GCIS-124
Software Development & Problem Solving

1: Python to Java

RIT | Golisano College of Computing and Information Sciences

SUN	MON (1/15)	TUE	WED (1/17)	THU	FRI (1/19)	SAT
	MLK Jr. Day		Unit 1: Python to Java			
	No Class		- Course Overview - Academic Honesty		Assignment 1.1 Due (start of class)	
SUN	MON (1/22)	TUE	WED (1/24)	THU	FRI (1/26)	SAT
	Unit 1: Python to Java		Unit 2: Java Classes			
	Assignment 1.2 Due (start of class)		Unit 1 Mini Practicum Assignment 1.3 Due (start of class)			

You Are Here

1.0 Accept the Assignment

You will create a new repository at the beginning of every new unit in this course. Accept the GitHub Classroom assignment for this unit and clone the new repository to your computer.

- Your instructor will provide you with a new GitHub classroom invitation for this unit.
- Upon accepting the invitation, you will be provided with the URL to your new repository, click it to verify that your repository has been created.
- You should create a `SoftDevII` directory if you have not done so already.
  - Navigate to your `SoftDevII` directory and clone the new repository there.
- Open your repository in VSCode and make sure that you have a terminal open with the **PROBLEMS** tab visible.

Asking for Help

Could be Improved

You have all the resources that you need to solve the problems that have been given to you, but sometimes you may not be able to find the answers to the problems that you encounter.

Asking for help in this course is both ***expected*** and ***encouraged***.

Spending time trying to solve the problem yourself will be a more valuable learning experience in the long run.

If you ***do*** ask for help, try to be as detailed as possible to make it easier for others to help you by providing as much detail as you can.

MUCH Better

Part 4.c. on homework 2.2 asks us to write a function that prompts the user to enter two floating point values and to print the product.

I reviewed slides 17-18 in the lecture, and I finished the practice activity. That all works fine.

But when I try to use the values entered by the user, I am getting an "input mismatch" error. Does anyone have a suggestion?

1. Begin by reviewing the lecture slides related to the problem that you are trying to solve
2. Try to solve the related class activities without looking at the answer
3. When asking for help, try to be as specific as possible about the problem you are having

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
- We will begin by focusing on ***procedural programming in Java***.

Why Another Language?

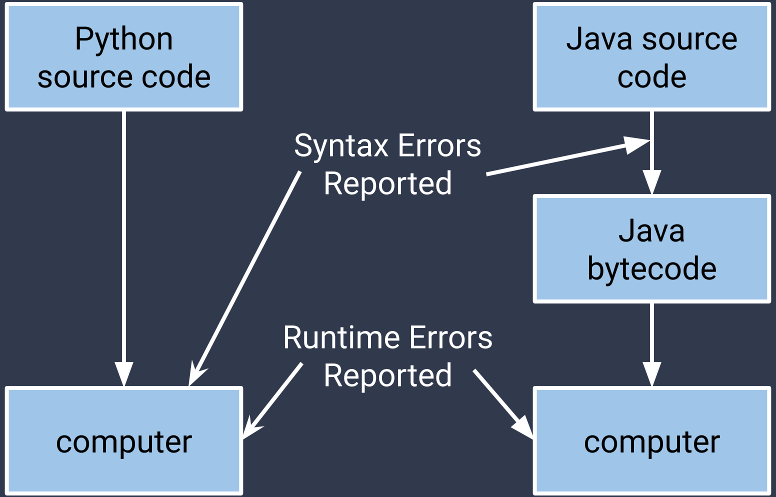
- Python is an incredibly powerful and flexible language.
  - It is also one of the most popular languages used today according to [GitHub]() and [Stack Overflow]().
- It is also a great learning language for inexperienced programmers because, while it is very powerful, it is also relatively quick and easy to pick up the basics.
  - "Hello, World!" is just `print("Hello, World!")`
- However, while Python supports classes and objects, it is not a great object-oriented language.
  - OO concepts like encapsulation, inheritance, overriding, and overloading are all very oddly implemented in Python when compared to other languages.
- There are also significant benefits to learning a new programming language.
  - Not only is it great for your resume, but the more languages that you learn, the more easily you will pick up the syntax of new languages.

I see that you know *two* languages. I'm sorry, but we only hire people who are irrationally attached to *one* language and unwilling to learn anything new.
Said no one, ever.

It's great to have a favorite language that is your "go to" for personal projects, but be careful not to get stuck in a professional rut.

Java

Python is ***implicitly compiled*** and executed in a single step. Both ***syntax*** and ***runtime errors*** are reported at the same time.



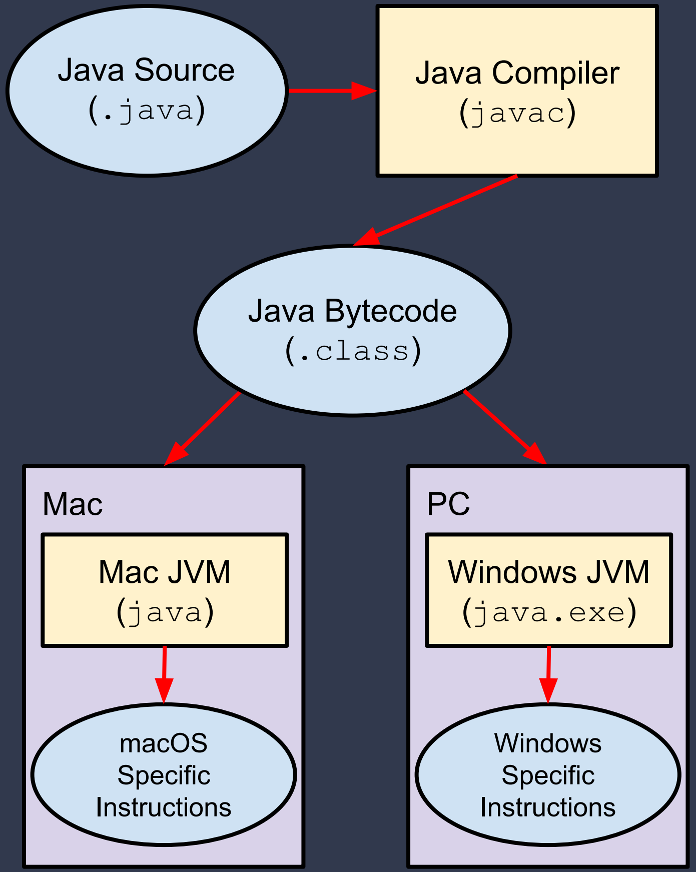
Java is ***explicitly compiled*** into ***bytecode*** first, at which time ***syntax errors*** are reported.

Once compiled, a Java program is executed as a separate step, at which time ***runtime errors*** are reported.

- Java is a ***fully*** object-oriented programming language.
    - This means that ***all*** code must be inside the ***body of a class***.
- Python is an ***interpreted*** language, meaning that code written in a Python program is interpreted at the time of execution.
    - This means that any ***syntax*** or ***runtime errors*** are reported at the same time.
- Java, on the other hand, is a ***compiled*** language.
    - Java programs are text files saved with a `.java` extension, e.g. `HelloWorld.java`.
    - Each must be ***explicitly compiled into bytecode*** using `javac` before it can be executed.
    - Any ***syntax errors*** are reported at this time.
    - The bytecode is saved in a `.class` file with the same name, e.g. `HelloWorld.class`.
- Once a Java program is compiled, it can be executed using `java`.
    - The program ***must*** include a ***main method*** with a specific signature to be executed.
    - Any ***runtime*** errors are reported at this time.

Java Bytecode

- ***High-level languages*** like Python and Java are meant for ***humans*** to understand.
  - A higher-level language has syntax that is closer to written or spoken language.
  - Python is a (slightly) higher-level language than Java.
  - On the other hand, ***assembly*** is a ***very*** low-level language.
- ***Machine code*** is very low-level code that can be executed directly on a computer.
  - Different computers understand different dialects of machine code.
- ***Bytecode*** is somewhere in the middle.
  - ***Lower-level*** than Python or Java source code.
  - ***Higher-level*** than machine code.
- Bytecode is much faster/easier to ***interpret*** (translate) into machine instructions than source code is.
- A ***virtual machine*** (***VM***) interprets bytecode into machine code at runtime.
  - In this way, bytecode is like machine code for the virtual machine.
  - The compiled bytecode never needs to change, but a different VM is needed for each operating system and/or processor to translate it properly.



