and 1.0.
randomNumbers.nextFloat();
//Returns the next random number as an int.  The number will be within the
range of an int, which is -2,147,483,648 to +2,147,483,648.
randomNumbers.nextInt();
//This method accepts an integer argument, n.  It returns a random number as
an int.  The number will be within the range of 0 to n.
randomNumbers.nextInt(int n);
```

```
randomNumbers.nextInt(int start, int bound);
```

You might remember the Math object has a method called Random. The primary difference between Java's `Random` class and the `Math.random()` method lies in their functionality and use cases:

`Random` Class

- **Instance-based**: You need to create an instance of the `Random` class to use it.
- **Features**: It provides methods for generating different types of random values, including integers (`nextInt()`), doubles (`nextDouble()`), booleans (`nextBoolean()`), floats (`nextFloat()`), and long values (`nextLong()`).
- **Seeding**: You can provide a seed to the `Random` class, which allows for generating a reproducible sequence of random numbers.
- **More control**: `Random` gives you more control over the range and type of random values generated.

Example:

```
import java.util.Random;
Random random = new Random();
int randomInt = random.nextInt(100);  // Random int between 0 and 99 double
randomDouble = random.nextDouble();  // Random double between 0.0 and 1.0`
```

`Math.random()` Method

- **Static method**: This is a static method and can be called directly without creating an instance.
- **Returns double**: It only generates random numbers of type `double` between `0.0` (inclusive) and `1.0` (exclusive).
- **No control over seed**: It does not allow you to set a seed or control over the range/type of the random numbers.

Example:

```
double randomValue = Math.random();  // Random double between 0.0 and 1.0

// Use Math.random() to generate a random double (0.0 - 1.0)
// multiply by 100 to turn it into an int.
double randomDouble = Math.random();
System.out.println(randomDouble);
```

```
System.out.println(randomDouble*100);
System.out.println(Math.round(randomDouble*100));
System.out.printf("%.0f %n", randomDouble*100);
```

Summary

- `Random` **class**: Offers more flexibility, including generating random integers, floats, booleans, and long values, with an optional seed.
- `Math.random()`: A simpler way to generate a random `double` between `0.0` and `1.0` with no need for creating an instance but less control over the result.

For most purposes, if you need various types of random values, use the `Random` class. If you just need a quick random double, `Math.random()` is convenient.

File Input and Output Overview

Java allows you to create and read files using a few Java objects.

Remember our friends like String, Scanner, Random, Math.

So I'd like to introduce you to the file-based objects we'll be working with:

- PrintWriter
- FileWriter
- File
- Scanner (you already know Scanner)

These objects must be imported, similarly to Scanner, except Scanner is in java.util, while file objects are in java.io:

```
import java.util.Scanner;   // Needed for Scanner class
import java.io.*;           // Needed for File I/O classes
```

> Quickly show the OpenJDK source code location.

PrintWriter - Write data to a file

It's called PRINT-writer because it has `print` and `println` methods like System.output.

- To open a file for text output you create an instance of the PrintWriter class.
  `PrintWriter outputFile = new PrintWriter("StudentData.txt");`
  Pass the name of a file that you want to open as an argument to the PrintWriter object

constructor. **Warning:** if the file already exists, it will be erased and replaced with a new file.
    - Unless you specify a directory, the file is written in the location where the java class is run from.
- The PrintWriter class allows you to write data to a file using the `print` and `println` (and `printf` methods, as you have been using to display data on the screen.
    - Just as with the System.out object, the `println` method of the PrintWriter class will place a newline character after the written data.
    - The `print` method writes data without writing the newline character.

First level Usage:

```
import java.io.*;

public class PrintWriterTesting {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter outputFile = new PrintWriter("PrintStudentData.txt");
        outputFile.println("Chris");
        outputFile.println("Michelle");
        outputFile.println("Michael");
        outputFile.close();
    }
}
```

You can also use the `print()` method to print without the new line. Remember, if you run the program multiple times, you overwrite the file each time.

Second level Usage (using Scanner, for loop, and non-self-contained initializer variable because we want to access it outside the scope of the loop):

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

public class PrintWriterTesting {
    public static void main(String[] args) throws FileNotFoundException {
        int i;
        Scanner kbd = new Scanner(System.in);
        PrintWriter outputFile = new PrintWriter("PrintStudentData.txt");
        for (i=1;i<4;i++){
            System.out.print("Enter student " + i + "'s name: ");
            String studentName = kbd.nextLine();
            outputFile.println(studentName);
        }
```

```
        System.out.println("Wrote " + (i-1) + " student names to file.");
        outputFile.close();
    }
}
```

Printf is also supported

```
outputFile.printf("%20s %n", studentName);
```

However, each time you run the program, the file gets overwritten.
(Run file multiple times to see appending names in `PrintStudentData.txt`).

