Module 1-17

Exceptions
File Input

Objectives

- Students should be able to describe the concept of exception handling
- Students should be able to implement a try/catch structure in a program
- Students should know about java.io library File and Directory classes
- Students should be able to explain what a character stream is
- Students should be able to use a try-with-resources block
- Students should know how to handle File I/O exceptions and how to recover from them
- Students should know how File I/O might be used on a job

Exceptions

What are Exceptions?

Exceptions are occurrences that alter the flow of the program away from the ideal or “happy” path.

- *Sometimes it’s the developer’s fault*: i.e. accessing an array element greater than the actual number of elements present.
- *Other times it’s not*: i.e. loss of internet connection, a data file that was supposed to be there has been removed by a systems admin.

Checked vs. Unchecked Exceptions

- Checked are compile-time exceptions
  - If code in a method throws a checked exception, method must handle it
    - Handle in method or pass up to parent
- Unchecked are run-time exceptions
  - User or code does something that causes program to stop running

Compile-time Exceptions (Checked Exceptions)

They are not runtime exceptions, but they must be handled or declared.

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

- **IOException**: A more general exception related to problems reading or writing to a file.
  - Note that FileNotFoundException extends from IOException.

Runtime Exceptions (Unchecked Exceptions)

Runtime exceptions are errors that occur whilst the program is executing in the JVM. Here are three common examples:

- **NullPointerException**: you tried to call a method or access a data member for a null reference.
- **ArithmeticException**: you tried to divide by zero.
- **ArrayIndexOutOfBoundsException**: you tried to access an array element with an index that is out of bounds.

Exceptions “Throwing”

Throwing means making everyone aware that a deviation from normal program flow has occurred.

- Throwing can be done behind the scenes by the JVM.
- It can be triggered via code, by using the *throw* statement.

Exceptions “Handling”

Handling are the action takens (defined by the programmer) when an exception is encountered.

Try / Catch

The Try Catch block follows the following format:

```
try {
   // Code where an exception might be triggered
}
catch (FileNotFoundException e) {
   // Catch and specify actions to take if an exception is encountered.
}
finally {
   // Action to take regardless of whether an exception was encountered.

}
```

Both the catch and finally blocks are optional.

Try / Catch

```
16          System.out.println("The following cities: ");
17          String[] cities = new String[] { "Cleveland", "Columbus", "Cincinatti" };
18          try {
19              System.out.println(cities[0]);
20              System.out.println(cities[1]);
21              System.out.println(cities[2]);
22              System.out.println(cities[3]);  // This statement will throw an ArrayIndexOutOfBoundsException
23              System.out.println("are all in Ohio."); // This line won't execute because the previous statement throws an Exception
24          } catch(ArrayIndexOutOfBoundsException e) {
25              // Flow of control resumes here after the Exception is thrown
26              System.out.println("XXX   Uh-oh, something went wrong...   XXX");
27          }
```

Exceptions Handling: Example

Consider the following example:

```
import java.io.FileNotFoundException;

public class SuspiciousClass {

   public void doSomething() throws
                  FileNotFoundException {

      throw new FileNotFoundException();
   }

}
```

An exception is programatically thrown.

```
public class MyMainClass {

   public static void main(String[] args)  {
      SuspiciousClass test = new
                     SuspiciousClass();
      test.doSomething();
   }
}
```

Java will complain as we try to invoke doSomething() as it expects us to handle or catch the exception.

Exceptions Handling: Example

Our first choice is to just state that on the main method (from which we call doSomething) that there is a possibility an exception will be thrown:

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

      SuspiciousClass test = new SuspiciousClass();
      test.doSomething();

}
```

Exceptions Handling: Example

Or, we could use a try / catch block to both catch the exception and specify a set of actions to do in the event we run into the exception.

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

    SuspciousClass test = new SuspciousClass();

    try {
        test.doSomething();
    }
    catch (FileNotFoundException e) {
        System.out.println("ok... that's fine, moving on.");
    }
}
```

You must specify the name of the exception along with a placeholder variable.

File Input

File Input

Java has the ability to read in data stored in a text file.

It is one of many forms of inputs available to Java:

- Command Line user input (we have covered this one)
- Through a relational database (Module 2)
- Through an external API (Module 4)

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.

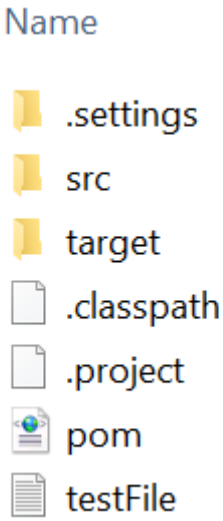
File <<variable name>> = **new File**(**<<Location of the file>>**);

In its simplest form it has a constructor that takes in the location of the file (including the name). Here is a concrete example:

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

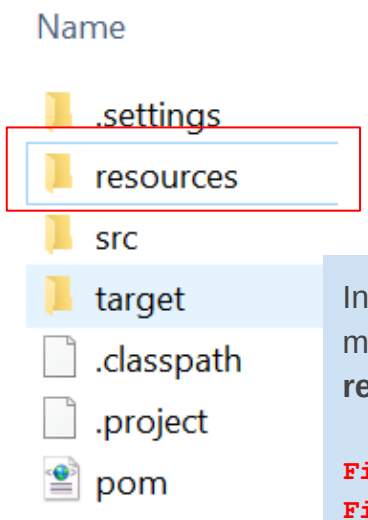
File Input : The File Class

The file location corresponds to the root of that particular Java project. Again, in this example our file is testFile.txt:



In this example, testFile.txt is located in the project root, we can refer to it like so:

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



In this example, testFile.txt has been moved **inside a folder called resources**.

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

File Input : The File Class Methods

There are several methods of the file class that can be used for file input:

- **.exists()**: returns a boolean to check to see if a file exists. We would not want to proceed to parse a file if the file itself was missing!

- **.isFile()**: returns a boolean to check to see if what we are looking at is a File. Returns false if it is not a file (perhaps a folder)
- **.getAbsoluteFile()**: returns the same File object you instantiated but with an absolute path. You can think of this as a getter. It returns a File object.

File and Scanner

A File object and a Scanner object will work in conjunction with one another to read the file data.

Once a file object exists, we instantiate a Scanner object with the file as a constructor argument. Previously, we used System.in as the argument.

File and Scanner: Example

Consider this example:

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

     File inputFile = new File("resources/testFile.txt");

     if (inputFile.exists()) {
          System.out.println("found the file");
     }

     try Scanner inputScanner = new Scanner(inputFile)) {

        while (inputScanner.hasNextLine()) {
            String lineInput = inputScanner.nextLine();
            String [] wordsOnLine = lineInput.split(" ");

            for (String word : wordsOnLine) {
                 System.out.print(word + ">>>");
            }
        }
    }
}
```

We need to handle an exception, but we can pass it up to the parent class.

New file object being instantiated.

Instantiating a scanner but using an “absolute path” file.

The while loop will iterate until it has processed all lines.

File and Scanner: Example

Here is a cleaner version of the example:

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

    File inputFile = new File("resources/testFile.txt");

    if (inputFile.exists())
         System.out.println("found the file");
    }

    try (Scanner inputScanner = new Scanner(inputFile)) {

        while (inputScanner.hasNextLine()) {
            String lineInput = inputScanner.nextLine();
            String [] wordsOnLine = lineInput.split(" ");

            for (String word : wordsOnLine) {
                System.out.print(word + ">>>");
            }
        }
    }
}
```