1.1 Verify Your Java Version

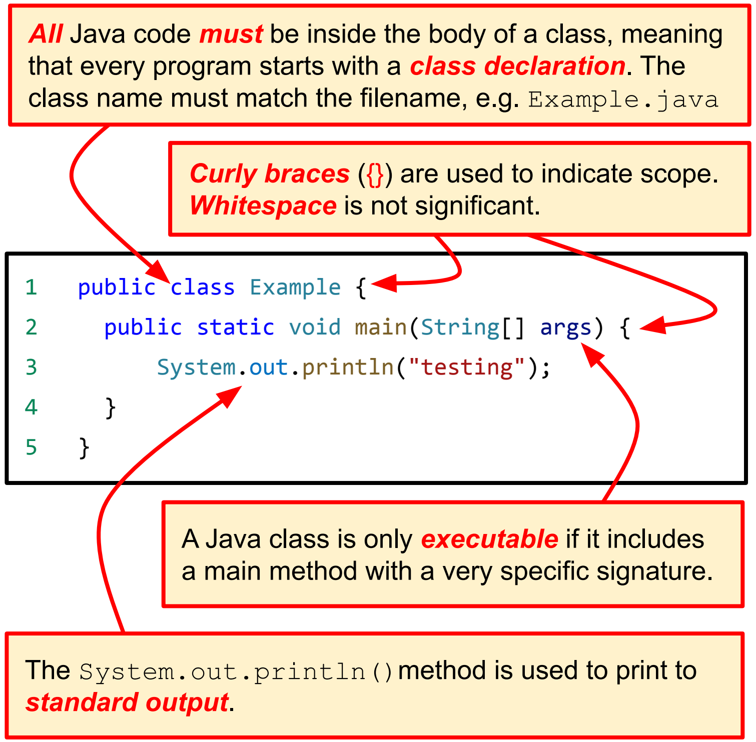
Before we can do anything else, you will need to make sure that the correct version of Java has been installed on your computer. Take a moment to verify that both Java and the Java Compiler have been installed and configured correctly.

- Open the terminal in VSCode (`CTRL-``) and run `javac -version` to verify the version of the ***Java Compiler*** that you have installed.
  - This is the executable that you will use to compile your Java programs.
- Next, run the `java -version` command to display the version of the ***Java Development Kit*** (***JDK***) that you have installed.
  - This is the executable that you will use to run your Java programs.
- Both executables should have the same version number.

```
dick@batcomputer MINGW64 ~/SoftDevII/unit01-nightw (main)
$ javac -version
javac 19.0.1
dick@batcomputer MINGW64 ~/SoftDevII/unit01-nightw (main)
$ java -version
java version "19.0.1" 2020-10-20
Java(TM) SE Runtime Environment (build 15.0.1+9-18)
Java HotSpot(TM) 64-Bit Server VM (build 15.0.1+9-18, mixed mode, sharing)
```

Scaffolding

- Unlike Python, Java requires quite a bit of ***scaffolding*** to write even the most basic programs.
  - Part of the reason for this is that ***all*** code in Java must be ***in the body of a class***, and so at a minimum a class must be declared.
- Java has a C-like syntax meaning:
  - ***Whitespace*** is ***not significant*** beyond improving readability for humans. This includes newlines.
  - ***Statements*** are terminated by a ***semi-colon*** (`;`) and may be written on multiple lines.
  - ***Curly braces*** (`{ }`) are used to indicate a ***block of statements***, e.g. the body of a method or a class.
- In Java, the `static` keyword marks a method or field as belonging to the ***class***.
  - The method or field can be accessed through the class ***without*** first creating an object.
  - ***All*** of the methods we write in this unit will be `static`.



All Java code ***must*** be inside the body of a class, meaning that every program starts with a ***class declaration***. The class name must match the filename, e.g. `Example.java`

Curly braces ({}) are used to indicate scope. ***Whitespace*** is not significant.

```java
public class Example {
  public static void main(String[] args) {
      System.out.println("testing");
  }
}
```

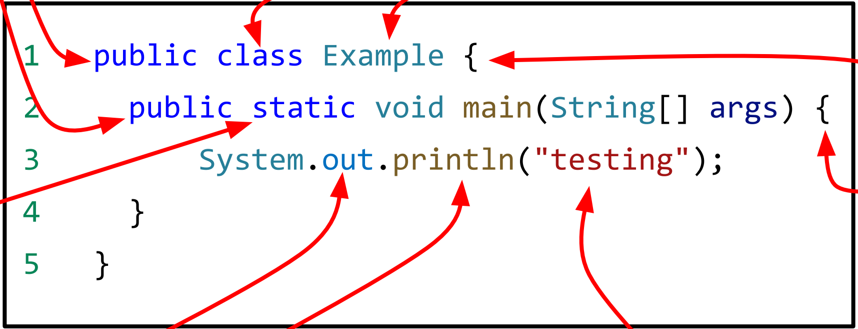
A Java class is only ***executable*** if it includes a main method with a very specific signature.

The `System.out.println()` method is used to print to ***standard output***.

A Closer Look (or "Why we don't use Java in 123.")

In Java, using the `public` ***access modifier*** makes things (classes, methods, fields) accessible from ***anywhere*** outside of the class. In this unit we will make all classes and methods `public`.

A Java ***class*** is declared using the `class` keyword, just like in Python. The name of the class must ***exactly match*** the name of the file (including case), e.g. the `Example` class must be in a file named `Example.java`.



```java
public class Example {
    public static void main(String[] args) {
        System.out.println("testing");
    }
}
```

The `static` keyword indicates that the method ***belongs to the class*** and can be called ***without*** creating an object.

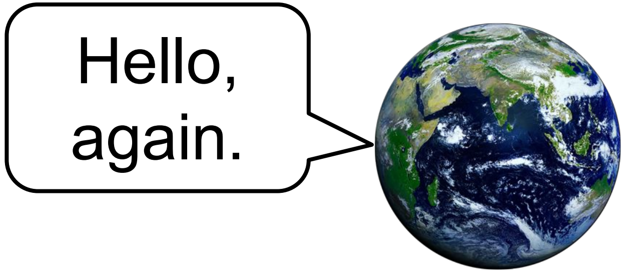
Whitespace is ***insignificant*** in Java. Instead, ***curly braces*** (`{}`) define blocks of code. Indents are used for ***readability only***.

In Java, `System.out` refers to ***standard output***. The `print` and `println` methods can be used to print virtually ***any*** type including strings, integers, and so on.

Java strings ***must*** be enclosed in ***double-quotes*** (`""`). You ***do not*** have the option of using single-quotes or triple-quotes of any kind.

1.2 Hello, World!

"Hello, World!" is often the first program written when learning the syntax of a new programming language. Let's start learning Java syntax by implementing it now.



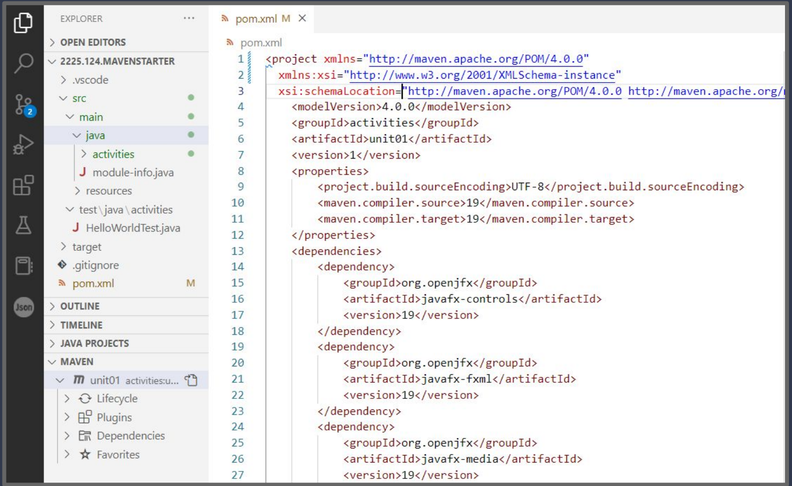
```java
public class Example {
  public static void main(String[] args) {
      System.out.println("testing");
  }
}
```

```
damian@batcomputer ~/SoftDevII/unit01
$ javac Example.java
damian@batcomputer ~/SoftDevII/unit01
$ java Example
testing
```

- Create a new Java class in a file named "`HelloWorld.java`" in the root folder of your project.
  - Define a `main` method with the appropriate signature and print `"Hello, World!"` to standard output.
- Open the terminal in VSCode (`CTRL-``) and use the Java Compiler to compile your new class.
  - e.g. `javac HelloWorld.java`
  - If there are any syntax errors, correct them in the editor and try to compile your class again.
  - List the files in the directory to verify that your `.class` file has been created.
- Once you have successfully compiled the class, run it using `java`.
  - e.g. `java HelloWorld`
  - ***Do not*** include the `.class` extension.

Maven

Maven projects must be organized in a very specific way so that Maven can find your Java code to compile and run it.



```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
    <modelVersion>4.0.0</modelVersion>
    <groupId>activities</groupId>
    <artifactId>unit01</artifactId>
    <version>1</version>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>19</maven.compiler.source>
        <maven.compiler.target>19</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-controls</artifactId>
            <version>19</version>
        </dependency>
        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-fxml</artifactId>
            <version>19</version>
        </dependency>
        <dependency>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-media</artifactId>
            <version>19</version>
```

Everything that Maven needs to know about your project is in a file named "pom.xml". You can feel free to look inside this file, but ***do not modify it!***

- Java projects can quickly become very large and complex.
    - You will write programs this semester that include hundreds of lines of code in dozens of different Java class files.
- It is also common that Java projects rely on third party libraries for functionality that is not included in the JDK, including some that we will be using this semester:
    - JUnit is a testing library similar to pytest that is used to write unit tests.
    - JavaFX is a set of libraries used to write Graphical User Interfaces (GUIs).
- Apache Maven is a build tool that is commonly used by software engineers in industry.
    - It is used to organize, compile, and run Java code.
    - It also manages third-party dependencies by automatically downloading them from the Internet.
- Maven projects ***must*** be organized in a specific way.
    - Your starter projects this semester will be configured to use Maven by default.

