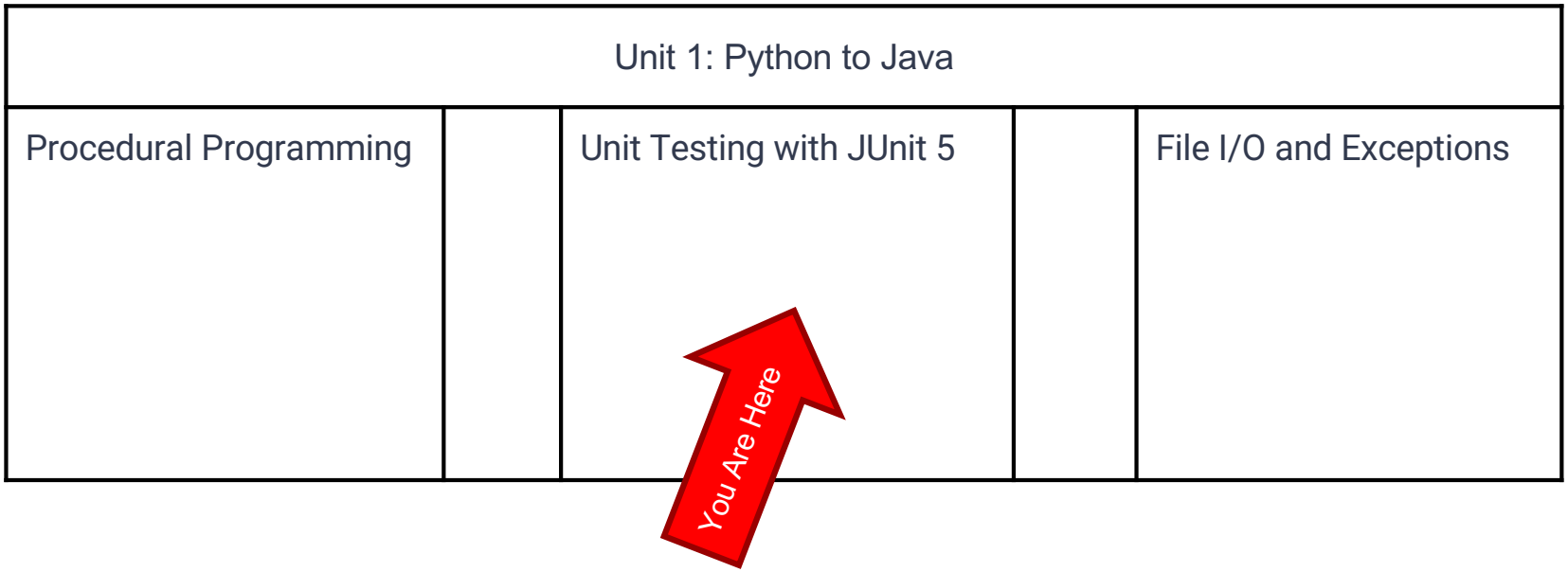
GCIS-124
Software Development & Problem Solving

1.2: Unit Testing with JUnit 5

RIT | Golisano College of Computing and Information Sciences

This Week: Python to Java



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 ***more Java syntax*** and creating ***unit tests with JUnit***.

- Java is a fully object oriented language, and so ***all*** code must be inside the ***body of a class***.
  - The `public` ***access modifier*** indicates that the class, method, or field is visible to the entire program.
  - The `static` ***modifier*** means that the class, method, or field is accessed through the ***class*** and not an object.
- ***Whitespace*** in Java is insignificant.
  - ***Blocks of code*** are enclosed in curly braces (`{}`).
  - ***Statements*** are terminated with a semicolon (`;`).
- ***Primitive types*** in Java include integers (`byte`, `short`, `int`, `long`), floating point values (`float`, `double`), characters (`char`) and Booleans (`boolean`).
- All other types in Java are ***reference types***.
- Java variables must be declared with a ***type*** and a valid ***identifier***. The variable may or may not be initialized with a value when it is declared, e.g.
  - `int x;`
  - `double pi = 3.14159;`
- The `System.out` is a reference to ***standard output***, and the `println` and `print` methods are used to print (with or without a newline, respectively).
  - `System.out.println("Hello, world!);`

Review: Java Basics

All code in Java must be within the body of a ***class***. The `public` access modifier indicates ***global visibility***.

