Week 4 Review

Types of Exceptions

Runtime exceptions are errors that occur while the program is executing in the JVM.

- Example:
    - **`ArrayIndexOutOfBoundsException`**: An element of an array was accessed with an index that is out of bounds.

Checked exceptions are exceptions which are defined as needing to be handled or declared in Java code.

- Example:
    - **`FileNotFoundException:`** This is thrown programmatically, when the program tries to do something with a file that doesn’t exist.

"Throwing" and Handling Exceptions

Throwing means making everyone aware that a deviation from normal program flow has occurred. You may also hear this referred to as *raising* an exception but Java uses the "throw" concept.

- The JVM may throw an exception behind the scenes.
- Java code may also explicitly throw exceptions to indicate exception situations have occurred.
- All Exceptions, whether they are Runtime Exceptions or Checked Exceptions, inherit from the **`Exception`** class.

Checked Exception Handling Flow

Wrapping code that needs to handle a checked exception in a try-catch block:

- Attempts to execute the code in the try block.
- If it succeeds, it moves past the try-catch block.
- If an exception of the type specified in the catch block is thrown, the code within the catch block will be executed before moving forward.

```
public static void main(String[] args)  {

      ExceptionExamples examples = new ExceptionExamples();

      try {
            String data = examples.readLineFromFile(null);
      } catch (FileNotFoundException e) {
            // do something to handle the exception
      }

}

public String readLineFromFile(String path)
      throws FileNotFoundException {

      String result ="";

      if (path == null) {
            throw new FileNotFoundException("null path");
      }

      // open and read file...

      return result;
}
```

The code within the **`try`** block is attempted.

If the specified exception is thrown, the code in the **`try`** block will be interrupted and the code within the **`catch`** block will be executed instead.

A Closer Look at the Try-Catch-Finally Block

```
try {
     String data =
          examples.readLineFromFile(null);
     System.out.println("This is the data: " + data);
} catch (FileNotFoundException fnfe) {
     System.out.println("File Error: "
          + fnfe.getMessage());
} catch (IllegalArgumentException iae) {
     System.out.println("Argument Error: "
          + iae.getMessage());
} finally {
     System.out.println("Ok... all done!");
}
```

Code in **`try`** block will attempt to execute.

If an exception is thrown, the flow is interrupted so this line will not be executed.

The **`catch`** block will check for **`FileNotFoundException`** and if that's what thrown, this code will execute. If not, it will move on to check for another exception if multiple catch blocks are included.

The **`finally`** block is optional, but if included, it will run after the rest of the code in the try-catch block is executed no matter whether an exception was thrown or not.

The exception object has a **`getMessage()`** method to get any message the was included when the exception was thrown.

A Few More Notes on Exceptions...

- Since all exceptions are inherited from the **`Exception`** class, We can make use of polymorphism in our code.
    - We can catch many different kinds of exceptions with a single catch block (if it makes sense).
- We can create our own exceptions to handle possible scenarios that arise in our code which we may want to throw exceptions for.
- When a runtime exception that we don't catch is thrown, we will see something called a **stack trace**. The stack trace shows you the whole stack of method calls that were made to get to the point where the exception was thrown and are invaluable for figuring where code went wrong.
    - The **`Exception`** object that is available in catch blocks, can be used to read, and even print, the stack trace.

File Input : The File Class

The file class is the Java class that encapsulates what it means to be a file containing data. This is an instantiation of a **`File`** object.

```
File inputFile = new File("testFile.txt");
```

The simplest form of the the File constructor takes a **`String`** indicating the path of the file to open.

The **`File`** class can be used to perform many actions on files or directories in the filesystem.

Data Streams

Methods exist to read all text in very quickly with one line of code. They pull the entire file into memory though.Typically, we don't want to do that, especially for large files. This is equivalent of sitting to watch a NetFlix movie and waiting for the entire movie to load before you start watching it!!!!

- A **stream** refers to a sequence of bytes that can read and write to some sort of backing data store.

- A stream is like an assembly line, where you process each thing as it comes through, in order.

- The **`Scanner`** class we used to read keyboard input is a way to read data streams.

File + Scanner

When we attempt to create the **`Scanner`** with a **`File`**, the compiler will force us to handle or re-throw **`FileNotFoundException`** so for now we will modify **`main`** to state that it re-throws it.

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

     File inputFile = new File("rtn.txt");

     if (inputFile.exists()) {
          try (Scanner scanner = new Scanner(inputFile)) {
               while(scanner.hasNextLine()) {
                    String lineFromFile = scanner.nextLine();
                    System.out.println(lineFromFile);
               }
          }
     }

}
```

Create a **`File`** object with path **`rtn.txt`**

Use the **`File`** **`exists()`** method to check if the FIle exists before attempting to open it

Create a **`Scanner`** object using the **`File`** object as the stream to read.

`Scanner` has a method that checks if it has any more lines in the file. We can loop while **`hasNextLine()`** is true.

Read each line into a String using the **`Scanner`** **`nextLine()`** method.

Did you notice that the **`Scanner`** was created as a parameter of a **`try`** block? What's that about? Stay tuned...

The Try With Resources Block

Now let's address that weird way Scanner was created as a parameter of a **`try`** block...