1.3 Verify Your Maven Version

Apache Maven is a build tool that is commonly used in industry to organize, compile, and run projects written in the Java programming language. Hopefully you downloaded and installed Maven by following the instructions for your operating system. Let's verify that now!

- If necessary, open the terminal in VSCode and run the following command:
  - `mvn -version`
- You should see output similar to the example below.
  - Make sure that Maven is using the correct version of Java to compile and run your code!

As you will see this semester, Java projects can get quite large and complex and have lots of dependencies on things like JUnit and JavaFX.

Maven makes building, compiling, and running Java projects with lots of dependencies relatively easy.

```
nightw@GypsyAvenger MINGW64 ~/SoftDevII/unit1-nightw (main)
$ mvn -version
Apache Maven 3.8.6 (84538c9988a25aec085021c365c560670ad80f63)
Maven home: C:\Program Files\apache-maven-3.8.6
Java version: 19.0.1, vendor: Oracle Corporation, runtime:
C:\Program Files\Java\jdk-19
Default locale: en_US, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family:
"windows"
```

1.4 Organizing Your Code

PROBLEMS ~~37~~

As mentioned previously, your project must be organized in a very specific way so that Maven can find, compile, and run your code - your Java code must be located inside of the "`src/main/java/unit01`" folder in your project. Let's move the code that we have written so far so that it is in the right place for Maven to find it.

Packages are a way to organize Java code by placing related classes in the same folder.

You must use a `package` statement that matches the name of the folder!

```java
package unit01;

public class HelloWorld {

}
```

- Expand the "`src/main/java/unit01`" folder in your project.
- Move the "`HelloWorld.java`" class into the folder by clicking and dragging it.
  - If VSCode does not automatically add "`package unit01;`" to the top of your file, you will need to add it manually.
  - Java packages are a way of organizing your code. We will discuss Java packages in more detail in the near future.
- Open the **PROBLEMS** tab in VSCode and make sure that you do not have any errors.
  - If there is an error that you cannot solve on your own, please raise your hand!
  - You may need to close and reopen the file for the errors to go away.
  - If all else fails, restart VSCode!

Variables & Assignment

- Java is a ***statically typed*** language, meaning that the type of a variable must be specified ***when it is declared***.
  - Once declared, only ***compatible*** values may be assigned to the variable.
- Unlike Python, a Java variable may be declared ***without*** immediately assigning a value.
  - e.g. `int radius;`
  - Variables may also be ***declared and initialized*** at the same time, e.g. `double pi = 3.14159;`
- Like Python, Java includes ***two*** basic kinds of types.
  - ***Primitive types*** include ***integers***, ***floating point values***, ***characters***, and ***booleans***.
    - Primitive types are very similar to Python's value types.
    - Java includes a ***character type***, a single character enclosed in single-quotes, e.g. `char c = 'c';`
  - ***Reference types*** are, well, everything else including ***strings***, ***arrays***, and other ***objects***.
    - Reference types work the same in Java as they do in Python.

Type	Description	Example
`byte`	8-bit signed integer, `-128` to `127`.	`123`
`short`	16-bit signed integer.	`12345`
`int`	32-bit signed integer.	`1234567`
`long`	64-bit signed integer.	`1234567l`
`char`	16-bit unsigned integer (Unicode).	`'a'`
`float`	32-bit signed floating point value.	`3.14159f`
`double`	64-bit signed floating point value.	`-0.1234`
`boolean`	Boolean value; `true` or `false`.	`true`

Java assumes that ***literal numbers*** are `int` or `double`, and so a suffix of `l` (lowercase `L`) is needed for a `long` literal (e.g. `3098765341l`), and `f` for a `float` literal (e.g. `1.3f`).

Variables in Java

```
public static void variables() {
        double weight = 65.5;

        int age;

        age = 10;

        System.out.println("weight = " + weight
        + ", age = " + age);
}
```

Variables in Java

Because Java is a statically typed language, variables must be declared with a valid type. Only values of a compatible type may be assigned to the variable.

Like any other statement in Java, a variable declaration must be terminated with a semicolon.

```java
public static void variables() {
    double weight = 65.5;

    int age;

    age = 10;

    System.out.println("weight = " + weight
    + ", age = " + age);
}
```

Unlike Python a variable may be declared without immediately assigning a value…

…but a value must be assigned before the variable can be used.

Unlike Python, the + operator can be used to concatenate ***any*** type onto a string (not just other strings).

1.5 Primitive Types

Primitive types in Java include integers (`byte`, `short`, `int`, `long`), floating point values (`float`, `double`), characters (`char`), and `boolean` values. Practice declaring a few variables using different primitive types and printing them to standard output.

```
public static void variables() {
    String name = "Buttercup";
    int age;
    age = 10;
    System.out.println("name = " + name
       + ", age = " + age);
 }
```

- Create a new Java class in a file named "`Primitives.java`" in the "`src/main/java/unit01`" folder.
- Define a `main` method with the appropriate signature.
  - Declare variables of at least ***four*** different ***primitive types*** and print the ***name***, ***type***, and ***value*** to standard output.
  - e.g. `int x = 5;`
  - ***Hint***: unlike Python, you can concatenate a value of ***any*** Java type onto a string using the + operator, e.g. `System.out.println("four = " + 4);`
- Use the "play" button in VSCode to run your class.

Arithmetic Operators

- Java supports ***most*** of the same basic ***arithmetic operators*** that Python does with a couple of notable exceptions.
  - There is no power (`**`) operator in Java, but the `Math.pow()` method may be used instead, e.g. `Math.pow(2, 7)` will return $2^7 = 128$.
  - There is no integer division operator (`//`) in Java, but using standard division with two integer values has the same result.
- The other basic operators all work the same with some possible caveats.
- Applying an operator to two values of the ***same type*** will produce ***a value of that type***.
  - For example, `10 / 3` will perform ***integer division***. In this case, the expression would evaluate to an `int` (`3`).
- If values of different types are mixed in the same expression, the result is "***promoted***" to the ***most complex type***.
  - For example `2.5 * 3` would evaluate to a `double` (`7.5`).

Operator	Description	Example
`+`	Addition	`double x = 5.1 + 3;`
`-`	Subtraction	`double z = 5.1 - 3;`
`*`	Multiplication	`int x = 3 * 4;`
`/`	Division	`double x = 5.1 / 3;`
`/ (integers)`	Integer Division	`int x = 5 / 2;`
`Math.pow(x, y)`	Power: X^Y	`double z;` `z = Math.pow(x, y);`
`%`	Modulo	`m = 10 % 3`

The `+` ***operator*** also works for ***string concatenation***, just as it does in Python. A notable difference is that ***any type*** can be concatenated onto a string without using a function like `str()`, so `"abc" + 123` ***is valid***.

1.6 Arithmetic

Arithmetic operators in Java work very similarly to those in Python, including the order of operations and the way that they behave when combining integer and floating point values. Practice using the Java operators now.

Operator	Description	Example
`+`	Addition	`double x = 5.1 + 3;`
`-`	Subtraction	`double z = 5.1 - 3;`
`*`	Multiplication	`int x = 3 * 4;`
`/`	Division	`double x = 5.1 / 3;`
`/ (integers)`	Integer Division	`int x = 5 / 2;`
`Math.pow(x, y)`	Power: X^Y	`double z;` `z = Math.pow(x, y);`
`%`	Modulo	`m = 10 % 3`

- Create a new Java class in the "`src/main/java/unit01`" folder in a file called "`Arithmetic.java`" and define a `main` method with the appropriate signature.
  - Use `System.out.println` to print the results of using arithmetic operators with combinations of different types, e.g.:
    - `int` and `int`
    - `int` and `double`
    - `double` and `double`
  - For example: `System.out.println("12 * 3.6 = " + (12 * 3.6));`
  - There is no need to declare variables, but remember that Java will consider a ***literal integer*** to be an `int` (32-bit) and a literal ***floating point value*** to be a `double` (64-bit).
- Use the VSCode run button to run the class.

Strings & Characters

`String` is the type used for Java strings. ***String literals*** must be enclosed in ***double-quotes*** (`""`).

```
String aString = "Jason";
char first = aString.charAt(0);
char last = aString.charAt(4);

System.out.println(first);
System.out.println(last);
```

Java strings do not support the use of square brackets (`[]`), but the `charAt(int index)` method will return the `char` at a specific index.

As with Python, ***valid indexes*** range from ***0*** to ***length-1***. Java ***does not*** support negative indexes.

- Java ***strings*** have a lot in common with Python strings.
  - They comprise ***characters***.
  - ***Literal strings*** are enclosed in ***double-quotes*** (`""`).
  - Strings are ***immutable***.
  - The `+` ***operator*** can be used to create a new string by ***concatenating*** two strings together.
  - Strings are ***reference types***.
- There are some key differences as well.
  - Literal strings ***cannot*** use single-quotes (`''`) or triple-quotes of either type.
  - Java includes a `char` type that represents a single character. Single quotes are used for `char` ***literals***, e.g. `'a'`, `'1'`, `'&'`.
  - The `+` ***operator*** can concatenate strings together with ***any*** other Java type.
  - Java strings ***do not*** work with square brackets (`[]`). Instead the `charAt(int index)` method is called on the string to get the `char` at a specific index.
  - Strings are not ***iterable***.
- Also like Python, Java makes use of a ***string literal pool***.
  - If the same string literal is used in more than one place, a single copy of the string is stored and reused.