Curly braces (`{}`) enclose ***blocks of code***.
Whitespace is only used for readability.

```
1  public class MyClass {
2      public static void main(String[] args) {
3          int x;
4          double pi = 3.14159;
5          System.out.println("Hello, GCIS-124!");
6      }
7  }
```

Being a ***statically typed*** language, Java variable declarations must include the ***type***. Only values of ***compatible types*** can be assigned.

In order to be ***executable***, a Java class must include a main method with a very specific signature.

Review: Java Conditionals and Loops

Boolean expressions in Java combine ***logical operators*** (and (`&&`), or (`||`), not (`!`), and xor (`^`)) and ***comparison operators*** (`==`, `!=`, `<`, `<=`, `>`, `>=`) and evaluate to `true` or `false`.

```
1  if(a && b || !c ^ d) {
2    System.out.println("foo");
3  } else if(x <= y) {
4    System.out.println("bar");
5  } else {
6    System.out.println("foobar");
7  }
```

Java ***conditionals*** combine `if(expression) {...}` and `else {...}` statements that work the same as they do in Python. There is no explicit `elif` in Java; instead `else if(expression) {...}` is used.

Java's ***"classic"*** `for` ***loops*** use a ***C-like syntax*** and are a more syntactically compact version of a counting `while` loop.

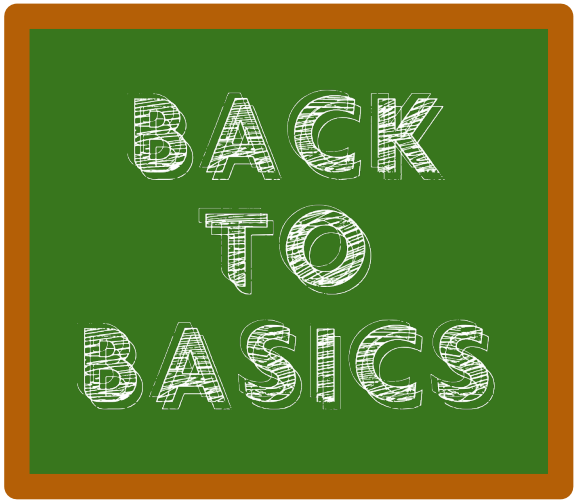
```
1  for(int i=1048576; i != 2; i=i/2) {
2    System.out.println(i);
3  }
```

```
1  int i = 1048576;
2  while(i != 2) {
3    System.out.println(i);
4    i = i / 2;
5  }
```

Java `while` loops work the same as Python `while` loops, with only a few syntactic differences: the boolean expression is in ***parentheses*** (`()`) and the body is in ***curly braces*** (`{}`).

1.2.2

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.

- Create a new Java class 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.
- ***Compile*** and ***run*** your new class from the terminal.

Review: Java Methods

A Java method signature ***must*** include a ***return type***, a ***name***, and ***zero or more parameters***.

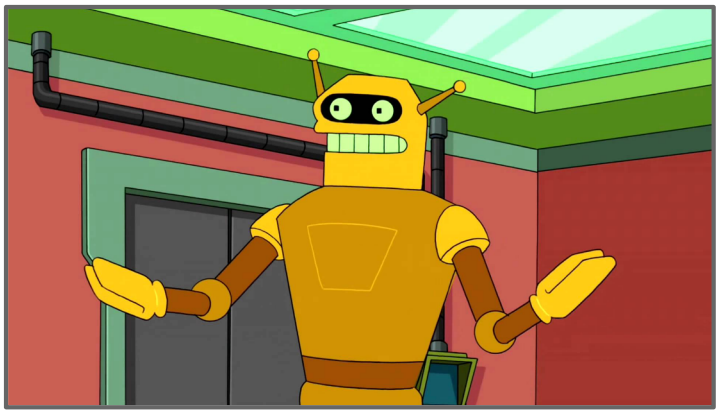
```
1  public static int factorial(int n) {
2     int result = 1;
3     while(n > 1) {
4        result = result * n;
5        n = n - 1;
6     }
7     return result;
8  }
```

Declaring a method `public` and/or `static` is ***optional*** (but we will do both throughout this unit).

