Exceptions & File I/O Part 1

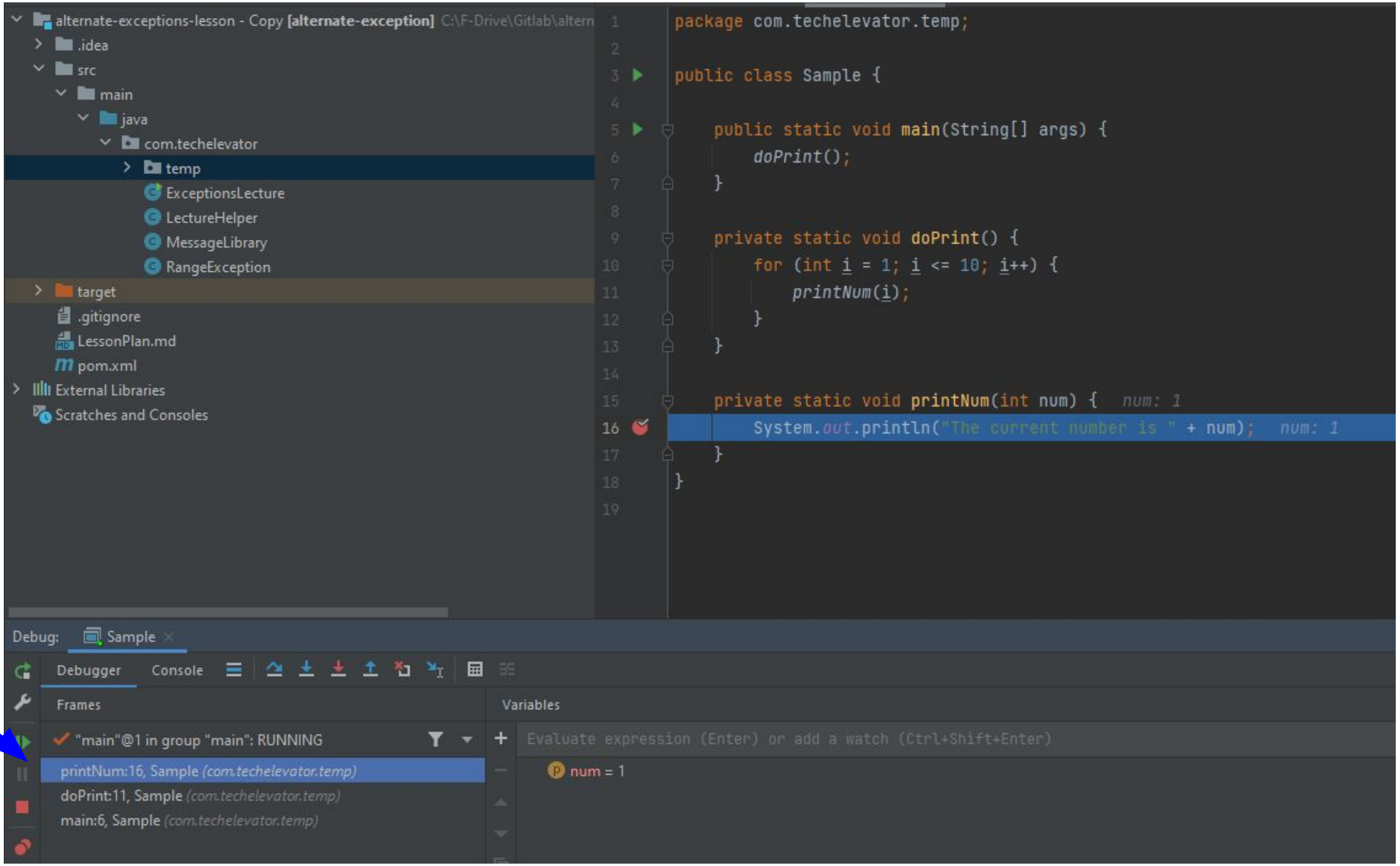
Today's Objectives

- Understanding exception handling
- Implementing a try/catch structure in a program
- **`java.io`** library **`File`** and **`Directory`** classes
- Character streams: what are they and how do we use them?
- Using the try-with-resources block
- Handling File I/O exceptions and how to recover from them
- Real world uses for File I/O

The Call Stack

main calls the **doPrint** method, which calls the **printNum** method.

This shows the call stack.