1.7 Java Strings

Making the transition from Python strings to Java strings can take some getting used to, especially because you can't access individual characters in a Java string using `[]`. Start getting used to using the `s.charAt(index)` to access the individual characters in Java strings.

```java
String aString = "Jason";
char first = aString.charAt(0);
char last = aString.charAt(4);

System.out.println(first);
System.out.println(last);
```

'J'	'a'	's'	'o'	'n'
0	1	2	3	4

Java strings are built on top of an ***array of characters***, each of which is accessible via its index using `charAt(int index)`.

- Create a new Java class in the "`unit01`" folder in a file called "`Strings.java`" and declare a `main` method with the appropriate signature.
  - Create a `String` ***variable*** containing ***at least 5 characters*** using the literal of your choice.
  - Use the `charAt(int index)` method to print the ***first*** and ***last*** characters in the string.
  - Print ***at least 3*** of the remaining characters.
- Use the VSCode run button to run the class.

Methods & Parameters

All of the methods that we write in this unit will be `public` (visible to ***all*** parts of the program) and `static` (called on the ***class***, ***not*** an object).

All Java methods must declare a ***return type***. A `void` return type indicates that the method does not return a value.

```java
public static void example(int x, double y) {
    System.out.println(x + " * " + y
        + " = " + (x * y));
}
```

Parameters must declare a ***type*** as well as a ***name***. When the method is invoked, a ***compatible*** argument ***must*** be provided for each parameter.

- You should recall that a function that belongs to a class is called a ***method***. ***All*** code in Java must be within the body of a class, therefore ***all*** functions in Java are methods.
  - Java methods ***must*** be declared between the ***curly braces*** (`{...}`) that define the ***body*** of a class.
- There are some key differences between Python and Java.
  - There is ***no*** `self` ***parameter***.
  - Like any other Java variable, parameters ***must*** be declared with a ***type*** as well as a ***name***.
  - All methods ***must*** declare a ***return type***. A method that does not return anything declares a `void` return type.
  - An ***access modifier*** ***<u>may</u>*** be used to set the ***visibility*** of the method outside of the class. In this unit, all of the methods that we write will be `public`.
- In Python, a method that does not explicitly return a value returns `None` by default. In Java, a method with a void return type ***does not return anything*** at all.
  - Trying to assign the return value of a void method to a variable is a ***compiler error***.

1.8 Hello, Methods!

Java methods must declare types for any parameters *and* the type that will be returned by the method (or void if nothing is returned). Practice now by writing a Java method and calling it from main.

- Open the `HelloWorld` class that you wrote previously and define a method called "`helloName`" that declares `String` parameters for a `first` and `last` name.
  - Don't forget to make the method `public` and `static`.
  - Print a message in the format `"Hello, <first> <last>!"` to standard output.
- Call your method from `main`.
- Run the class.

```
public static void example(int x, double y) {
   System.out.println(x + " * " + y
      + " = " + (x * y));
}
```

Return Values

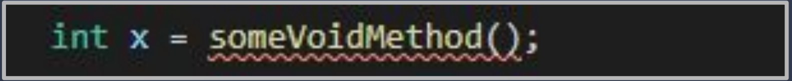
- All Java methods ***must*** declare a ***return type***, even if no value is returned.
  - The return type is declared as part of the ***method signature***.
- Unlike Python, there is ***no*** default return type in Java. The `void` return type indicates that a method ***does not*** return a value of ***any type***.
  - A `void` method ***cannot*** be assigned to a variable, and trying to do so will cause a ***compiler error***, e.g. `int x = someVoidMethod();`
  - A `void` method ***cannot*** return a value, though it may use an empty `return` ***statement***, e.g. `return;`
- If a method declares a return type of ***anything other than*** `void`, then a value of that type ***must*** be returned by the method using a `return` statement, e.g. `return 5;`
  - This also means that, if there are ***multiple branching paths*** through the method, each and every branch ***must terminate*** in a `return` statement.
  - A method that does not return a value of the declared type ***will not compile***.

A method that declares a void ***return type*** does not return a value of any type, though a return statement can be used without a value.

```java
public static void someVoidMethod() {
    System.out.println("no return");
    return;
}

public int intReturn() {
    return 5;
}
```

A method that declares a return type of any other type ***must*** return a value of a ***compatible type***. If not, the method will ***not compile***.

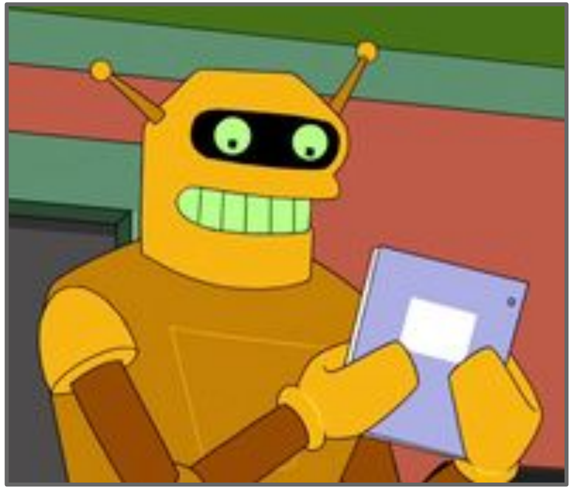
Assigning a method with a `void` return type to a variable of any type will cause a ***compiler error***.



```java
int x = someVoidMethod();
```

1.9 Calculon

In Python, all methods return a value (even if that value is `None`). In Java, a method must declare its return type - if the return type is not `void`, the method ***must*** return a value of the correct type or the program will not compile. Try writing a few methods that return values.



```java
public int intReturn() {
      return 5;
}
```

- Create a new Java class in the "`unit01`" folder in a file called "`Calculon.java`".
- Define a method for each of the four basic arithmetic operations, each of which declares ***two floating point parameters*** and returns a floating point result.
  - `add`
  - `subtract`
  - `multiply`
  - `divide`
- Test your methods by calling them from `main` and printing the results to standard output.
- Run the class.

Conditionals

Python	Java	Example
`not`	`!`	`!a, !(a || b)`
`or`	`||`	`a || b`
`and`	`&&`	`a && b`
`^`	`^`	`a ^ b`
`is, is not`	`==, !=`	`a == b, a != b`
`==`	`== (primitives)` `equals(Object)`	`x == 5` `a.equals(b)`
`<, <=`	`<, <=`	`a < b, a <= b`
`>, >=`	`>, >=`	`a > b, a >= b`

A side-by-side comparison of ***logical*** and ***comparison operators*** in Python and the equivalent operators in Java.

- Recall that a ***boolean expression*** is one that, when evaluated, results in a `boolean` ***value***, i.e. `true` or `false`.
- Just like Python, ***logical*** and ***comparison operators*** can be combined to create complex boolean expressions in Java.
- Also like Python, Java supports ***conditional statements*** that work very much the same.
  - `if(expression) {...}` - executes the statements in the body if the expression is `true`.
  - `else {...}` - executes the statements in the body if all of the ***preceding conditions*** are `false`.
- The most notable difference is that Java ***does not*** include an `elif` statement.
  - Instead, `else if(expression) {...}` is used.

```java
if(a && b) {
    System.out.println("foo");
} else if(b ^ c) {
    System.out.println("bar");
} else {
    System.out.println("foobar");
}
```

1.10 Conditioning

Python	Java	Example
not	`!`	`!a, !(a || b)`
or	`||`	`a || b`
and	`&&`	`a && b`
`^`	`^`	`a ^ b`
`is`, is not	`==, !=`	`a == b, a != b`
`==`	`== (primitives)` `equals(Object)`	`x == 5` `a.equals(b)`
`<, <=`	`<, <=`	`a < b, a <= b`
`>, >=`	`>, >=`	`a > b, a >= b`

```
if(a && b) {
    System.out.println("foo");
} else if(b ^ c) {
    System.out.println("bar");
} else {
    System.out.println("foobar");
}
```

By now you have lots of experience using conditionals in Python. Conditionals in Java work exactly the same way, but use a different syntax. Practice the Java syntax by writing a method that prints a different message depending on whether or not a number is divisible by 2, 3, or 5.

- Create a new Java class in a file named "`Conditional.java`" and define a method named "`evenlyDivisible`" that declares a parameter for an integer `n`.
    - Print ***exactly one*** of the following messages depending on the value of `n`:
        - The number is ***even***
        - The number is ***divisible by 3***
        - The number is ***divisible by 5***
        - The number is ***odd but not divisible by 3 or 5***
- Call your method from `main` with several values of `n`.
- Run the class.

`while` Loops

- Java `while` loops work ***exactly*** the same way as while loops in Python with a few small ***syntactic differences***.
  - The boolean expression that controls the loop ***must*** be enclosed in ***parentheses*** (`()`).
  - If the ***body*** of the loop contains more than one statement, it ***must*** be enclosed in ***curly braces*** (`{}`).
- Java also supports statements for more ***finely grained*** control of a `while` loop.
  - The `break` statement will ***terminate*** a loop ***immediately***, regardless of the boolean expression.
  - The `continue` statement will ***stop the current iteration***. The boolean expression will be evaluated to determine whether or not the next iteration should be executed.

