GCIS-124
Software Development & Problem Solving

1.3: File I/O and Exceptions

RIT | Golisano College of Computing and Information Sciences

This Week: Python to Java

Unit 1: Python to Java				
Procedural Programming		Unit Testing with JUnit 5		File I/O and Exceptions

Python to Java

This unit will mostly focus on showing how to use Java to do things that you already know how to do in Python. This mostly involves learning ***Java syntax***.

As we move further into the semester, we will more deeply explore ***object-oriented programming***, and advanced topics such as ***threading*** and ***networking***.

- In GCIS-123 we wrote software exclusively in the Python programming language.
  - For most of the course we used ***procedural programming***, which is a term for programs that use ***functions*** to implement most of the program requirements.
  - Towards the end of the course, we introduced ***object-oriented programming***.
- In GCIS-124 we will be using the ***Java Programming Language***, a fully object-oriented language.
  - Unlike Python, in Java ***all*** code must be inside of a class.
- During this unit we will focus on learning how to use Java to implement many of the programs that you already know how to write in Python.
  - Types, & Variables ✅
  - Methods, Parameters, Arguments, & Return Values ✅
  - Boolean Expressions, Conditionals, & Loops ✅
  - Unit Testing with JUnit ✅
  - Exceptions & File I/O ❌
- Today we will focus on ***exception handling*** and ***file I/O*** with text files.

Review: All the Java

```
1  public class Summer {
2
3     public static int natSum(int n) {
4        if(n <= 0) {
5           return 0;
6        } else {
7           int sum = 0;
8           while(n > 0) {
9              sum += n;
10             n = n - 1;
11          }
12          return sum;
13       }
14    }
15
16    public static void main(String[] args) {
17       int total = natSum(100);
18       System.out.println("Sum 1-100: "
19          + total);
20    }
21 }
```

Not every Java class needs to be executable, but an ***executable*** Java class must have a main method with a specific signature.

- ***All*** code in Java must be within the body of a class.
  - ***Whitespace*** in Java is insignificant.
  - ***Statements*** are terminated with a semicolon.
  - ***Curly braces*** (`{}`) are used to define ***blocks of code***.
- Java is ***statically typed***, meaning that ***all*** variables must be declared with a ***type*** and a ***name***.
  - Java includes ***8 primitive types*** including ***integers***, ***floating point values***, ***characters***, and ***booleans***. All other types are ***reference types***, including strings.
  - Only values of a ***compatible type*** can be assigned to a variable, e.g. `int x = 5;`
- Java Strings ***must*** be enclosed in ***double-quotes*** (`""`), and the + operator can be used to concatenate ***any type*** onto a string, e.g.
  - `String s = "abc" + 123 + false;`
- Java ***method signatures*** have several parts:
  - An optional ***access modifier*** like public determines visibility of the methout outside of the class.
  - A static modifier indicates that the method belongs to the class and can be called without an object.
  - A method ***must*** return a value of its declared ***return type*** unless it is void.
  - A valid method name.
  - ***Zero or more parameters*** including types and names.
- Java supports `while` loops and "classic" `for` loops.
- `System.out.println()` is used to print to ***standard output***.

1.3.2

Reverse a String

```
int length = string.length();
char c = string.charAt(3);
String cat = "abc" + "def";

for(int i=5; i<11; i++) {
   System.out.println(i);
}
```

Some code examples that you will find useful when solving this problem.

Let's warm up by practicing some of the Java basics that you learned earlier in this unit, including writing methods, returning values, and looping over strings; write a method that returns a reversed copy of a string.

- Create a new Java class in a file named "`Miscellany.java`" and define a method named "`reverseChars`" that declares a `String` parameter.
  - Use a loop to create a copy of the string with the characters reversed.
  - Return the reversed copy.

Java Arrays

Arrays are allocated as a ***contiguous*** block of memory that can be efficiently accessed using an ***index*** that ranges from ***0*** to ***length-1***.

An array can be visualized as a ***table*** with a single row. The number of columns is determined by the length of the array.

Arrays are created to be a specific size using the `new` keyword, which tells Java to ***allocate memory*** and ***store*** something there.