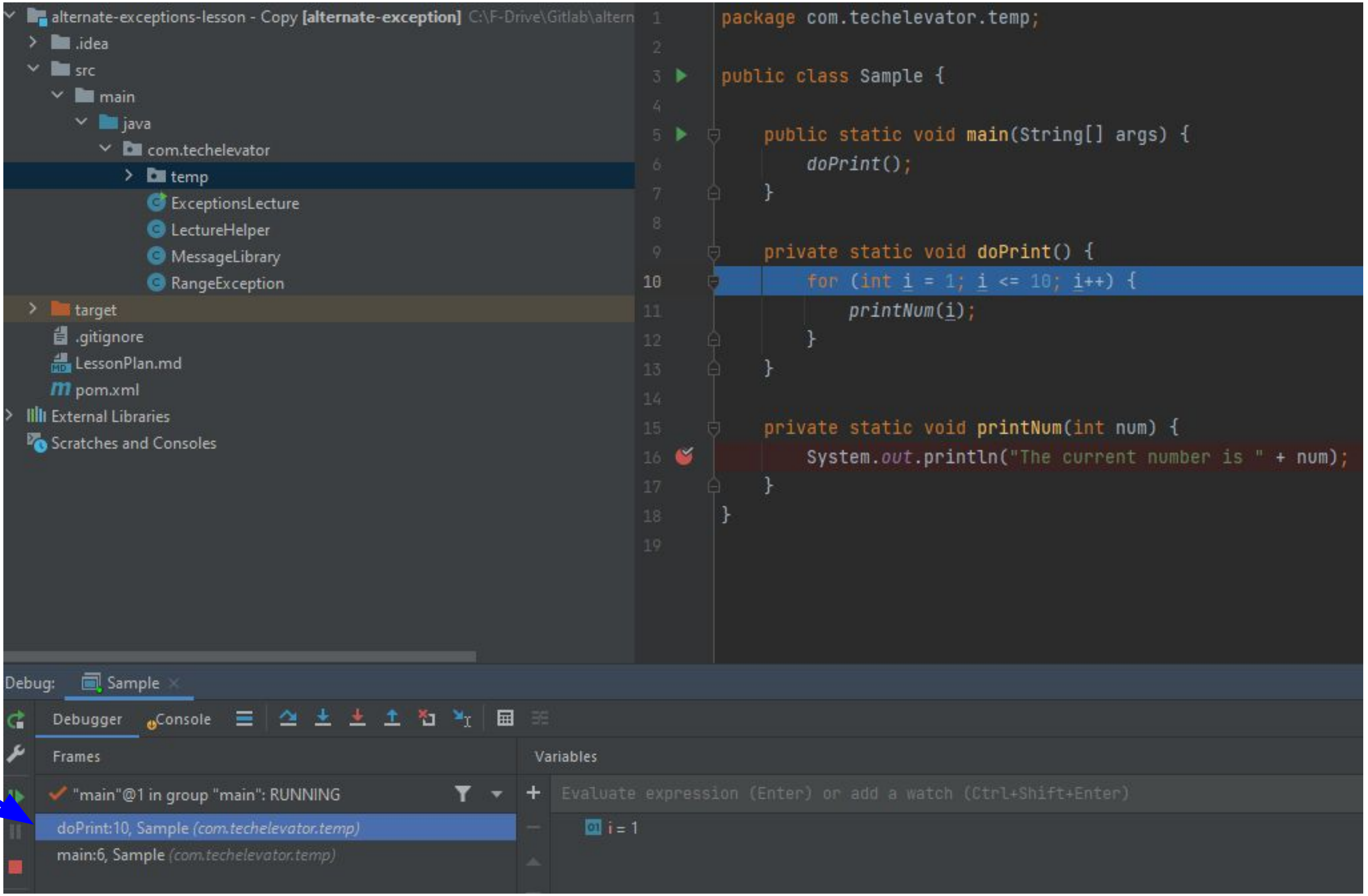
```java
package com.techelevator.temp;

public class Sample {

    public static void main(String[] args) {
        doPrint();
    }

    private static void doPrint() {
        for (int i = 1; i <= 10; i++) {
            printNum(i);
        }
    }

    private static void printNum(int num) {   num: 1
        System.out.println("The current number is " + num);   num: 1
    }
}
```

```
"main"@1 in group "main": RUNNING
printNum:16, Sample (com.techelevator.temp)
doPrint:11, Sample (com.techelevator.temp)
main:6, Sample (com.techelevator.temp)
```

```
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
num = 1
```

When **printNum** returns to **doPrint**, **printNum** is popped off the call stack.