A `while` loop will execute ***zero or more*** iterations depending on whether the boolean expression is `true` the very first time it is evaluated.

```java
int i = 1048576;
while(i != 2) {
  System.out.println(i);
  i = i / 2;
}
```

The loop will continue iterating until the boolean expression evaluates to `false`.

The `break` and `continue` statements work the same as they do in Python and can be used for more finely grained control of the iteration.

1.11 Counting Up with `while`

The `while` loop in Java works just like Python's `while` loop, but uses a different syntax (you may be sensing a theme here). Practice Java's `while` loop syntax by implementing a method that uses a `while` loop to print a count up from `0` to `n`.

In Java, the boolean expression used by a `while` ***loop*** must be enclosed in ***parentheses***...

```java
int i = 128;
while(i != 2) {
  System.out.println(i);
  i = i / 2;
}
```

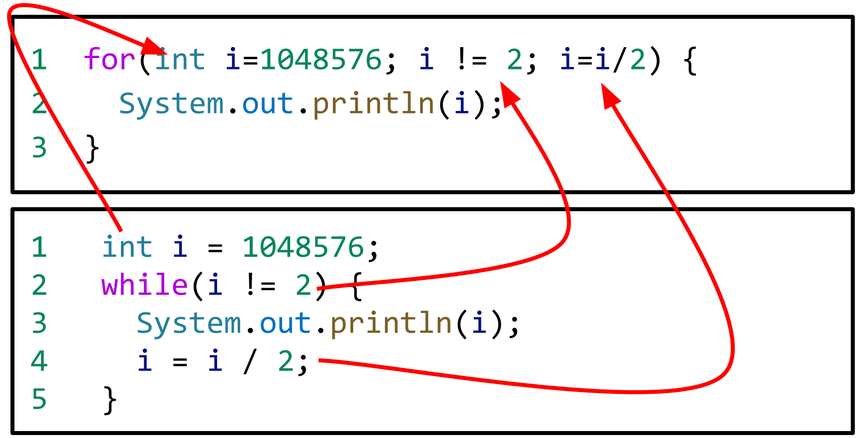
...and the statements in the ***body*** of the loop must be enclosed in ***curly braces*** (`{}`).

- Create a new Java class in a file named "`CountUp.java`" and define a method named "`countWhile`" that declares a parameter for an integer `n`.
  - Use a `while` ***loop*** to print a count from `0` ***to*** `n`.
  - Return the ***sum*** of the numbers that are counted.
- Call your method from `main` and print the sum.
- Run the class.

"Classic" `for` Loops

- Java's ***"classic"* `for` *loop*** is a more ***syntactically compact*** version of the counting `while` loop.
- The ***loop declaration*** includes three statements separated by ***semicolons*** (`;`).
  - The ***initialization statement*** initializes the counting variable and is executed ***exactly once*** before the first iteration, e.g. `int i=1048576;`
  - The ***boolean expression*** that controls the loop. It is evaluated ***before*** each iteration of the loop, e.g. `i!=2;`
  - The ***modification statement*** that is executed ***after*** each iteration of the loop, e.g. `i=i/2;`
- All three statements are ***optional***.
  - If omitted, the boolean expression is ***always*** `true`.
- Java ***also*** includes a `for` ***each loop*** similar to Python's that works with iterable types.
  - We'll take a look at it later in this unit.

The ***"classic"* `for` *loop*** uses a ***C-like syntax*** to make a more syntactically compact version of a counting `while` loop.



```java
for(int i=1048576; i != 2; i=i/2) {
  System.out.println(i);
}
```

```java
int i = 1048576;
while(i != 2) {
  System.out.println(i);
  i = i / 2;
}
```

The ***initialization statement***, ***boolean expression***, and ***modification statement*** are all included in the loop declaration.

A Closer Look at "Classic" `for` Loops

A counting `while` loop has a few basic parts…

The initialization of the counting variable, which happens before the first iteration.

The boolean expression that stops the loop. This is evaluated *before* each iteration.

```
int count = 0;
while(count < 11) {
    System.out.println(count);
    count++;
}
```

The statement that modifies the counting variable. This is usually executed at the *end* of an iteration.

A for loop is a more compact syntax for all of the same parts.

The initialization of the counting variable, which happens before the first iteration.

The boolean expression that stops the loop. This is evaluated *before* each iteration.

```
for(int c=0; c<11; c++) {
    System.out.println(c);
}
```

The statement that modifies the counting variable. This is executed at the *end* of an iteration.

1.12 Counting up with `for`

Python does not have a loop like Java's "classic" or "C-Style" `for` loop. However, it works just like a counting `while` loop with a more compact syntax. Practice the syntax now by implementing a second method that counts from `0` to `n` using a `for` loop.

Java's `for` ***loop*** is a ***syntactic shortcut*** for a counting `while` loop.

The ***counting variable*** is initialized in the loop ***header***. The ***boolean expression*** is evaluated ***before*** each iteration.

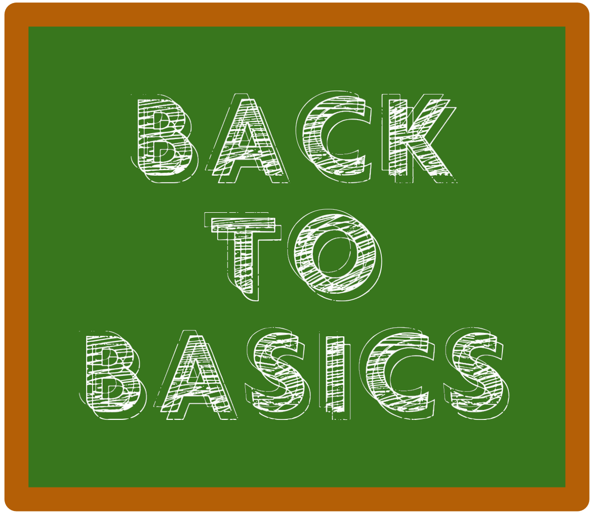
```java
for(int i=1024; i != 2; i=i/2) {
    System.out.println(i);
}
```

The ***modification statement*** is executed ***after*** each iteration.

- Open your `CountUp` class and define a method called "`countFor`" that declares a parameter for an integer `n`.
    - Use a `for` ***loop*** to print a count from `0` ***to*** `n`.
    - Return the ***sum*** of the numbers that are counted.
- Call your method from `main` and print the sum.
- Run the class.

1.13 Java Basics

Up to this point you have only written a small amount of Java code, so let's do a quick practice exercise so that you can continue to reinforce Java syntax. Write a new class that includes a main method that uses a loop to print the integers between 1 and 100 that are multiples of 3 or 7.



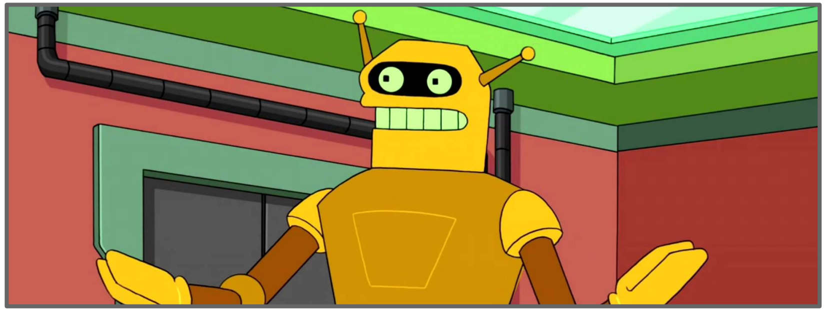
Java programs require a lot of scaffolding, even for something as simple as "Hello, World!"

- Create a new Java class in a file called "`Basics.java`" and define a `main` ***method*** with the appropriate signature.
  - Use a loop to print all of the integers that are multiples of 3 or 7 between 1 and 100.
  - ***Challenge***: ***Do not*** print numbers that are multiples of ***both*** 3 and 7.
- Run the class.

```java
public class MyClass {
  public static void main(String[] args) {
        int x;
        double pi = 3.14159;
        System.out.println("Hello, GCIS-124!");
     }
}
```

1.14 Calculon++

In the last class, you wrote a four-function calculator in a class named `Calculon`. Let's practice writing methods by adding a fifth function; a `raise` method that uses a loop to compute and return an exponent.



```
public static int factorial(int n) {
   int result = 1;
   while(n > 1) {
      result = result * n;
      n = n - 1;
   }
   return result;
}
```

- Open your `Calculon` class and define a new method named "`raise`" that declares parameters for a floating point `base` and an integer `exponent`.
  - Use a loop to compute the result of raising the `base` to the power of the `exponent`.
- ***Call*** your new method from `main`.
- Run the class.

Casting Numbers

- In Python, type conversions are handled using the ***constructor*** of the desired type, e.g.
  - `a_string = str(123)`
  - `a_list = list("abcdef")`
- In Java, converting a ***less complex type*** to a ***more complex type*** is considered ***safe*** because there is no risk of ***data loss***. Such conversions may happen automatically, e.g.
  - `long x = 123; // int to long`
  - `double y = 12.34f; // float to double`
- However, converting from a ***more complex type*** to a ***less complex type*** risks data loss and is considered to be ***unsafe***. Attempting to do so will cause a compiler error (***incompatible types***), e.g.
  - `int x = 3246123456l; // long to int`
  - `long y = 1234.567; // double to float`
- You may ***force*** the conversion by ***casting***, wherein the desired type is specified in parentheses on the right side of the assignment statement, e.g.
  - `int x = (int)3246123456l;`
  - `long y = (long)1234.567;`
- This is a signal to the compiler that it is OK if there will be some data loss, e.g. the fractional part of a decimal.