Exceptions

When something unexpected happens in a Java program, an exception is thrown. Some Java classes REQUIRE you to throw an exception. It is the Java Object's way of forcing you to tell the user why the program didn't work.

You may have noticed when using PrintWriter that the `main` method has additional words:

```
public static void main(String[] args) throws FileNotFoundException
```

The new words in the `main` method signature (`throws FileNotFoundException`) indicate that a user COULD run into the problem of not being able to create a file (or the file doesn't exist in the specified location).

If you hover over PrintWriter in Intellij, you'll see that PrintWriter takes one parameter (`String filename`) and throws one exception (`FileNotFoundException`), which under these conditions:

> *If the given string does not denote an existing, writable regular file and a new regular file of that name cannot be created, or if some other error occurs while opening or creating the file.*

The method that is executing when the exception is thrown (in our case, `main`) must either handle the exception or pass it up the line.

When an exception (an error or unexpected event) occurs in a method, the method has two choices:

1. **Handle the exception**: The method can catch and manage the exception using a try-catch block. This means the method takes responsibility for dealing with the error.
2. **Pass it up the line**: If the method doesn't handle the exception, it must declare this by using a `throws` clause. This means the responsibility for handling the exception is passed to the method that called it.

Here's an example of handling the exception with try/catch:

```
import java.io.*;

public class PrintWriterDemo {
    public static void main(String[] args) {
        try {
            PrintWriter outputFile = new PrintWriter("");
            outputFile.println("Carrie");
            outputFile.println("Donnie");
            outputFile.println("Mike");
            outputFile.close();
        } catch (Exception e){
            System.out.println("Can't write to this location \n" + e );
        }
    }
}
//output
Can't write to this location
java.io.FileNotFoundException:  (No such file or directory)
```

However, we are focused right now on passing the exception up the line. To do this, the method must use a `throws` clause in the method header. To insert a throws clause in a method header, simply add the word throws and the name of the expected exception.

PrintWriter objects throw a `FileNotFoundException`, so we write the throws clause like this:

```
import java.io.*;

public class PrintWriterDemoPassUpLine {
    public static void main(String[] args) throws FileNotFoundException {
        PrintWriter outputFile = new PrintWriter("PrintStudentData.txt");
        outputFile.println("Carrie");
        outputFile.println("Donnie");
        outputFile.println("Mike");
        outputFile.close();
```

```
    }
}
```

There are many different types of Exceptions used by Java Objects. FileWriter, which we'll see next, throws an `IOException`, so we write the throws clause like this:

```
import java.io.*;
public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("names.txt", true);
        fw.write("This is being written by FileWriter!");
        fw.close();
}
```

Here is a list of common Java Errors and Exception Types.

NOW, you may be thinking, what if I use two objects that each throw different exceptions? In some cases you must declare each exception after the `throws` keyword in the method signature:

```
public static void main(String[] args)
        throws FileNotFoundException, IOException {
        // main method that uses PW and FW.
}
```

However, there is actually a hierarchy of exceptions. In this case, the `FileNotFoundException` is a specific exception within `IOException`. If you hover the `FileNotFoundException` in IntelliJ, you will see this message:

```
There is a more general exception, 'java.io.IOException', in the throws list
already.
```

So you can safely remove the `FileNotFoundException` exception and only keep the `IOException`.

```
public static void main(String[] args) throws IOException {
        // main method that uses PW and FW.
}
```

FileWriter - Append data to a file

To avoid erasing a file that already exists, create a FileWriter object, then create a PrintWriter object in this manner:

```
FileWriter fw = new FileWriter("names.txt", true);
PrintWriter outputFile = new PrintWriter(fw);
```

The two statements can be combined into a single statement:

```
PrintWriter outputFile = new PrintWriter(new FileWriter("names.txt", true));
```

The reason to use these two objects (`PrintWriter` and `FileWriter`) together, is because `FileWriter` has this constructor:

```
// Constructor
FileWriter(File file, boolean append)
// Example
FileWriter fw = new FileWriter("names.txt", true);
```

Calling FileWriter with these two arguments creates a **FileWriter** object using a named **File** object (whether it exist or not.. just as long as its location is accessible).