The call stack now looks like this.



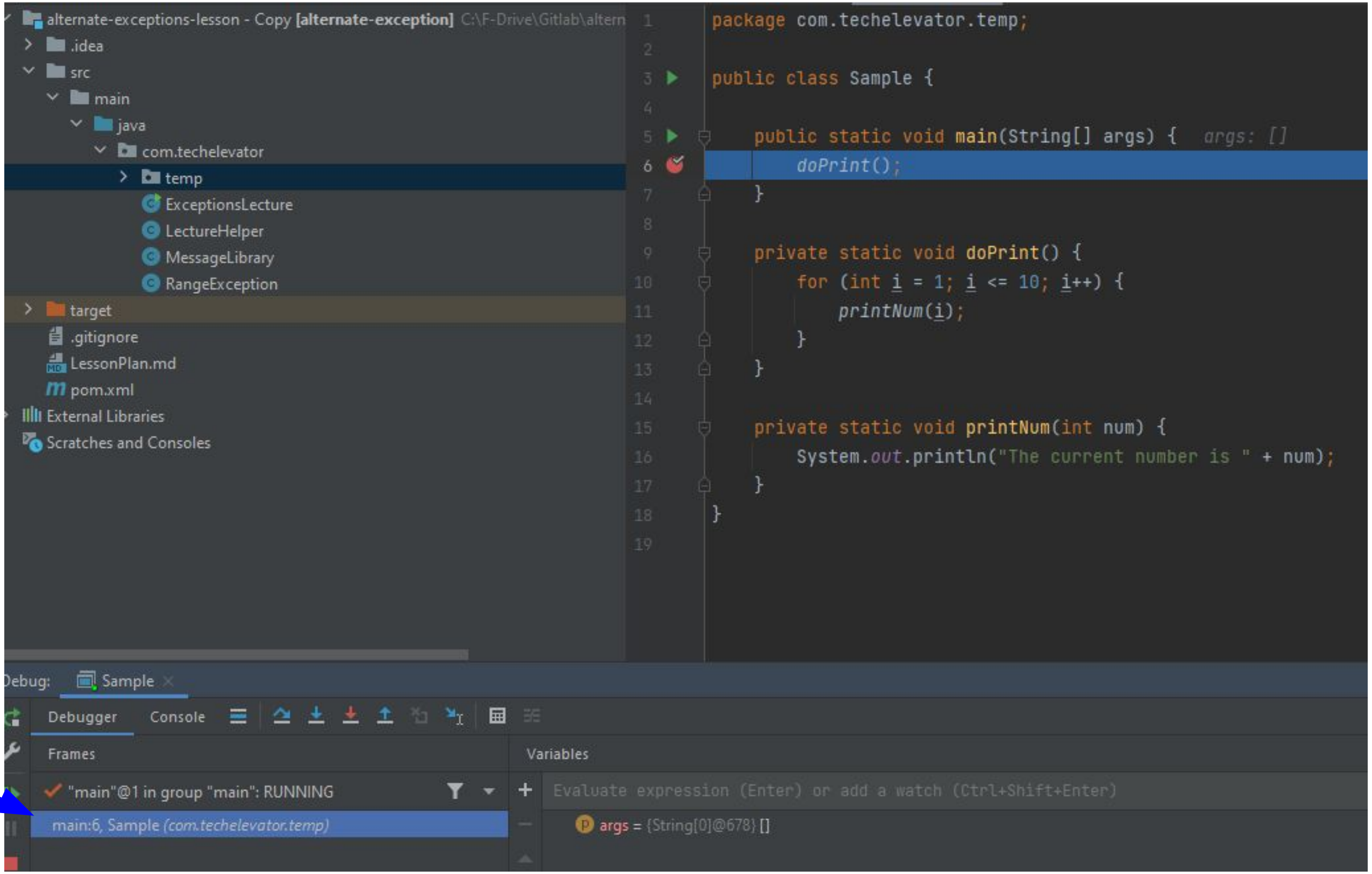
```
package com.techelevator.temp;

public class Sample {

    public static void main(String[] args) {
        doPrint();
    }

    private static void doPrint() {
        for (int i = 1; i <= 10; i++) {
            printNum(i);
        }
    }

    private static void printNum(int num) {
        System.out.println("The current number is " + num);
    }
}
```

```
"main"@1 in group "main": RUNNING
doPrint:10, Sample (com.techelevator.temp)
main:6, Sample (com.techelevator.temp)
```

```
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
i = 1
```

Finally, when **doPrint** returns to **main**, **doPrint** is popped off the call stack.