In the case of an array, the memory allocated is a contiguous block large enough to hold the number of elements specified when the array is created.

- A ***data structure*** is a grouping of related elements.
  - In the previous course, we worked with lots of different data structures including lists, sets, dictionaries, stacks, and queues.
- You should recall that an ***array*** is the most basic kind of data structure, and it has the following properties.
  - Arrays are ***fixed length***; arrays are created to be a ***non-negative size***, and the size never changes.
  - Arrays can store elements of ***any type***.
  - Individual elements are accessed using an ***index*** that ranges from ***0*** to ***length-1***.
- A Java array is declared using ***any type*** with square brackets, e.g.
  - `String[] strings;`
  - `int[] integers;`
- An array is ***initialized*** using the new keyword and the size of the array in square brackets, e.g.
  - `integers = new int[10];`
  - The ***length*** field can be used to get the length of the array, e.g. `integers.length`
- Java fills each array with ***default values*** appropriate for the type.
  - ***0*** for ***numeric types*** (including `char`).
  - `false` for ***booleans***.
  - `null` for ***reference types***.

1.3.4

```java
int[] numbers = new int[5];
numbers[0] = 2;
numbers[1] = 3;
numbers[2] = 5;
numbers[3] = 7;
numbers[4] = 11;

for(int i=0; i<numbers.length; i++) {
  System.out.println(numbers[i]);
}
```

Creating an array works a little differently in Java, but using one is very similar to the `arrays` module in Python.

Array of Squares

Implement a method that returns an array of squares.

- Open your `Miscellany` class and define a method named "`squares`" that declares a parameter for an integer `n`.
  - Create an ***integer array*** large enough to hold `n` elements.
  - Use a loop to set the value at each index to the ***square of the index***
    - i.e. 0^2, 1^2, 2^2, ..., $n\text{-}1^2$.
  - ***Return*** the array.
- Call your `squares` method from main and print the results.
  - Use the static `Arrays.toString()` method to print your array to standard output.
- ***Compile*** and ***run*** your class from the VS Code terminal.

2-Dimensional Arrays

You can depict a ***two-dimensional array*** with 4 rows and 6 columns as a table...

	0	1	2	3	4	5
0	5	3	7	0	0	0
1	13	16	0	3	4	17
2	2	10	5	6	5	14
3	1	18	11	4	3	12

The ***table*** is really an ***array of arrays***. Each ***row*** is an ***array of values***, e.g. row 0 is the array containing the values `[5, 3, 7, 0, 0, 0]`.

Let's take a closer look...

- In Python we experimented with two-dimensional lists. In Java, it is possible to create ***two-dimensional arrays***.
  - A 2D array is ***an array of arrays***.
  - Like any other array, a two-dimensional array ***must*** be declared with a ***type***, and can only be used to store values of that type.
  - Two-dimensional arrays are declared using two sets of square brackets, e.g. `int[][] table;`
  - 2D arrays are initialized with ***two sizes***, which you can think of as the number of ***rows*** and ***columns*** in the array, e.g. `table = new int[4][6];`
- The values in a two dimensional array are accessed using two indexes.
  - You can think of these as the index of the ***row*** and ***column*** of the data that you are looking for.
  - For example: `int value = table[4][5]` will retrieve the value from ***row 4***, ***column 5*** in the 2D array.
- You can also fetch the array for an entire row all at once by using only one index.
  - For example: `int[] row = table[3]` will grab the array that contains the values in the ***row 3***.

1.3.4

```
// 5 rows, 4 columns
int[][] table = new int[5][4];

// row 1
int[] entireRow = table[1];
// the value at row 3 column 2
int individualValue = table[3][2];

for(int row=0; row<table.length; row++) {
  for(int col=0; col < table[row].length; col++) {
    System.out.println(table[row][col]);
  }
}
```

In a 2D array a single index is used to get an entire row, and two indexes are needed to retrieve an individual value from a row.

Multiplication Tables

Write a method that, given a number of rows and columns, returns a multiplication table.