If the second argument is `true` , bytes will be written to the end of the file rather than the beginning (letting us append data rather than overwrite). It throws an **IOException** under these conditions:

> *if the named file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason*

After you instantiate the `FileWriter` object, write to the file as you would with the `PrintWriter` methods (`print` , `println` , or `printf`). Full example:

```
import java.io.*;

public class PrintWriterTestWithFileWriter {
    public static void main(String[] args) throws IOException {

        FileWriter fw = new FileWriter("names.txt", true);
        PrintWriter outputFile = new PrintWriter(fw);
        outputFile.println("Bobby");
        outputFile.println("Billy");
        outputFile.println("Jackson");
```

```
        outputFile.close();

        }
}
```

(Run file multiple times to see appending names in `names.txt`).

What's the difference between FW and PW anyway?

Specify a location for your file

On a Windows computer, paths contain backslash () characters. If the backslash is used in a string literal, you must use two of them because backslash is the Escape Character (for example, newline is `\n` in a String.

Therefore you must tell the compiler to not treat the backslash as an escape by typing two of them in a row). For example:

```
PrintWriter outFile = new PrintWriter("D:\\PriceList.txt");
```

A location you may want to use is your OneDrive account. You would specify the path like so:

```
PrintWriter outFile = new PrintWriter("C:\\Users\\pete\\OneDrive - Bentley
University\\CS180\\names.txt");
```

But typing this all out can be cumbersome. Remember our friends `System.out` and `System.in`? The System object has many methods and properties that you can access. One property is `user.home`:

```
System.getProperty("user.home")
// /Users/pete or C:\Users\pete
```

You can make use of this information to make your program more portable:

```
FileWriter fw = new FileWriter(
        System.getProperty("user.home") +
                "/Documents/OneDrive - Bentley University/names.txt", true);
```

If I shared this program with you, FileWriter wouldn't look for the `/Users/pete` directory, but whatever your home directory is.. regardless whether you're on Mac or Windows.

TODO

Information from the slides that I need to test for Tuesday:

- Double backslash is only necessary if the backslash is in a string literal. If the backslash is in a String object then it will be handled properly. Need to test String literal vs. String object.
- Java allows Unix style filenames using the forward slash (/) to separate directories. Need to test this on Windows. It works on Mac, but Mac is basically unix.
- Java for accessing System variables. On Windows, there is an environment variable for OneDrive.
- What is the difference between FileWriter and File? You can still write to a File object.

Stray observations

You must close the PrintWriter object or nothing will be written to the file!!

File and Scanner - Reading data from a file

File Object Methods

```
import java.io.*;
// File object demonstration.
// Uses methods such as getName(), getAbsolutePath(), exists(), mkdir(),
//      delete(), renameTo(), and length() (size in bytes).

public class FileDemoByPeter {
    public static void main(String[] args) throws IOException {

        // Create or Overwrite data to a file called friends.txt in current
directory.
        File oldFile = new File("friends.txt");
        // A file can also be a directory if no file type is specified.
        File dir = new File("testDirectory");

        // Add file contents using PrintWriter.
        PrintWriter fileContents = new PrintWriter(oldFile);
        fileContents.println("First line of the file.");
        fileContents.printf("Second line of the file called %s %n",
oldFile.getName());
                fileContents.printf("Absolute path of this original file: %s
%n", oldFile.getAbsolutePath());
                fileContents.print("This is the last line of the file! \n");
```

```
        // File isn't written until PrintWriter is closed.
        System.out.printf("Size of UNCLOSED file in bytes: %d %n",
oldFile.length());
        fileContents.close();
        // File gets written with new content after PrintWriter is closed.
        System.out.printf("Size of CLOSED file in bytes: %d %n",
oldFile.length());

        // The File object has exists() and mkdir() methods.
        // Use a Unary operator to say: if directory does not exist, then make
it!        if (!dir.exists()){
            dir.mkdir();
        }

        // Specify the destination location for the file
        File newFile = new File(dir + "/" + oldFile);

        if (newFile.exists()){
            newFile.delete();
            // Move file to new directory.
            oldFile.renameTo(newFile);
        }

        System.out.println("Old filepath: " + oldFile.getAbsolutePath());
        System.out.println("New filepath: " + newFile.getAbsolutePath());

        System.out.println("Can new file be read? " + newFile.canRead());
        System.out.println("Can new file be written to? " +
newFile.canWrite());
        System.out.println("Can new file be executed? " +
newFile.canExecute());

    }
}
```