The call stack now looks like this.



```java
package com.techelevator.temp;

public class Sample {

    public static void main(String[] args) {   args: []
        doPrint();
    }

    private static void doPrint() {
        for (int i = 1; i <= 10; i++) {
            printNum(i);
        }
    }

    private static void printNum(int num) {
        System.out.println("The current number is " + num);
    }
}
```

```
"main"@1 in group "main": RUNNING
main:6, Sample (com.techelevator.temp)
```

```
args = {String[0]@678} []
```

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.

- *Still other times, the developer uses exceptions to communicate anticipated possible errors that might occur in their code:* i.e. not having enough money in an account to make a withdrawal.
  .
- Java uses classes inherited from the **`Exception`** class to handle exception situations in code.

Using Exceptions to Communicate Possible Error Conditions in Code

Why Use Exceptions To Communicate Error Conditions?

Let's look at this scenario without using exceptions:

- We have a method that counts from **`start`** to **`end`**:

```java
public String count(int start, int end) {
    String result = "";
    for (int num = start; num <= end; num++) {
      result += num;
      if (num < end) {
        result += ", ";
      }
    }
    return result;
}
```

- What happens when **`end`** is less than **`start`**?

Introducing Try/Catch

We wrap the code we are attempting in a **`try`** block.

If no exception is thrown, the code will execute completely and move past the **`try/catch`** block.

```
try {
    String countString = helper.count(start, end);
    System.out.println(String.format("CountString: %s",
        countString));
} catch (RangeException ex) {
    System.out.println(ex.getMessage());
}
```

If our error condition occurs and the **`count`** method throws a **`RangeException`**, the code will **stop executing** at that point (the line that prints the results will **NOT** be executed) and jump to the code in the **`catch`** block, which handles the **`RangeException`**.

The **`catch`** block receives a **`RangeException`** object that we are capturing in the **`ex`** variable here.

We can call **`getMessage`**() on the **`ex`** variable to get the message contained in the **`RangeException`**.

Exception Handling Rules

When calling a method that throws an exception, you have two options:

- Handle the exception using a **`try/catch`** block.
- "Pass the buck" to the calling method by declaring that YOUR method throws the exception.
  - The calling method will not have the same choice to make.

Runtime Exceptions

Checked Exceptions vs. Runtime Exceptions

- Exceptions that follow the rules we just discussed are called **checked exceptions**.

- Not all exceptions can be predicted.

- There is a category of exception that behaves a bit differently to handle unexpected errors: **runtime exceptions**.

Runtime Exceptions

- `RuntimeExceptions` **CAN** be handled but **AREN'T required to be**.

- `RuntimeExceptions` can be explicitly thrown but can also be thrown by the JVM as a result of something unexpected happening. Some examples of unexpected `RuntimeExceptions`:
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`

Exception Polymorphism

Exception Polymorphism

- Exceptions are classes and can have member variables, methods, and multiple constructors like any other class.

- All exceptions (including **`RuntimeException`**) are inherited from the **`Exception`** class.

- Polymorphism applies to **`Exceptions`** like any other class. So a **`RangeException`** "is a(n)" **`Exception`**.

```
public class RangeException extends Exception {

      // other code

}
```

CATCH BLOCK HIERARCHY

- You can have multiple **`catch`** blocks.
- Once the code falls into a specific **`catch`** block, none of the other **`catch`** blocks will execute.
- If the code to handle multiple **`Exceptions`** is the same, you can combine them into one **`catch`** block.

```
try {
      // some code
} catch (RangeException | IllegalArgumentException ex) {
      System.out.println(ex.getMessage());
}
```

A Closer Look at the Try-Catch-Finally Block

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

```
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!");
}
```

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.