- Open your `Miscellany` class and define a new method named "`multiplicationTable`" that declares a parameter for `rows` and `columns`.
  - Create ***a two-dimensional array*** of the specified size.
  - Use loops to set the value at each index to the ***product of its row and column***, starting at 1. Do not include 0s!
- Call your method from `main`.
  - Use a loop to print each row in the table using the `static` `Arrays.toString()` method, e.g. `Arrays.toString(table[0])` will print the first row.
- ***Compile*** and ***run*** your class from the VS Code terminal.

1.3.5

Invalid Input

The latest and greatest version of the Calculon program that you wrote should prompt the user to enter two integers and use them to test the five functions of your calculator. What happens if you enter non-numeric input when prompted to enter an integer? Run your `Calculon` class and find out!

- If necessary, ***compile*** and ***run*** the `Calculon` class that you wrote earlier in this unit.
  - You should be prompted to enter two floating point operands.
  - What happens if you enter ***non-numeric input***?

Java Exceptions

Code that may ***throw*** an exception will crash your program if the exception is not ***handled***.

The code may be enclosed in the body of a `try` ***block***. The try must be followed by a `catch` that specifies the type of exception it handles.

```
1  Scanner scanner = new Scanner(System.in);
2  try {
3    System.out.print("Enter a number: ");
4    int x = scanner.nextInt();
5  } catch(InputMismatchException e) {
6    System.out.println("Invalid integer!");
7  }
```

The code in the `catch` ***block*** is executed iff an exception of a matching type is thrown in the `try` block.

- In Python, we saw lots of different kinds of ***errors***, e.g.
  - Trying to convert a non-numeric string into an integer raises a `ValueError`.
  - Trying to index into an integer raises a `TypeError`.
  - Trying to access an invalid index in a list raises an `IndexError`.
  - Trying to open a file that doesn't exist raises a `FileNotFoundError`.
- In Java these types of errors are called ***exceptions***.
  - Specifically, trying to read non-numeric input from the user as an integer causes an `InputMismatchException`.
  - You may have seen other exceptions before, including `IndexOutOfBoundsException` and `NullPointerException`.
- Java code that causes an exception is said to ***throw*** the exception.
- As with Python, if nothing is done to handle the error, your program will ***crash***.
  - In Python, we used `try`/`except` to handle errors that were raised.
  - In Java, `try`/`catch` works almost exactly the same way. One notable difference is that the catch ***must*** specify the ***type*** of exception that it handles.

1.3.6

try/catch

```java
Scanner scanner = new Scanner(System.in);
try {
  System.out.print("Enter a number: ");
  int x = scanner.nextInt();
} catch(InputMismatchException e) {
  System.out.println("Invalid integer!");
}
```

Handling errors in Java works a lot like it does in Python. Instead of using try/except, we will use try/catch to handle exceptions that may occur. Try it out by modifying your Calculon class so that it displays an error if the user enters invalid input to either prompt.

- Open your `Calculon` class and navigate to the `main` method.
  - Use a `try`/`catch` to handle the exception that occurs if the user types invalid input.
  - Print an error message and exit gracefully.
- ***Compile*** and ***run*** your code from the VS Code terminal.

- The `java.io.File` class represents a ***file*** or ***directory*** in the computer's file system.
  - ***Importing*** the class will allow your program to use it without specifying the full class name.
- A ***file object*** is created by calling the `File` class constructor with the path to the file.
  - An ***absolute path*** may be used.
  - A ***relative path*** may also be used; such paths are relative to the directory from which the program is executed.
  - e.g. `File f = new File("a_file.txt");`
- An instance of the `File` class cannot be used to read from or write to the specified file, but does provide many methods that can be used to get information about the file.
  - `exists()` - returns `true` if a file with the specified path exists in the file system, and `false` otherwise.
  - `isDirectory()` - returns `true` if the file is a directory, and `false` otherwise.
  - `getAbsolutePath()` - returns the absolute path to the file as a `String`.
  - `length()` - returns the number of bytes of data in the file as a `long`.

Java Files

An ***instance*** of the `java.io.File` class can be created with an ***absolute*** or ***relative*** path to a file in the file system.

A ***relative path*** is relative to the directory in which the program was executed. For example, `"data/file.txt"` refers to a file named `"file.txt"` in the `"data"` subdirectory.

```
1  File file = new File("a_file.txt");
2  String path = file.getAbsolutePath();
3  long size = file.length();
4  boolean dir = file.isDirectory();
```

Once created, a file object can be used to get information about the file including its ***absolute path***, ***length***, and whether or not it is a ***directory***.