Safe Type Conversions (automatic)	Example
smaller integer → larger integer	`int x = 1234;` `long y = x;`
integer → floating point	`int x = 1234;` `double y = x;`

Unsafe Type Conversions (*<u>requires</u>* casting)	Example
larger integer → smaller integer	`long x = 1234l;` `int y = (int)x;`
floating point → integer	`float x = 12.34f;` `int y = (int)x;`

Conversions between ***incompatible types***, e.g. `boolean` and `int`, will result in a ***compilation error*** under any circumstances (even with a cast).

1.15 Casting

Casting is only necessary when assigning a value of a more complex type to a variable of a less complex type, e.g. a `double` value to an `int` variable. Try it out now to see what happens when you cast values of different types.

```
double x = 47.2;
int y = (int)x; // cast needed
```

When casting, the type to cast into is specified in parentheses next to the value being cast.

- Create a new Java class in a file called "`Casting.java`" and define a `main` method with the appropriate signature.
  - Experiment with casting by creating variables different types and trying to cast them into variables of another type. Print the values before and after casting. Try at least two of the suggestions below:
    - `int` to `long`
    - `long` *>3 Billion* to `int`
    - `char` to `int`
    - `int` in the range *33 ≤ i ≤ 126* to `char`
    - `boolean` to `int`
- Run the class.

Standard Input

The `Scanner` class is in the `java.util` package, and so must be ***imported***. Imports are done at the top of the class file.

```java
import java.util.Scanner;
```

The `new` keyword will create a new `Scanner` by ***calling its constructor***. The `Scanner` needs to be configured to read from a specific data source, e.g. ***standard input*** (`System.in`).

```java
Scanner scanner = new Scanner(System.in);
System.out.print("Enter age: ");
int age = scanner.nextInt();
int months = age * 12;
System.out.println("Age in months: " + months);
scanner.close();
```

Methods like `next()`, `nextLine()`, and `nextInt()` will return input typed by the user into the terminal. The `Scanner` should be ***closed*** when it is no longer needed.

- `System.out` is a reference to ***standard output*** and the `println` and `print` functions can be used to direct output to the terminal.
- Similarly, `System.in` is a reference to ***standard input***, and it ***can*** be used to read user input from the terminal, but it's a little unwieldy to use.
- As an alternative to `System.in`, Java provides the `java.util.Scanner` class, which can be used to read data from any input source, e.g. standard input, files, etc.
  - A `Scanner` needs to be told from where to read by passing the input source into its ***constructor***, e.g. `Scanner s = new Scanner(System.in);`
- `Scanner` provides lots of useful methods:
  - `next()` returns the next word typed by the user (up to the next whitespace).
  - `nextLine()` return everything up to the point where the user pressed the enter key.
  - `nextInt()`, `nextLong()`, `nextFloat()`, etc. returns the next word as the corresponding type.
- It is considered good programming practice to ***close*** a `Scanner` when you are finished using it.

1.16 Hello, You!

You may remember that "Hello, You!" is a modified version of the classic "Hello, World!" program that prompts the user to enter their name and prints a customized "Hello!" message to standard output. Practice using `Scanner` to implement it now.

```
import java.util.Scanner; // import!
```

```
Scanner scanner = new Scanner(System.in);
System.out.print("Enter age: ");
int age = scanner.nextInt();
int months = age * 12;
System.out.println("Age in months: " + months);
scanner.close();
```

- Create a new Java class in a file named "`Hello.java`" and define a new method named "`helloYou`".
  - Prompt the user to enter their ***name***.
    - Hint: the `System.out.print()` method will not terminate with a newline, so the user can type on the same line as the prompt.
  - Use a `Scanner` to read the user's input and store it in a variable.
    - Remember, `Scanner` is in the `java.util` package. Don't forget to import it!
  - Print a message in the format `"Hello, <name>!"`
- ***Call*** your method from `main`.
- Run the class.

1.17 Calculon+++

We'll be using the `Scanner` a lot to read user input from standard input. Let's practice a little more by making an improvement to the five function calculator by prompting the user to enter the values to add, subtract, multiply, divide, and raise.

```java
import java.util.Scanner; // import!
```

```java
Scanner scanner = new Scanner(System.in);
System.out.print("Enter age: ");
int age = scanner.nextInt();
int months = age * 12;
System.out.println("Age in months: " + months);
scanner.close();
```

- Open your `Calculon` class and update your `main` method.
  - ***Prompt*** the user to enter two floating point operands.
  - ***Print*** the results of calling ***all five methods*** on your calculator.
- Run the class.

JUnit Unit Tests

- ***JUnit*** is a ***unit testing framework*** that is similar to pytest in Python.
  - You will write a separate ***JUnit test*** for each Java class in your program.
  - The unit test ***is a Java class***. By convention, the unit test is named the same as the class under test with `"Test"` ***suffix***, e.g. `"MyClassTest"`.
  - You will write ***one or more test methods*** for each non-trivial method in your Java class.
- JUnit uses Java ***annotations*** to find test methods inside of your unit test.
  - `@Test` annotates each ***test method***.
  - JUnit and/or VSCode will not be able to find the tests if they are not annotated properly.
- JUnit also includes many ***built in assertions***, e.g.
  - `assertEquals(expected, actual)`
  - `assertEquals(float1, float2, delta)`
  - `assertTrue(actual)`
  - `assertNotNull(actual)`
  - etc.
- Using JUnit with VSCode requires configuring your project to include ***the JUnit library***.
  - You should have downloaded and configured this as part of the pre-semester assignment.

Each ***test method*** is marked with the `@Test` annotation so that JUnit knows which methods to execute.

```java
public class EuclidGCDTest {
    @Test
    public void gcd100() {
        // setup
        int a = 100;
        int b = 60;
        int expected = 20;

        // invoke
        int actual = EuclidGCD.gcd(a, b);

        // analyze
        assertEquals(expected, actual);
    }
}
```

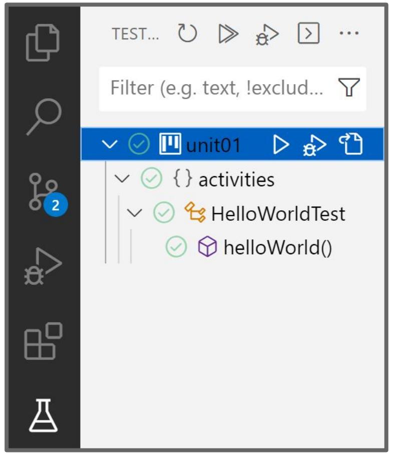
A good JUnit tests follow the same pattern as pytest: ***setup***, ***invoke***, and ***analyze*** using ***assertions***.

1.18 Running Your First JUnit Test

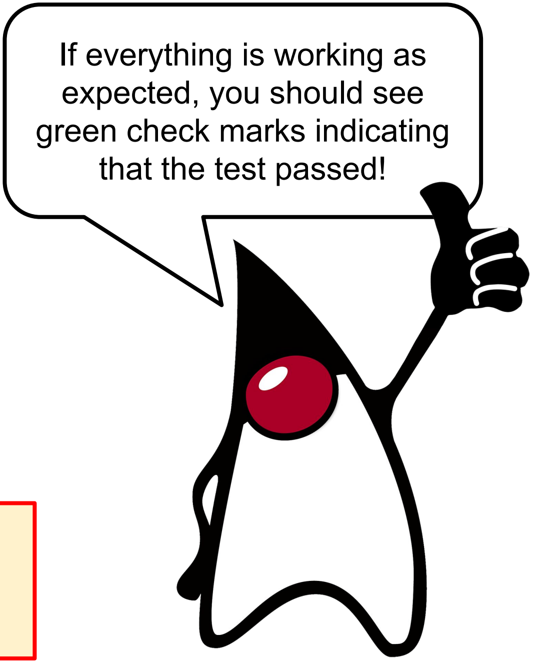
The project that you have been given should already be configured to run JUnit tests, including a "`HelloWorldTest`" that has been provided for you. Make sure that everything is configured properly by using VSCode's Test Runner ( ) to run the provided unit test now.

Open the provided `HelloWorldTest`. You can find it under "`test`" in your root project folder.

```
import org.junit.jupiter.api.Test;

public class HelloWorldTest {
    @Test
    public void helloWorld() {
        assertEquals("Hello, World!",
            "Hello, World!");
    }
}
```



Open the VSCode Test Runner ( ) and use "Run Test" ( ) to run the provided test.


If everything is working as expected, you should see green check marks indicating that the test passed!

PROBLEMS 37

Whenever you see this image, it's a reminder to check your PROBLEMS tab.

If VSCode has identified any problems, try to fix them. If you're not sure how, raise your hand and ask for help!

And remember: it's not about *avoiding* making mistakes. It's about learning how to fix them as we make them.

1.19 Writing Your First JUnit Test PROBLEMS ~~37~~

Now that we have verified that we can run JUnit tests and have moved all of our code into the right location in the project, let's write our own unit test. Tests in Java should be set up the same as they are in Python (setup, invoke, analyze) and should be small, fast, and repeatable.