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
- Through a web interface using the Spring framework
- Through an external API

File Input : The File Class

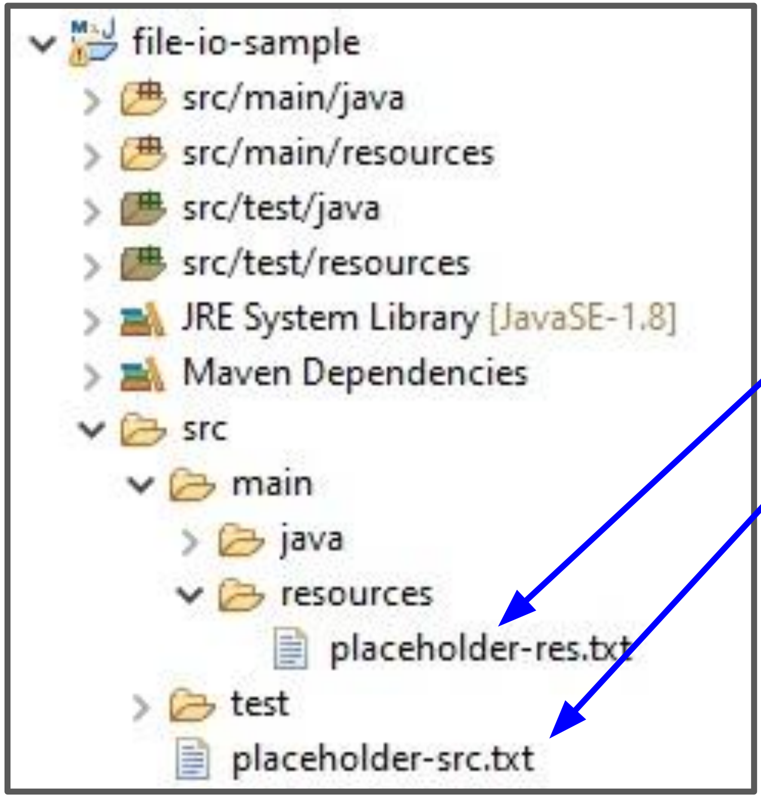
The file class is the Java class that encapsulates what it means to be a file containing data. This is an instantiation of a **`File`** object.

```
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.

File Input : The File Class



```
file-io-sample
  src/main/java
  src/main/resources
  src/test/java
  src/test/resources
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
    main
      java
      resources
        placeholder-res.txt
    test
    placeholder-src.txt
```

The location in the File constructor is relative to the root of the Java project.

```java
File resourcesfile =
        new File("src/main/resources/placeholder-res.txt");

File srcFile =
        new File("src/placeholder-src.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 indicating whether a file exists. We would not want to proceed to parse a file if the file itself was missing!

- **`.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.

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 and Scanner

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

- Once a **`File`** object exists, we can instantiate a **`Scanner`** object with the **`File`** as a constructor argument. Previously, we used System.in as the argument to indicate we were reading from the keyboard.

- Before we look at that though… let's review how we have been creating **`Scanner`** objects...

Review of Scanner

Let's review how we created a **`Scanner`** object previously:

```
    Scanner input = new Scanner(System.in);

    System.out.print("Type something: ");
    String data = input.nextLine();

    System.out.println(data);
```

We created a new **`Scanner`** object and passed the keyboard input stream in the constructor.

There is an issue with this code. When you open a **`Scanner`**, or any other resource that can be opened and closed, you should close it when you are done.

Not doing so can cause all kinds of issues (one example is that if you are using a **`File`** that is buffering data, the data may never get "flushed" and written to the filesystem if you don't close the **`File`**).

Review of Scanner

Let's review how we created a **`Scanner`** object previously:

```
Scanner input = new Scanner(System.in);

System.out.print("Type something: ");
String data = input.nextLine();

System.out.println(data);
```

We created a new **`Scanner`** object and passed the keyboard input stream in the constructor.

There is an issue with this code. When you open a **`Scanner`**, or any other resource that can be opened and closed, you should close it when you are done.

Not doing so can cause all kinds of issues (one example is that if you are using a **`File`** that is buffering data, the data may never get "flushed" and written to the filesystem if you don't close the **`File`**).

But wait…. what happens if we call **`input.close()`** ? If you do this you will find your program **can no longer read from the keyboard!!!** Why? Because you closed the input stream associated with keyboard when you closed the **`Scanner`**!!! **`System.in`** is a special stream that **shouldn't be closed** (it will be closed when your program exits), but other inputs used with **`Scanner`** should always be closed.

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...

```java
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 **<u>try-with-resources</u>** 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.");
          }
     }
}
```

Exceptions When Reading Streams

Exceptions can often occur when reading streams. Here are some example of common scenarios when I/O Exceptions might occur:

- Directory not found
- End of stream reached
- File not found
- Path too long (Windows only)

What is File I/O Used for In the Real World?

Here are just a few examples of when you might read or write files in you future career:

- Importing Bulk Data Sets
- Desktop Applications - Reading in Configuration Settings
- Video Games - Data File
- Transmitting data to other systems