1.3.7

File Info

A `File` object is created with string specifying a filename or a path to a file.

```
File file = new
File("some_file.txt");
String name = file.getName();
boolean exists = file.exists();
```

Once you have created a file object, you can call methods on it to get information about the file.

Try using VS Code's autocomplete feature to see which methods are available.

In Java, `File` objects can be used to get lots of different information about files and directories. Try it out now by implementing a method that uses a `File` object to print detailed information about any file.

- Create a new Java class in a file named "`Files.java`" and define a method named "`info`" that declares a parameter for a `filename`.
  - Create a `File` using the `filename` and print the following:
    - The ***name*** of the file.
    - The ***absolute path*** to the file.
    - Whether or not the file ***exists***.
    - If the file exists, print its ***length*** in bytes.
- Call your function from `main` with several different filenames.
  - ***Hint***: use the names of files in your repository.
- ***Compile*** and ***run*** your new class from the VS Code terminal.

- A `java.io.FileReader` can be used to read ***character data*** (***text***) from a file.
  - A `FileReader` can be created by passing an ***absolute*** or ***relative path*** into its ***constructor***.
  - e.g. `new FileReader("a_file.txt")`
- Once created, a `FileReader` provides several methods for reading text.
  - `read()` - returns the next character of data.
  - `read(char[] buffer)` - reads up to `buffer.length` characters into the given buffer.
- A `FileReader` is a little hard to use because it only supports reading characters (one at a time or in chunks). A `java.io.BufferedReader` can be constructed with a `FileReader` and provides a `readLine()` method that reads up to the next newline from the file and returns it as a `String`.
  - e.g. `String s = reader.readLine();`
  - The method ***returns*** `null` when the end of the file has been reached.
- The `FileReader` and/or `BufferedReader` should be ***closed*** when no longer needed.

Reading Text Files

A `java.io.FileReader` can be used to read ***characters*** (one at a time or in chunks) from the file with the specified ***path***.

```
1  FileReader fileReader =
2     new FileReader("a_file.txt");
3  BufferedReader reader =
4     new BufferedReader(fileReader);
5  String line = reader.readLine();
6  reader.close();
7  fileReader.close();
```

A `java.io.BufferedReader` can be created with a `FileReader`. Its `readLine()` method can be used to read ***one line of text*** at a time.

Opened files should always be ***closed*** when no longer needed to avoid locking the file (and preventing other processes from using it).

1.3.8

Reading Text Files

Unfortunately, reading text files in Java is a lot more complicated than it is in Python. There is a lot more setup *and* you can't just iterate through the lines in the file exactly. Let's practice a little now.

```
FileReader fileReader =
   new FileReader("a_file.txt");
BufferedReader reader =
   new
BufferedReader(fileReader);
String line = reader.readLine();
reader.close();
fileReader.close();
```

- Open your `Files` class and define a method named "`printFile`" that declares a parameter for a `filename`.
  - Create a `FileReader` using the `filename`.
  - Create a `BufferedReader` using the `FileReader`.
  - Use a loop to print the lines in the file one at a time.
    - Use the `readLine()` method on the `BufferedReader`.
    - If `readLine()` returns `null`, you have reached the ***end of the file***.
    - Don't forget to close the `FileReader` and `BufferedReader` when you are done!
- Call your method from `main` using one of the provided files in the data directory in your repository.
  - e.g. `"data/alice.txt"`
- What do you notice in VS Code?
- What happens when you try to compile and run your code?

Checked Exceptions

Because `IOException` is a ***checked exception***, it must be explicitly handled using a `try/catch` or by ***rethrowing*** the exception.

Many of the methods on the various I/O classes throw `IOException` (including the ***constructors***).

```
1  public static String readOneLine()
1                       throws IOException {
1      FileReader file =
1          new FileReader("a_file.txt");
1      BufferedReader reader =
1          new BufferedReader(file);
1      String line = reader.readLine();
1      reader.close();
1      return line;
1  }
```

A `throws` ***declaration*** is added to the method signature. In the event that the specified type of exception occurs, it is automatically ***rethrown***.

