

Module 1 Day 16

Exceptions

File I/O: Processing File Input

Module 1 Day Unit 16

Can you ... ?

1. ... describe the concept of exception handling
2. ... implement a try/catch structure in a program
3. ... use and discuss the `java.io` library's `File` and `Directory` classes
4. ... explain what a character stream is
5. ... use a try-with-resources block
6. ... handle File I/O exceptions and write code to recover from them
7. ... talk about ways that File I/O might be used on the job

Exceptions

What are Exceptions?

Exceptions are events that alter the flow of the program away from the intended, 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.

Runtime 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.

Checked Exceptions

A checked exception is a type of exception that must be either caught or declared in the method in which it is thrown. For example, the `java.io.IOException` is a checked exception.

- **FileNotFoundException:** This is thrown programmatically, when the program tries to do something with a file that doesn't exist.
 - We just saw this!
- **IOException:** A more general exception related to problems reading or writing to a file.
 - Note that `FileNotFoundException` extends from `IOException`.

Bottom Line: They are not runtime exceptions, but they ***must*** be handled or declared as thrown.

Exceptions “Throwing”

Throwing means immediately halting execution and issuing a warning to make everyone aware that some deviation from normal program flow has occurred.

- Throwing can be done behind the scenes by the JVM. As is the case for RunTime Exceptions and handled exceptions.
- It can also be triggered manually via code by using the *throw* statement. This allows us to use logical tests to create our own exceptions when necessary.

Exceptions “Handling”

Exception handlers are the actions taken (defined by the programmer) when an exception is encountered.

Exceptions Handling: Example

Code declared as throwing an exceptions must have that exception handled by the caller:

```
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. This pattern of “passing the buck” or “hot potato” is a bad practice and simply elevates the error further and further up in the application, potentially making it more difficult to address.


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

Exceptions Handling: Example

Instead, we should use a try / catch block to both catch the exception and specify a set of actions to execute when we run into the **caught** exception.

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



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.

File Input

File Input: The “I” in File I/O

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

It is just one of many forms of inputs available in Java. Others include:

- Command Line user input (From Module 1 Week 2)
- Through a relational database (Coming Soon! In Module 2)
- Through a web interface using the Spring framework (Module 3)
- Through an external API (Module 3)

File Input : The File Class

The **File** class is the class that encapsulates what it means to be a file in the file system. File objects are instantiated as :

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

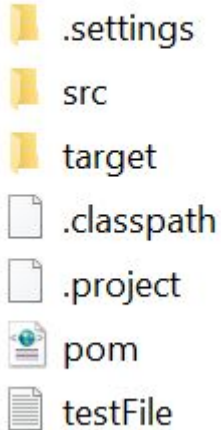
In its basic form, it has a constructor that takes in the location of the file (including the name and extension). As a concrete example:

```
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:

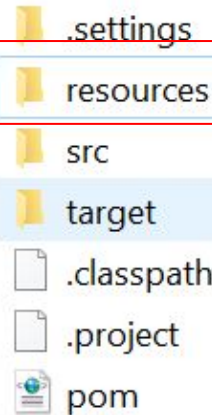
Name



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

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

Name



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 two methods of the file class that are essential 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!
- **.getAbsolutePath()**: 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.

The .exists() method can be used to notify the user and prompt them to enter a valid file or ***throw an exception*** to a method that requests user input for re-entry.

File and Scanner

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

Once a file object exists, we instantiate a Scanner object with the file as a constructor argument just as we used `System.in` as the constructor argument in prior weeks.

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.getAbsolutePath())) {  
  
        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, more on this later.

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

Without all the markup and callouts ...

```
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.getAbsolutePath())) {  
  
        while (inputScanner.hasNextLine()) {  
            String lineInput = inputScanner.nextLine();  
            String [] wordsOnLine = lineInput.split(" ");  
  
            for (String word : wordsOnLine) {  
                System.out.print(word + ">>>");  
            }  
        }  
    }  
}
```