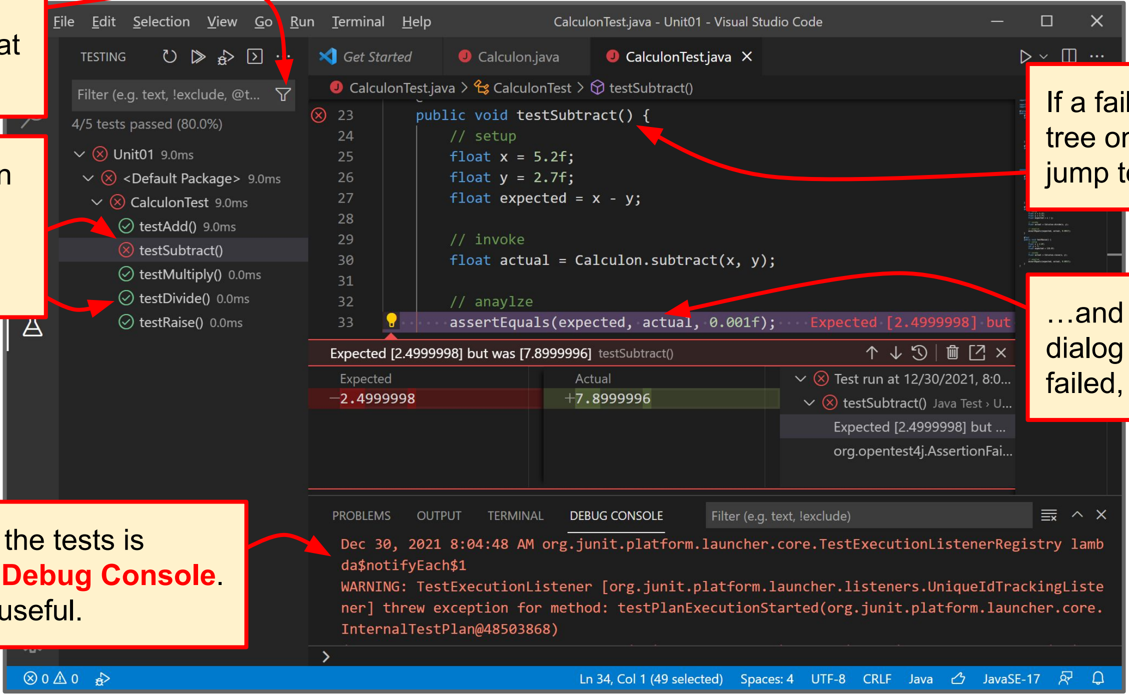
```java
@Test
public void exampleTest() {
  // setup
  int x = 5;
  int y = 2;
  int expected = 7;

  // invoke
  int actual = x + 1;

  // analyze
  assertEquals(expected,
               actual);
}
```

- Create a new JUnit test named "`CalculonTest`" in the same folder as "`HelloWorldTest`" and define a test method named "`testAdd`".
  - Make sure to add the `@Test` ***annotation*** to the method!
  - Call the `static add` method on your `Calculon` class and save the result in a new variable, e.g. `float actual = Calculon.add(5.1f, 7.2f);`
  - Use the JUnit `assertEquals` ***method*** to assert that the ***actual*** value matches the ***expected*** value.
    - ***Hint***: only type the first few characters of the name, e.g. "`assertE`", and then use VSCode's autocomplete feature to import the `assertEquals` method from JUnit.
- Click the ***flask icon*** in your VSCode sidebar to open the ***Java Test Runner***.
  - Click the ***run all tests button*** at the top of the test runner.
  - You will see that VSCode runs each test method in each of your tests, usually from top to bottom.
  - If a test passes, it is marked with ✔ and if it fails, it is marked with **X**.

Test Output in VSCode

Filter to see ***all*** tests, only ***failed*** tests, or only tests that ***passed***.

When expanded, the tree on the left shows tests that passed with a ✔ and tests that failed with an **X**.



```java
public void testSubtract() {
    // setup
    float x = 5.2f;
    float y = 2.7f;
    float expected = x - y;

    // invoke
    float actual = Calculon.subtract(x, y);

    // anaylze
    assertEquals(expected, actual, 0.001f);    Expected [2.4999998] but
```

```
Expected [2.4999998] but was [7.8999996]  testSubtract()
Expected                 Actual
-2.4999998               +7.8999996
Test run at 12/30/2021, 8:0...
  testSubtract()  Java Test › U...
    Expected [2.4999998] but ...
    org.opentest4j.AssertionFai...
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Filter (e.g. text, !exclude)

```
Dec 30, 2021 8:04:48 AM org.junit.platform.launcher.core.TestExecutionListenerRegistry lamb
da$notifyEach$1
WARNING: TestExecutionListener [org.junit.platform.launcher.listeners.UniqueIdTrackingListe
ner] threw exception for method: testPlanExecutionStarted(org.junit.platform.launcher.core.
InternalTestPlan@48503868)
```

⊗ 0 ⚠ 0
Ln 34, Col 1 (49 selected) Spaces: 4 UTF-8 CRLF Java JavaSE-17

If a failing test is clicked in the tree on the left, the editor will jump to that test…

…and show a detailed (if clunky) dialog explaining which assertion failed, and why.

The output from the tests is displayed in the **Debug Console**. But it's not very useful.

1.20 More JUnit Tests

We will be writing lots of JUnit tests this semester to verify that our code is working properly. Let's get some more practice by writing tests for some of the other functions for the `Calculon` five-function calculator. We will want at least one test for each function.

```
@Test
public void exampleTest() {
  // setup
  int x = 5;
  int y = 2;
  int expected = 7;

  // invoke
  int actual = x + 1;

  // analyze
  assertEquals(expected,
               actual);
}
```

- Open the `CalculonTest` JUnit test and define at least one test method for each of the remaining methods in your calculator.
  - `subtract`
  - `multiply`
  - `divide`
  - `raise`
- After writing each new test, ***run*** your JUnit test using the ***Java Test Runner*** to make sure that the test passes.
  - Remember, if you'd like to see why a test ***failed***, click on it in the test runner tree and the editor will jump to it and show you why.
- Remember: you can also run your tests using Maven.
  - `mvn clean compile test`
  - What does it look like when a test fails?

1.21 Reverse a String

PROBLEMS ~~37~~

Let's practice some of the Java basics that we've 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.

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

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

Remember: you can access individual characters in a string using the `charAt(index)` method inside of a `for` loop.

- Create a new Java class named "`Miscellany`" 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.
  - Remember you can use concatenation (the + operator) to concatenate *anything* onto a string.
  - Return the reversed copy.
- Run the class.

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.

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

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***.

1.22 Array of Squares

Like so many other things we have talked about in this unit, arrays in Java are very similar to those that you used in Python. The only real difference is the syntax for creating and using them. Practice now by implementing a method that returns an array of squares.

```
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]);
}

String s = Arrays.toString(numbers);
```

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

- 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 `Arrays.toString()` method to print your array to standard output (see the example to the left).
- Run the class.

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...

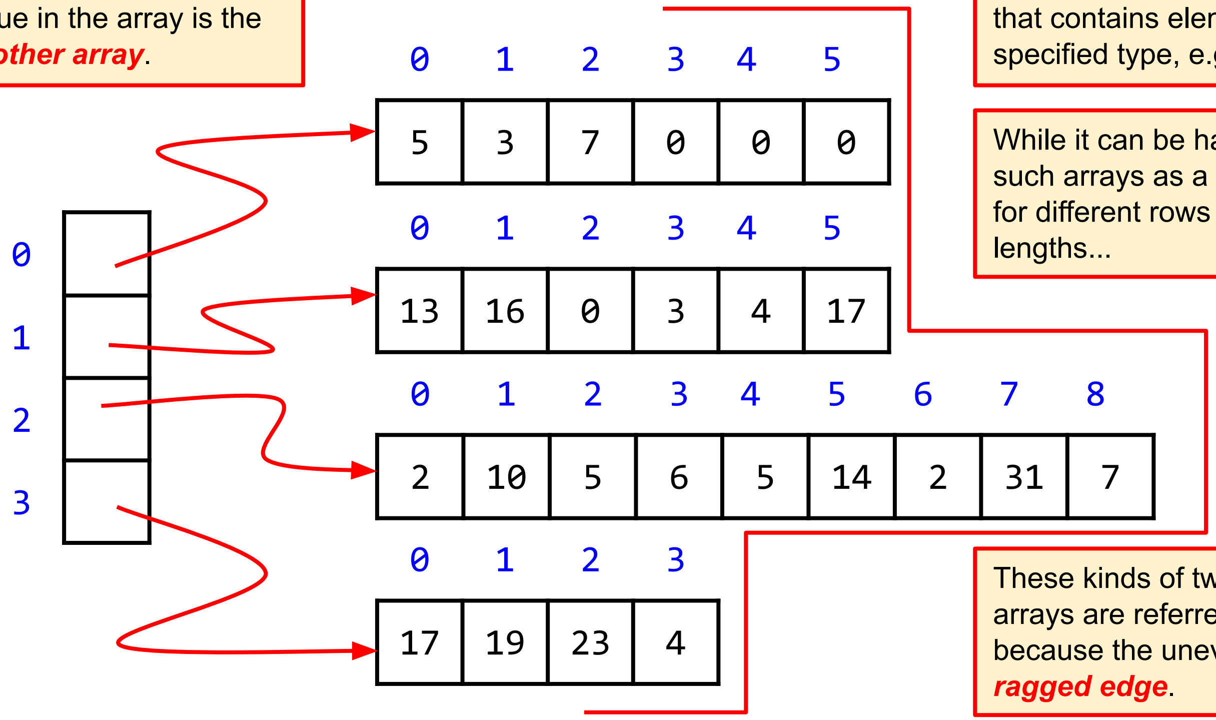
- In Python we experimented with two-dimensional lists. In Java, it is possible to create ***two-dimensional arrays***.
  - A 2D array is ***an array of arrays***.
  - Like any other array, a two-dimensional array ***must*** be declared with a ***type***, and can only be used to store values of that 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***.

A Closer Look at Two-Dimensional Arrays

The first "***dimension***" is really just an array. Each value in the array is the ***address*** of ***another array***.

And each "***row***" is really an array that contains elements of the specified type, e.g. integers.

While it can be handy to imagine such arrays as a table, it is possible for different rows to be different lengths...



These kinds of two-dimensional arrays are referred to as "***ragged***" because the uneven lengths create a ***ragged edge***.

1.23 Multiplication Tables

You may remember using 2D-lists in Python to create multiplication tables. Let's try the same thing in Java using a two dimensional array instead.