File with Scanner

You use the File object and the Scanner object to read data from a file:

```
File myFile = new File("Customers.txt");
Scanner inputFile = new Scanner(myFile);
```

Usage example:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FileDemoByPeterWithScanner {
    public static void main(String[] args) throws FileNotFoundException {
        File myFile = new File("testDirectory/friends.txt");
        Scanner inputFile = new Scanner(myFile);
        /*
        // Read the first line of the file.
        String str = inputFile.nextLine();
        System.out.println(str);        */
        // Read and print every line using a while loop and Scanner's
hasNext() method.        while (inputFile.hasNext()) {
            String str = inputFile.nextLine();
            if (str.contains("Second")){
                System.out.println("FOUND!");
            }
            System.out.println(str);

        }

        inputFile.close();
    }
}
```

`try` -with-resources

The try-with-resources statement can be used to open a file and automatically close the file.

PrintWriter Example:

```
import java.io.*;
public class TryWithResources {
    public static void main(String[] args) throws FileNotFoundException {
        try (PrintWriter outputFile = new PrintWriter("MusicArtists.txt")){
                    outputFile.println("The Cure");
                    outputFile.println("The Smiths");
                    outputFile.println("Joy Division");
                    outputFile.println("Green Day");
                    outputFile.println("The The");
                    outputFile.println("The Beach Boys");
                    outputFile.println("Talking Heads");
                    outputFile.println("Neutral Milk Hotel");
```

```
                    outputFile.println("The Magnetic Fields");
                    outputFile.println("The Velvet Underground");
                }
    }
}
```

- This example opens a file named MusicArtists.txt
- Code inside the braces can use the PrintWriter object to write data to the file
- When the code inside the braces has finished, the file is automatically closed

Level 1 Scanner Example:

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

public class TryWithResources {
    public static void main(String[] args) throws FileNotFoundException{
                int currentLine = 1;
        try (Scanner inputFile = new Scanner(new File("MusicArtists.txt"))){
            while(inputFile.hasNext()){
                String str = inputFile.nextLine();
                System.out.printf("%-5d %s %n", currentLine, str);
}
        }
    }
}
```

- This example opens a file named MusicArtists.txt
- Code inside the braces can use the Scanner object to read data from the file
- When the code inside the braces has finished, the file is automatically closed

Level 2 Scanner Examples:

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

public class TryWithResources {
    public static void main(String[] args) throws FileNotFoundException{

        int theAccumulator = 0;
        int bandsWithTheInTheName = 0;
        int bandsWithoutTheInTheName = 0;
        int currentLine = 1;
```

```
        try (Scanner inputFile = new Scanner(new File("MusicArtists.txt"))){
            while(inputFile.hasNext()){
                String str = inputFile.nextLine();
                System.out.printf("%-5d %s %n", currentLine, str);
                if (str.contains("The")){
                    bandsWithTheInTheName++;

                    int firstThe = str.indexOf("The");
                    int secondThe = str.lastIndexOf("The");
                    if (firstThe != secondThe){
                        theAccumulator+=2;
                    } else {
                        theAccumulator++;
                    }
                } else {
                    bandsWithoutTheInTheName++;
                }
                currentLine++;
            }
        }

        System.out.printf("%n%-30s %d %n", "Bands with \"The\" in name:",
bandsWithTheInTheName);
        System.out.printf("%-30s %d %n", "Bands without \"The\" in name:",
bandsWithoutTheInTheName);
        System.out.printf("%-30s %d %n", "Total \"The\"s:", theAccumulator);
    }
}
```

Opening multiple files

Opening two files with a try-with-resources statement:

```
try (Scanner inputFile = new Scanner(new File("File1.txt"));
     PrintWriter outputFile = new PrintWriter("File2.txt")){
   // Statements that work with both files…
}
```

- This example opens a file named File1.txt for reading and a file named File2.txt for writing
- Code inside the braces can use the Scanner object to read data from File1.txt and the PrintWriter object to write data to File2.txt
- When the code inside the braces has finished, both files are automatically closed