```
try (Scanner scanner = new Scanner(inputFile)) {
     while(scanner.hasNextLine()) {
          String lineFromFile = scanner.nextLine();
          System.out.println(lineFromFile);
     }
}
```

In the past, opening and closing resources often involved repeating code to make sure that resources got closed whether the code ran as expected or wound up in a **`catch`** block due to an exception.

The **try-with-resources** version of a **`try`** block was created to handle this. If you create a resource as a parameter of a **`try`** block, the resource will be closed as soon as the **`try`** block is exited, whether that is through normal flow or when an exception is thrown and the code jumps to a **`catch`** block.

Note that a try-with-resources block can be used to provide this "auto-closing" mechanism even in scenarios where it is not necessary to catch an exception. In this special case, the try block does not always require a **`catch`** or **`finally`** clause.

Catching FileNotFoundExceptions

This is an example of handling the possibility of a **`FileNotFoundException`** when opening a **`File`** rather than having the method pass the buck up the chain.

Here we use the try-with-resources block to create the **`Scanner`** resource and add a **`catch`** block to handle the possible exception.

Note that the **`main`** method no longer re-throws the exception.

```
public static void main(String[] args)
                  {

     File inputFile = new File("rtn.txt");

     if (inputFile.exists()) {
          try (Scanner scanner = new Scanner(inputFile)) {
               while(scanner.hasNextLine()) {
                    String lineFromFile = scanner.nextLine();
                    System.out.println(lineFromFile);
               }
          } catch (FileNotFoundException e) {
               System.out.println("File not found.");
          }
     }
}
```

Finding Additional Info About a File Path

The **`File`** class has several methods which can be used to find extra information about the specified path. You have already seen **`exists()`** but here are a few of the available methods of this sort:

- **`getName()`** - returns name of file (just the name, not any path info)
- **`getAbsolutePath()`**- we saw this yesterday… returns absolute path of the File
- **`exists()`** - indicates whether the file or directory pointed to by File exists
- **`isDirectory()`** - indicates whether the path points to a directory
- **`isFile()`** - indicates whether the path points to a directory
- **`length()`** - size of file in bytes

Using the File Class to Create a Directory

One of the things the File class can be used for is creating a new directory.

Create new **`File`** object with path of directory to create as the param.

```
File newDirectory = new File("myDirectory");

if (newDirectory.exists()) {
     System.out.println("Sorry, "
     + newDirectory.getAbsolutePath() + " already exists.");
}
else {
     newDirectory.mkdir();
}
```

Check if the directory already exists using the **`File`** object's **`exists()`** method before creating it.

Call the **`File`** object's **`mkdir`** (remember that?) command to create the directory.

Using the File Class to Create a File

We can also use the File class to create a file (also relative to the root of the project).

Must use a **`try-catch`** block here since the **`createNewFile`** method of the **`File`** class declares it may throw an **`IOException`**.

```
try {
     File newFile = new File("myDataFile.txt");
     newFile.createNewFile();
} catch(IOException e) {
     System.out.println("Exception occurred: "
          + e.getMessage());
}
```

Create new **`File`** object with path of file to create as the param.

Catch and handle the possible **`IOException`**.

Use the **`createNewFile`**() method of the **`FIle`** class to create the file on the filesystem

Using the File Class to Create a File in a Directory

The **`File`** class has an overloaded constructor which takes an extra parameter specifying the path in which the file or directory should be created should be created.

```
File newFile = new File("myDirectory", "myDataFile.txt");
```

Extra param which tells **`File`** to put the path in the second parameter into the directory at this path. Can be used to add a file to the directory or create a subdirectory.

Introducing… Buffers

A **buffer** is like a bucket to which text is initially written. It is only after we invoke the `.flush()` method that the bucket's contents are transferred to the file. `Printwriter` creates a buffered stream that gets flushed when the buffer is full or `.flush()` is manually called and the `Printwriter` is closed.

```
try (PrintWriter writer = new PrintWriter(newFile.getAbsoluteFile())) {
     writer.print(message);
} catch(FileNotFoundException e) {
     System.out.println("File does not exist.");
}
```

Remember how we mentioned the try-with-resources block was created to avoid writing lots of repetitive, cluttered code? The try block in the previous example can be rewritten like this - the try-with-resources takes care of flushing the buffer and closing the `Printwriter` resource when the try block exits!!!!

File Append Example

We set a **boolean** indicating whether to append based on whether the file being written to already exists.

```
File newFile = new File("myDataFile.txt");
String message = "Appreciate\nElevate\nParticipate";

boolean append = newFile.exists() ? true : false;
try (PrintWriter writer =
     new PrintWriter(new FileOutputStream(newFile, append))) {
     writer.append(message);
} catch(IOException e) {
     System.out.println("Exception: " + e.getMessage());
}
```

We use the **append** method of **PrintWriter** which will append if the stream it is created with is set to append.

We create the **PrintWriter** using a **FileOutputStream**, which is created using the **File** object and the **boolean** we created as the param which indicates whether or not to append.