TECH ELEVATOR

Module 1, Week 8

- What is method overloading and how do you implement it?
- What are some of the similarities and differences between abstract classes and interfaces?
- What does it mean for a method to be static?

HouseKeeping

- File I/O reading
- LMS will be down Tuesday, October 10th between 3:30 – 5:30pm CST
- Cross Cohort Happy Hour on Thursday, October 19th
- When to reach out for help
- Whiteboarding sessions
- All Quizzes must be completed to move onto the next module.
- I'll be out on Wednesday next week

Week 8 Agenda

- File I/O reading
- File I/O writing
- Handy tips
- Quick run-through of end-of-module project
- Info on the end of module assessment
- Clone and work on pair programming exercise.

File

- Java has the ability to read and write data stored in a file.
- Here are a few examples of when you might read or write files:
  - Importing Bulk Data Sets
  - Desktop Applications - Reading in Configuration Settings
  - Video Games - Data File
  - Transmitting data to other systems
- The `java.io.File` class is a Java class that represents and can be used to perform actions on a file or directory in the filesystem.

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

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

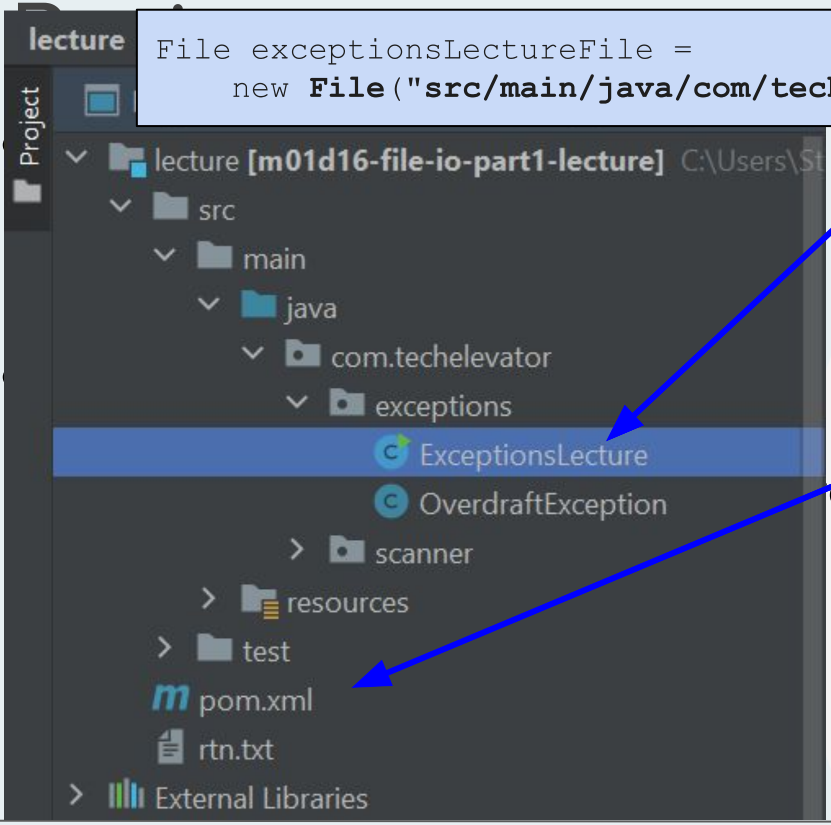
Note that this does not create a new file. It creates a new `File` object instance in memory. You can then call methods like `exists()`, `isFile()` on the instance. You can also invoke the `createNewFile()` method to create an actual file in the file system.

File

If the pathname passed to the File constructor is relative, then it should be relative to the root of the Java project.

```
File exceptionsLectureFile =
    new File("src/main/java/com/techelevator/exceptions/ExceptionsLecture.java");
```

```
File returnText= new File("rtn.txt");
```



eek 8)

File

There are several methods of the file class that are useful for file input:

- `exists`:
    - Returns a `boolean` indicating whether a file exists on the filesystem. We would not want to proceed to parse a file if the file itself was missing!
- `isFile`:
    - Returns a `boolean` indicating if what we are looking at is a File. Returns false if it is not a file (perhaps a directory).
- `getAbsoluteFile`:
    - Returns the same `File` object you instantiated but with an absolute path.

```
public static void main(String[] args)throws FileNotFoundException
    File inputFile = new File("rtn.txt");
    if (inputFile.exists()) {
        // do file related code here…
    }
}
```

Scanner

- Remember how we used the `Scanner` class to read keyboard input from the `System.in` standard input stream.

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

- We used `scanner.nextLine()` to get text from the input stream up until a newline.

```
String userInput = scanner.nextLine();
```

- We can also use a `Scanner` object with a `File` to read file data.

```
File inputFile = new File("rtn.txt");
Scanner scanner = new Scanner(inputFile);
String lineFromFile = scanner.nextLine();
```

File Input

Examples:

- Tutorial
- VEE app - TextFileStorage.java

File Output

- Java supports writing data to a text file
- Uses a buffer - what's that, and why?
  - A buffer holds data until it reaches a certain size, at which point it is emptied. This helps improve performance since writing to disk is relatively slow.
- What are other ways to communicate data back to the user?
  - System.out to write a message to the console
  - Write to a database (module 2)
  - Send data to an API (module 2)
  - Send HTML back to the user (module 3)
- Java OutputStream

- 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.