- Up until this point, we have had to deal with ***unchecked exceptions*** like `InputMismatchException`.
  - You are ***not*** explicitly required to handle an unchecked exception in any way.
  - Of course, in the event that an unchecked exception is thrown, it will crash your program if it is not handled.
- Java also includes ***checked exceptions***.
  - If a program calls a method that throws a checked exception, the programmer ***must*** handle the exception in some way.
  - Failure to handle a checked exception will cause a ***compiler error***.
  - In a way, this can be very good - checked exceptions do not "sneak through" and crash your program like unchecked exceptions can sometimes do.
- A checked exception can be handled in two ways.
  - Using a `try/catch` in the same way as we did previously with unchecked exceptions.
  - ***Rethrowing*** the exception by adding a `throws` ***declaration*** to the method, e.g.
    - `void method() throws IOException`
- Nearly every method used to read data from files can throw a `java.io.IOException`.
  - `IOException` is ***checked***, and so ***must*** be handled.

1.3.9

One way to handle a checked exception is to ***rethrow it***.

```
public static void doSomething()
  throws AnException {
  int x = aMethod();
}
```

Another is to `try`/`catch` it.

```
public static void doSomething() {
  try {
    doSomething();
  } catch(RuntimeException e) {
    doSomethingElse();
  }
}
```

Checked Exceptions

Nearly every Java I/O operation may throw an `IOException` if something goes wrong. `IOException` is checked, and so must be handled in some way. Update your `Files` class to better handle checked exceptions if and when they occur.

- Open your `Files` class and navigate to the `printFile` method.
  - Update the method signature to declare that it `throws IOException`.
- Use a `try`/`catch` in your `main` method to handle the exception.
  - If an `IOException` occurs, print an error message and exit.
- ***Compile*** and ***run*** your code from the VS Code terminal.

- The `java.io.FileWriter` class provides methods for writing ***character data*** to a file.
  - `write(char c)` - writes a single character.
  - `write(char[] buffer)` - writes an entire array of characters all at once.
  - `flush()` - forces any data that has been ***buffered in memory*** to be written to the file. If you do not ***flush*** before closing the file, your data may not be written!
- A `FileWriter` is a little hard to use because it only supports writing characters. The `java.io.PrintWriter` class can be constructed ***with*** a `FileWriter` and provides methods that are easier to use. Anything ***printed*** to the `PrintWriter` is ***written*** out to the `FileWriter`.
  - `print(String s)` - writes the string to the FileWriter.
  - `println(String s)` - writes the string followed by a newline.
  - `print(int i)` - prints the integer as a string, e.g. `"123"`. There is a similar methods for each primitive type.
  - `flush()` - flushes any buffered data out to the file. Again, data that is not flushed may not be written!

Writing Text Files

A `FileWriter` can be created with a ***filename*** to write ***character data*** out to the file, but a `FileWriter` can be a little ***clunky*** to use.

A `PrintWriter` can be created with a `FileWriter`.

```
1  FileWriter fw = new FileWriter("a_file.txt");
2  PrintWriter writer = new PrintWriter(fw);
3  writer.print("Your age is: ");
4  writer.print(18);
5  writer.println(" years old.");
6  writer.flush();
7  writer.close();
```

A `PrintWriter` provides many ***convenience methods*** for printing data. Anything ***printed*** using the `PrintWriter` is ***written*** to the `FileWriter` as character data.

Data should be ***flushed*** and the PrintWriter ***closed***.

1.3.10

Writing Text Files

```
1  FileWriter fw =
2     new FileWriter("a_file.txt");
3  PrintWriter writer =
4     new PrintWriter(fw);
5  writer.print("Your age is: ");
6  writer.print(18);
7  writer.println(" years old.");
8  writer.flush();
9  writer.close();
```

A `FileWriter` and `PrintWriter` can be used in conjunction to make writing text fairly easy.

Write a command-line text editing tool that saves text that the user types to a file.