If a return type other than void is declared, the method ***must*** return a value of a compatible type.

- You should recall that a function that belongs to a class is a ***method***.
- Because ***all*** Java code must be part of a class, ***all*** functions in Java are methods.
- The ***signature*** of a Java method includes several parts:
  - An optional ***access modifier*** that determines the visibility of the method outside of the class. In this unit, all of our methods will have `public` access.
  - The `static` ***modifier*** is used if the method belongs to the class. In this unit, all of our methods will be `static`.
  - A ***return type***.
    - A method that does not return a value declares the `void` return type. Such methods ***must not*** return a value.
    - Otherwise, the method ***must*** return a value of the declared return type.
  - A valid ***name***.
  - A list of ***zero or more parameters***, each of which must specify a ***type*** and a ***name***. When the method is called, and argument ***must*** be provided for each parameter.

1.2.3



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.

- 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`.
- ***Compile*** and ***run*** your class from the command line.

- 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;`
  - `double y = 12.34f;`
- 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 = 32461234561;`
  - `long y = 1234.567;`
- 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)32461234561;`
  - `long y = (long)1234.567;`
- This is a signal to the compiler that it is OK that there will be some data loss, e.g. the fractional part of a decimal.

Casting Numbers

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 (*requires* casting)	Example
larger integer □ smaller integer	`long x = 1234l;` `int y = (int)x;`
floating point □ integer	`float x = 12.34f;` `int y = x;`

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

1.2.4

Casting

Sometimes casting can have… ***unintended consequences***.

```
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.

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.

- 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 $33 \leq i \leq 126$ to `char`
    - `boolean` to `int`
- ***Compile*** and ***run*** your new class from the VS Code terminal.

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.

```
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`).

```
1  Scanner scanner = new Scanner(System.in);
2  System.out.print("Enter age: ");
3  int age = scanner.nextInt();
4  int months = age * 12;
5  System.out.println("Age in months: " + months);
6  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.

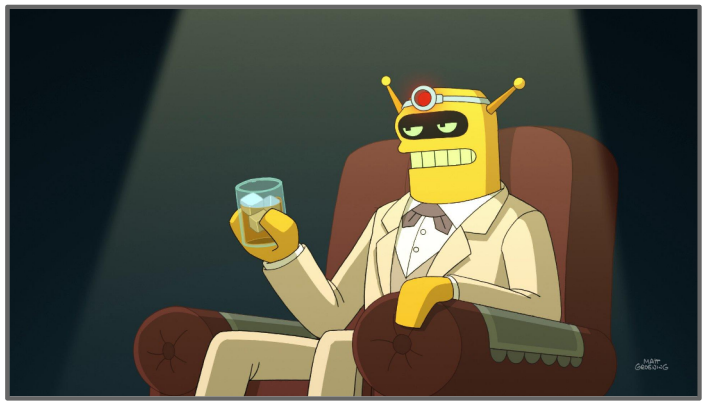
1.2.5

Hello, You! v2.0

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.

- 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.
  - 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`.
- ***Compile*** and ***run*** your class from the VS Code terminal.

1.2.6



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.

- 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.
- ***Compile*** and ***run*** your class from the VS Code terminal.

- ***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 unit tests and test methods.
  - `@Testable` annotates the ***unit test class***.
  - `@Test` annotates each ***test method***.
  - JUnit and/or VS Code 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 VS Code requires configuring your project to include ***the JUnit library***.
  - You should have downloaded and configured this as part of the pre-semester assignment.

JUnit Unit Tests

The `@Testable` annotation marks a class as a JUnit test so that VS Code's ***Java Test Runner*** can find it in your project.

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

