Module 1-16

File Input
Exceptions

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 a web interface using the Spring framework (Module 3)
- Through an external API (Module 4)

File Input : The File Class

The **File** class (from the java.io package) allows us to operate on files and directories. This is an instantiation of a File object.

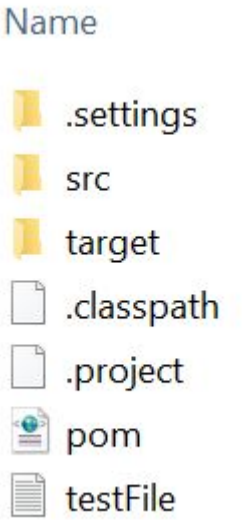
File <<variable name>> = **new File**(**<<Location of the file>>**);

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"**);

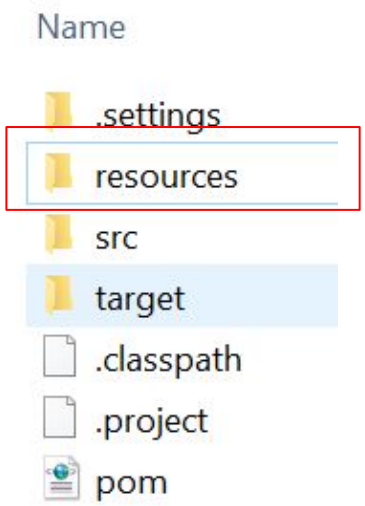
File Input : The File Class

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 created **inside a folder called resources**.

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

File Input : The File Class Methods

Like all classes, File has useful methods, for example:

- **`.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!

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.
- Recall that previously we used System.in as the argument.

File and Scanner: 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, 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.

Let’s create a file scanner

Exceptions

Vocab

There are generally three categories of things that can ruin your day:

- **Compile-time errors**: Syntax problems in your code, code does not compile.
- **Exceptions**: Your code compiles but encounters unexpected issues during runtime.
- **Errors**: Your code compiles but encountered a very serious issue during runtime.

Our focus today will be on exceptions.

Checked vs Unchecked Exceptions

- A **checked exception** is verified during compilation. If you are calling a method that throws such an exception than the compiler will force you to deal with it somehow.
- An **unchecked exception** is not verified during compilation. They happen, and you may choose to deal with it in your code.

We'll discuss what "dealing with it" means.

Common Checked vs Unchecked Exceptions

Checked	Unchecked
IOException: Unexpected issue while reading input or creating output.	**NullPointerException**: Attempting to use an object whose reference points to null.
FileNotFoundException: Java could not open a file. (actually a subclass of IOException)	**ArrayOutOfBoundsException**: Attempting to access an index that is out of bounds for the current data structure.
	Arithmetic Exception: Attempting to divide by zero.

Let’s look at some examples

Dealing with Exceptions

Two ways of addressing exceptions in your code:

- Use *throws*
- Use a try-catch block

Throws vs Try-Catch

- In plain English if I say that my method ***throws*** a given exception I am saying:

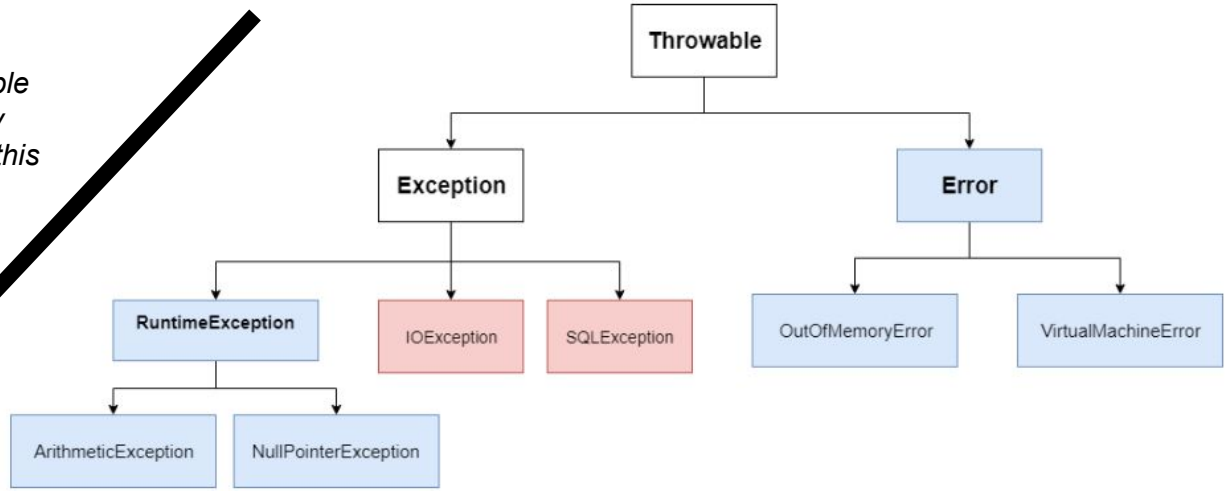
I am aware that this code I'm calling might produce an exception. The method that calls my method will have to deal with it. Hey, at least I'm letting them know!

- Plainly speaking a **try-catch** is like saying:

My code could produce an exception but I will try running it anyways. If the exception happens, I will write special code to deal with it so my program does not terminate abruptly.

Exceptions and Errors

We can catch multiple exceptions, but they must conform to to this hierarchy



Blue ones represent "Unchecked Exceptions" whereas Red ones represent "Checked Exceptions". Source

Source: https://medium.com/@recepinancc/til-19-checked-vs-unchecked-exceptions-731c10ce4ec2

Exceptions Hierarchy

If you include multiple exceptions, the order must be from specfic to general.

```
// This will compile:
try {
        output = thisObj.myMethod();
} catch (SQLException e) {
        e.printStackTrace();
} catch (Exception e) {
        e.printStackTrace();
}
```

```
// This won't compile:
try {
        output = thisObj.myMethod();
} catch (Exception e) {
        e.printStackTrace();
} catch (SQLException e) {
        e.printStackTrace();
}
```

Throw

- The singular ***throw*** is used to purposely create an exception.
- Do not confuse this with *throw**s***, which we discussed a few slides ago!
- Generally speaking we use throw in conjunction with custom exceptions

Throw example

```
int i = 5;
try {
		if (i%2 != 0) {
			throw new IOException();
			//System.out.println("Dead code");
		}
} catch (Exception err) {
		System.out.println("I hate odd numbers");
}

System.out.println("Nothing to see here.");
```

- Within the try block, nothing after the throw statement will run, in fact the compiler will flag it as unreachable code.
- Given that i is 5, the expected output of this code will be:

  I hate odd numbers
  Nothing to see here.

- If i were an even number, the output would just be:

  Nothing to see here.

Create Your Own Exception

- We can use some simple inheritance to implement our own exceptions.

```
public class MaxAmountException extends RuntimeException {

    public MaxAmountException()
    {
        super("Exceeded maximum amount");
    }
}
```

- We can then “invoke” the exception with the **throw** statement:

```
throw new MaxAmountException();
```

Let’s see an example of throwing a custom exception.