- Create a new Java class in a file named "`TextEdit.java`" and define a `main` method with the appropriate signature.
  - Prompt the user to enter a ***filename*** and use it to create a `FileWriter`.
  - Use the `FileWriter` to create a `PrintWriter` that you will use to ***print lines to the file***.
  - Use a `Scanner` and a ***loop*** to allow the user to enter text into ***standard input***. Use the `PrintWriter` to write each line of text to the file. Stop if the user enters a ***blank line***.
  - Don't forget to ***close*** your `Scanner` and `FileReader`!
  - Use a `try/catch` to handle any exceptions and exit gracefully.
- ***Compile*** and ***run*** your new class from the terminal.

`try`-with-resources

When using `try`***-with-resources***, resources are allocated in a ***semicolon-delimited list*** inside parentheses (shown in ***orange*** for emphasis).

```
1  try(FileWriter out =
2          new FileWriter(name);
3      PrintWriter writer =
4          new PrintWriter(out)) {
5
6      writer.println("Hello, File!");
7      writer.flush();
8  }
```

The ***scope*** of any resources that are opened is the `try` ***block*** - they can be used normally anywhere between the curly braces.

The resources are ***automatically closed*** when execution exits the `try` block, even if an exception is thrown in the block.

- A process that opens a file maintains a ***lock*** on that file until it is ***closed***.
  - This lock will prevent other processes from interacting with the file.
- However, ***properly*** closing a file can be challenging under some circumstances.
  - What if there is an exception?
  - What if the file fails to open at all?
  - What if closing the file throws an exception?!
- You should remember that Python includes a `with-as` statement that insures that a file is always closed, regardless of whether an error is raised.
- Java includes a similar feature: `try`***-with-resources***.
  - Resources like `FileInputStream` or `PrintWriter` are initialized in ***parentheses*** after the `try`.
  - Any such resources are ***automatically closed*** when the body of the try exits (after the closing curly brace).
- A `try`-with-resources ***does not*** need to be followed by a `catch` block.
  - If a `catch` block ***is*** used, the resources will be closed ***after*** it is executed.
  - If you ***do*** omit the catch block, a checked exception (like `IOException`) will need to be ***rethrown***.

1.3.11

`try`-with-resources

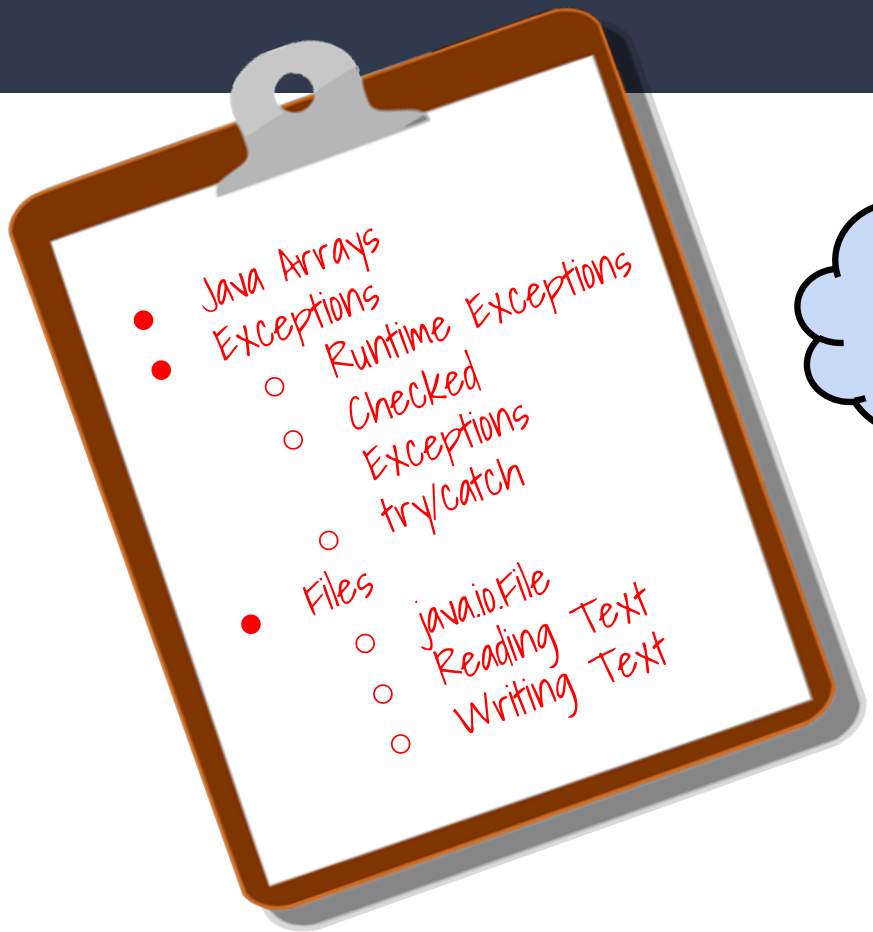
```
1  try(FileOutputStream out =
2          new FileOutputStream(name);
3      PrintWriter writer =
4          new PrintWriter(out)) {
5
6      writer.println("Hello, File!");
7      writer.flush();
8  }
```