```
1  @Testable
2  public class EuclidGCDTest {
3      @Test
4      public void gcd100() {
5          // setup
6          int a = 100;
7          int b = 60;
8          int expected = 20;
9
10         // invoke
11         int actual = EuclidGCD.gcd(a, b);
12
13         // analyze
14         assertEquals(expected, actual);
15     }
16 }
```

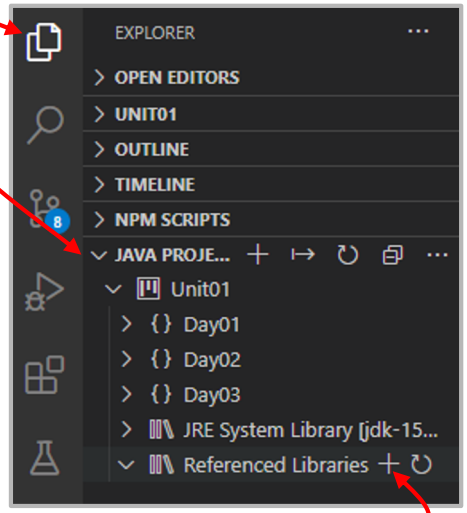
A good JUnit tests follow the same pattern as pytest: ***setup***, ***invoke***, and ***analyze*** using ***assertions***.

1.2.7

Configuring VS Code for JUnit

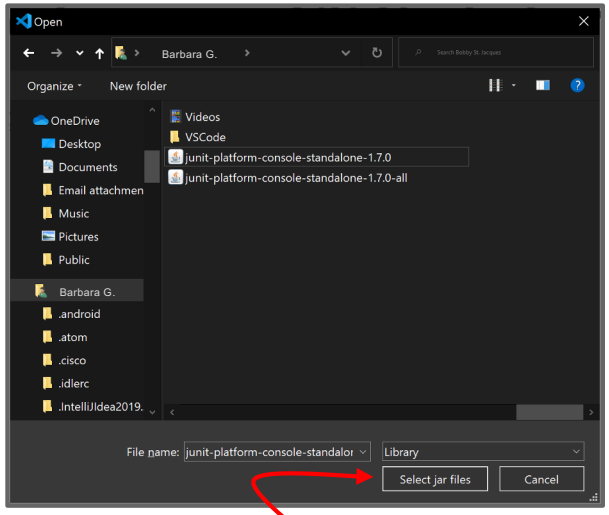
Before we can get started writing unit tests, we will need to configure your VS Code project to use the JUnit library.

Open the ***VS Code Explorer*** and expand the ***JAVA PROJECTS*** section.



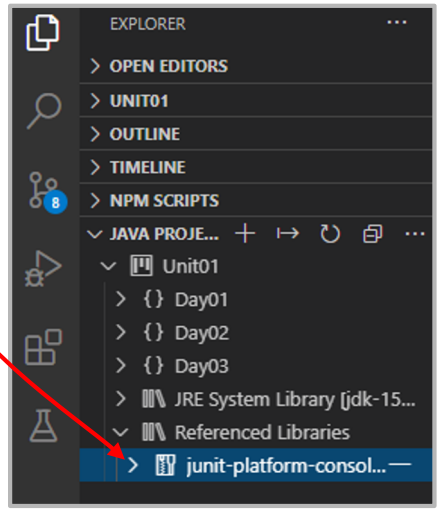
Mouse over ***Referenced Libraries*** until the + appears and click it.

Navigate to the location on your computer where you saved the JUnit library...



...select it in the dialog and press the ***Select jar files*** button.

The JUnit library should now appear under ***Referenced Libraries***.



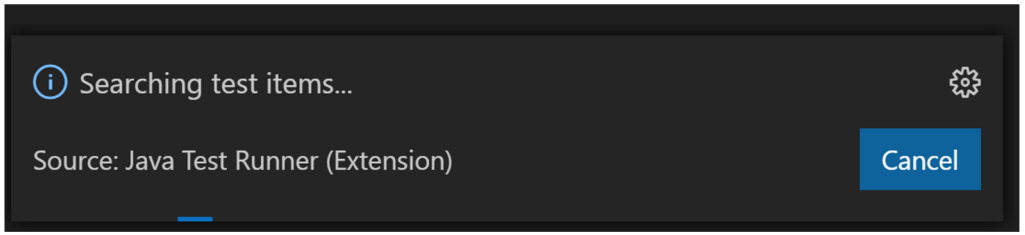
1.2.8

JUnit 5

Bootstrapping JUnit Tests

VS Code's test runner () will only work if at least one class in the project has been marked with the @Testable annotation. Let's create an empty test and use the test runner to find it.

- Create a new JUnit test in a file named "`CalculonTest.java`".
  - Add the `@Testable` annotation to the class so that VS Code's Java Test Runner can find it when you run all tests.
  - ***Hint***: just type the first few characters, e.g. "`@Testa`", and use VS Code's autocomplete feature to import the annotation.
- Click the ***flask icon*** in your VS Code sidebar to open the ***Java Test Runner***.
  - Click the ***run all tests button***  at the top of the test runner.
  - If everything works correctly, the dialog below should appear briefly. You should not see any other output (because we don't yet have any tests to run).