```
// 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.

- 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.
- Run the class.

1.24 Invalid Input

We wrote many Python programs that expected the user to use standard input to enter numeric values into a prompt. If they entered non-numeric values, the program would crash with a `ValueError`. What happens if the user enters non-numeric data into your `Calculon` program?

In Python, trying to convert an invalid string into an integer or float will cause an error.

If the error is not handled using `try/except` then the program will crash.

- Run the `Calculon` program.
  - Use standard input to enter non-numeric data for one of the two values.
  - What happens?

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.

```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!");
}
```

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.

1.25 Using `try/catch` in Java Programs

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.

```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!");
}
```

- Open the `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.
- Run your updated class and enter non-numeric data.
  - What happens?

- 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.

```java
File file = new File("a_file.txt");
String path = file.getAbsolutePath();
long size = file.length();
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***.

1.26 Getting File Info

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.

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 VSCode's autocomplete feature to see which methods are available.

- Create a new Java class named "`Files`" 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.
- Run the class.

* On macOS you may need to use the absolute path to the file.

Reading Text Files

- 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.

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

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

1.27 Reading Text Files

PROBLEMS ~~37~~

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 VSCode?
- What happens when you try to 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***).

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

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.

1.28 Handling Checked Exceptions

PROBLEMS ~~37~~

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();
   }
 }
```

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.
- Run the class.

- 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.

```java
FileWriter fw = new FileWriter("a_file.txt");
PrintWriter writer = new PrintWriter(fw);
writer.print("Your age is: ");
writer.print(18);
writer.println(" years old.");
writer.flush();
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 should be ***closed***.

1.29 Writing Text Files

Using Java to write to text files is just as complicated as reading files. It will take lots of practice for you to get used to it. Using the code examples below and the code you wrote for previous activities in this unit, write a command-line text editing tool that saves text that the user types to a file.

```java
FileWriter fw =
    new FileWriter("a_file.txt");
PrintWriter writer =
    new PrintWriter(fw);
writer.print("Your age is: ");
writer.print(18);
writer.println(" years old.");
writer.flush();
writer.close();
```

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

- Create a new Java class named "`TextEdit`" 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.
- Run the class.

`try`-with-resources

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

```java
try(FileWriter out =
        new FileWriter(name);
    PrintWriter writer =
        new PrintWriter(out)) {

  writer.println("Hello, File!");
  writer.flush();
}
```

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***.

1.30 Using `try`-with-resources in Java

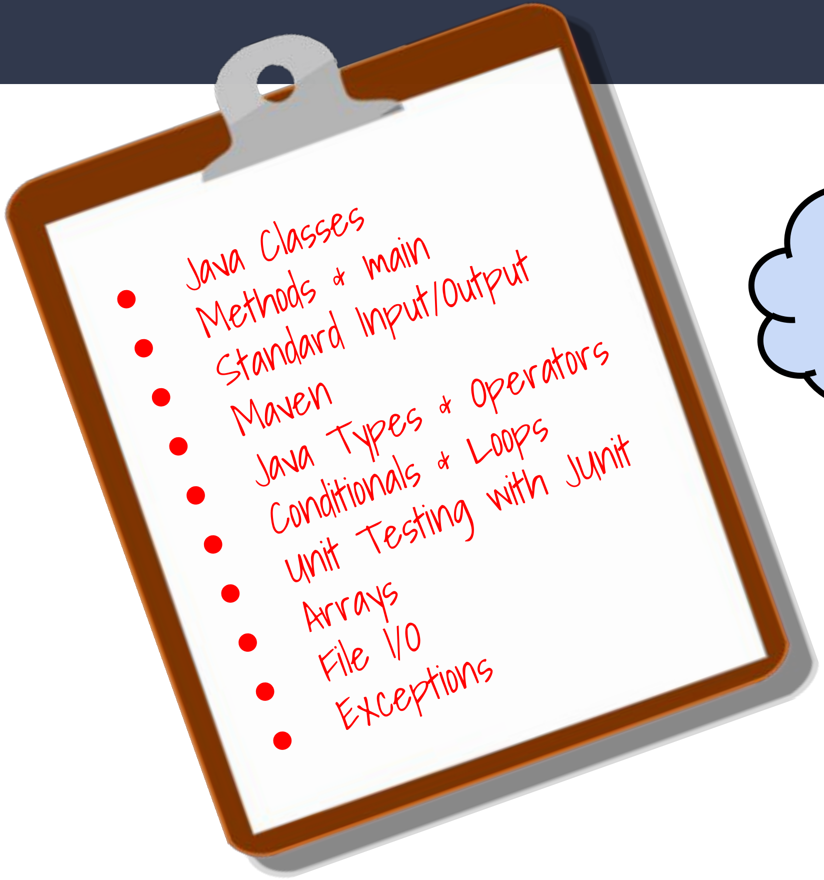
In Python we used with-as to open files and make sure that the file was closed (even if an error occurred). Using `try`-with-resources in Java works exactly the same way. Practice using it by updating your `TextEdit` class to use try-with-resources to insure that the output file is closed.

```java
try(FileWriter out =
        new FileWriter(name);
    PrintWriter writer =
        new PrintWriter(out)) {

   writer.println("Hello, File!");
   writer.flush();
}
```

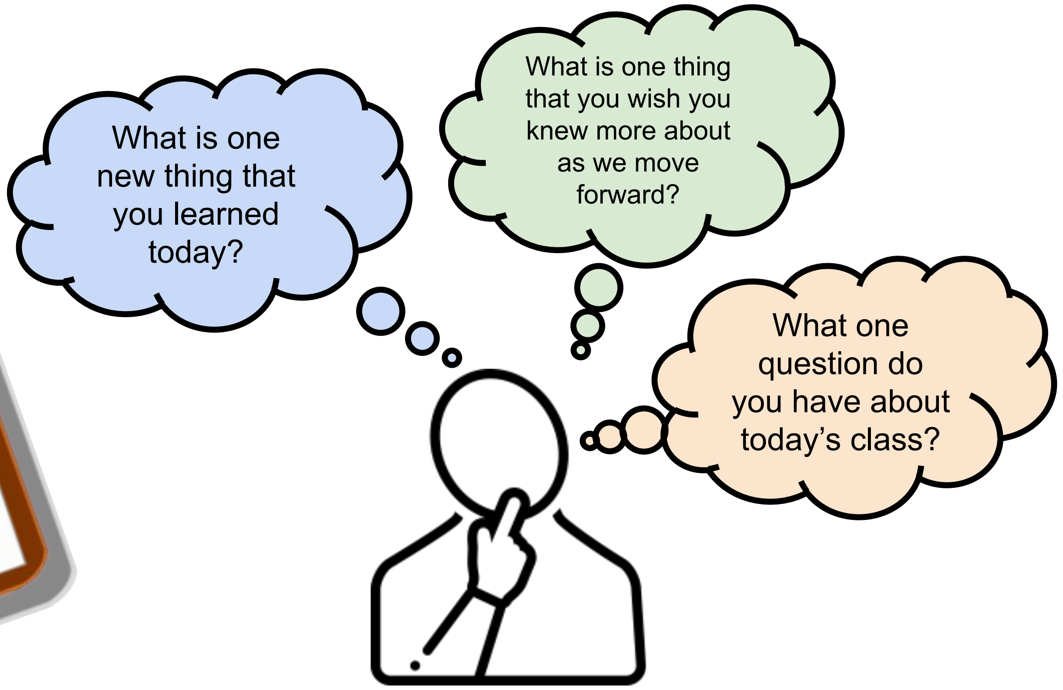
Any resources opened between the parentheses after the `try`-with-resources are ***automatically closed*** when the try block exits.

- 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.
- Run the class.

Summary & Reflection



- Java Classes
- Methods & main
- Standard Input/Output
- Maven
- Java Types & Operators
- Conditionals & Loops
- Unit Testing with JUnit
- Arrays
- File I/O
- Exceptions


What is one new thing that you learned today?
What is one thing that you wish you knew more about as we move forward?
What one question do you have about today's class?

Please answer the questions above in your notes for today.