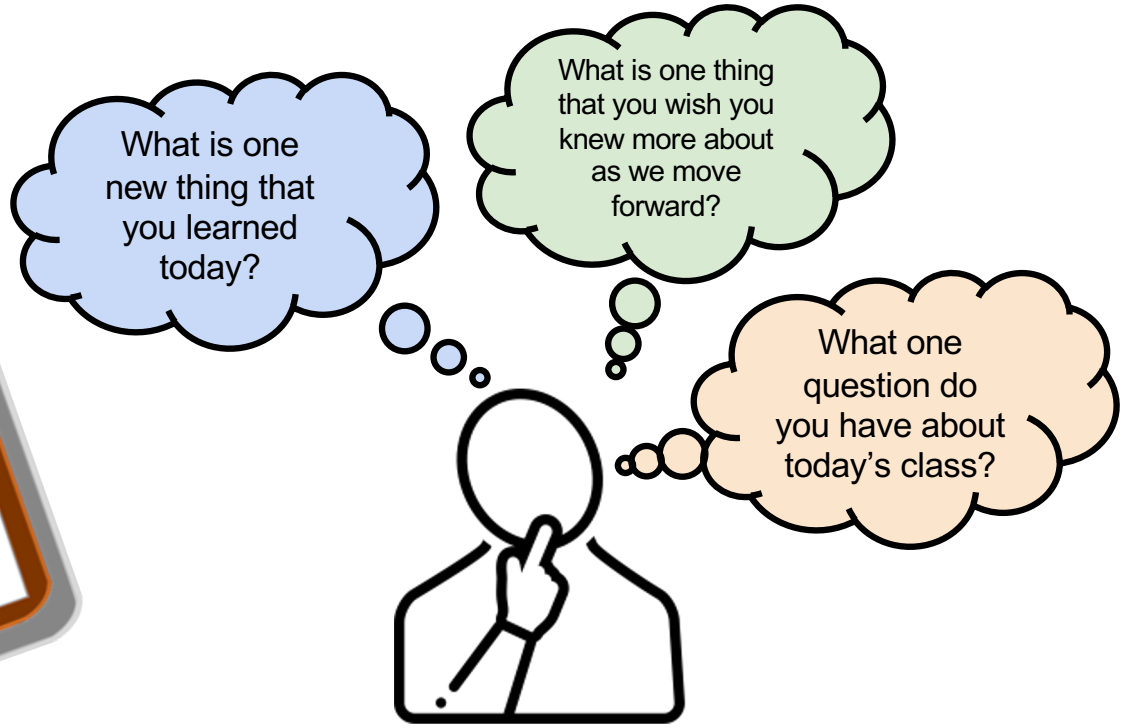
Any resources opened between the parentheses after the `try`-with-resources are ***automatically closed*** when the try block exits.

Update your `TextEdit` class to use try-with-resources to insure that the output file is closed.

- Open your `TextEdit` class and navigate to the `main` method.
  - Modify your implementation so that it uses `try`***-with-resources*** to open ***both*** the `FileWriter` and the `PrintWriter`.
  - You may delete the explicit calls to close both.
  - You will still need to make sure that the `Scanner` is closed.
- ***Compile*** and ***run*** your class from the terminal.

Summary & Reflection





Please answer the questions above in your notes for today.