We need to throw the checked exception.

Instantiate a new **`File`** object

Instantiate a new **`Scanner`** object with the file.

Loop until we've processed all the lines of the file.

```
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);
            }
        }
    }
}
```

Notice the unusual **`try`** block. More to come…

Try With Resources

- The `try-with-resources` statement is a try statement that declares one or more resources.
- A resource is an object that must be closed after the program is finished with it.
- The `try-with-resources` statement ensures that each resource is closed at the end of the statement.
- Any object that implements `java.io.Closeable`, can be used as a resource.
- Since the `System.in` object is opened by the JVM, you should leave it to the JVM to close it. If you close it and later on try to use `System.in`, you'll probably be surprised to find that you no longer can. Don't use `System.in` with the `try-with-resources` statement!

The **`Scanner`** is opened as a resource and will automatically be closed, even if an exception is thrown from the try block.

```
public void displayFile(String filename)throws FileNotFoundException {
     try (Scanner scanner = new Scanner(new File(filename))) {
          while(scanner.hasNextLine()) {
               System.out.println(scanner.nextLine());
          }
     }
}
```

FileNotFoundException

- This is an example of handling the possibility of a `FileNotFoundException` when opening a `File` rather than having the method "pass the buck".
- 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 method does not have to re-throw the exception.

```
public void displayFile(String filename)  {
    try (Scanner scanner = new Scanner(new File(filename))) {
        while(scanner.hasNextLine()) {
            System.out.println(scanner.nextLine());
        }
    } catch (FileNotFoundException e) {
        System.out.println("File not found: " + e.getMessage());
    }
}
```

More File Methods

- The `File` class has several methods which can be used to find extra information about the specified path.
- You have already seen `exists`, `isFile` and `getAbsolutePath` but here are a few more of the available methods:
    - `getName` - returns name of the File (just the name, not any path info)
    - `isDirectory` - indicates whether the path points to a directory
    - `length` - size of file in bytes

Create 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.

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

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

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

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

Note that the directory will be created at the root level of the project.

Create File

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

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

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

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

Catch and handle the **`IOException`** if it is thrown.

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

Create File

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

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

Again, note that this does not create a new file. It creates a new `File` instance in memory containing the file path. You can then call methods like `createNewFile()` to create an actual file in the file system.

Extra parameter 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.

File Write

Text to write to file.

Create a **`PrintWriter`** with the **`File`** object.

"prints" the text in **message** to the **`PrintWriter`**.

```
File newFile = new File("myDataFile.txt");
String message = "Appreciate, Elevate, Participate";

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

What is **`flush`**? Stay tuned....

Close the **`Printwriter`**.

Catch the possible **`FileNotFoundException`**.

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)) {
     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

- The previous example overwrites the file's contents every time it is run. Sometimes, a file might need to be appended to, preserving the existing data content. The `PrintWriter` supports two constructors:
  - `PrintWriter(file)`, where file is a `File` object.
  - `PrinterWriter(new FileOutputStream(file, mode))`
    - The parameter needs to be an instance of the `OutputStream` class.
    - The `mode` parameter used to construct the `OutputStream` instance is a boolean indicating whether or not you want to create the object in append mode.

File Append

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 = "\nAppreciate\nElevate\nParticipate";

boolean appending = newFile.exists() ? true : false;
try (PrintWriter writer = new PrintWriter(new FileOutputStream(newFile, appending)))
{
     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 the **`File`** object and the **boolean** we created as the param which indicates whether or not to append.

Try Multiple Resources

We can use `try-with-resources` to manage multiple resources at once.

```
File sourceFile = new File("myDataFile.txt");
File destinationFile = new File("myOutputFile.txt");

try (
  Scanner sourceScanner = new Scanner(sourceFile);
  PrintWriter destinationWriter = new PrintWriter(destinationFile)){

    while (sourceScanner.hasNextLine()) {
        String line = sourceScanner.nextLine();
        destinationWriter.println(line);
     }
}
```

`try-with-resources` declares multiple resources, each assigned to a variable and delimited by a semicolon

File Output

Examples:

- VEE app - TextFileStorage.java
- Tutorial

Testing File I/O

- Types of testing (LMS)
- What might be some obstacles you'd encounter with **integration testing**?
  - Can be harder to write (require more code == developer effort)
  - Usually take much longer to run
  - Can be flaky since they interact with real servers/dependencies
- What might be some obstacles you'd encounter with **testing file I/O**?
  - Setting up test files to read from
  - Creating test files that are written that don't stick around

Testing File I/O

Example - VEE app, TextFileStorageTest.java

Old vs. New Java Libraries

Java tutorial

Week 9 - End-of-module project

- The format will be identical to the format of week 4
- Late in the week, I'll ask for 3-4 volunteers to share their code during our class discussion for a code review
- Pair programming: code reviews in groups

Week 10 - End-of-module assessment

Saturday, October 21st (3 hours)

- Exam + coding portion
- Practice assessment is available after next week's class
  - Take it early to help guide your review efforts
  - Take it as many times as you'd like
- Day of
  - You can use your notes and the LMS
  - Breakout rooms
  - Manually graded
- No pair programming in the afternoon

You'll be great!