```java
@Testable
public class ExampleTest {

}
```

The VS Code ***Java Test Runner*** looks for classes marked with the `@Testable` annotation.

If the annotation is not present, then the test runner may not find your test when running all tests.

1.2.9


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

  // invoke
  int actual = x + 1;

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

A First JUnit Test

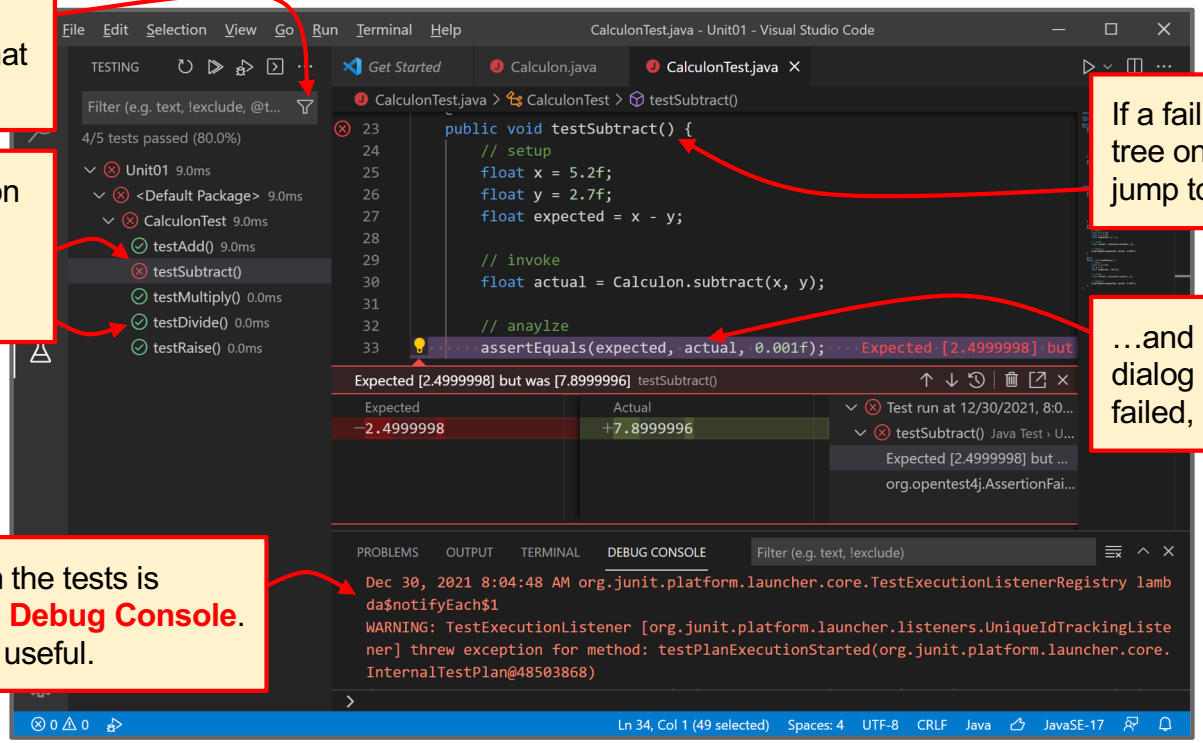
Now that we have an empty unit test, let's write our first test method. Tests in Java should be set up the same as they are in Python (setup, invoke, analyze) and should be small and fast.

- Open the `CalculonTest` JUnit test 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 VS Code's autocomplete feature to import the `assertEquals` method from JUnit.
- Click the ***flask icon*** in your VS Code sidebar to open the ***Java Test Runner***.
  - Click the ***run all tests button***  at the top of the test runner.
  - You will see that VS Code 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  .

Test Output in VS Code

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**.



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.

1.2.10

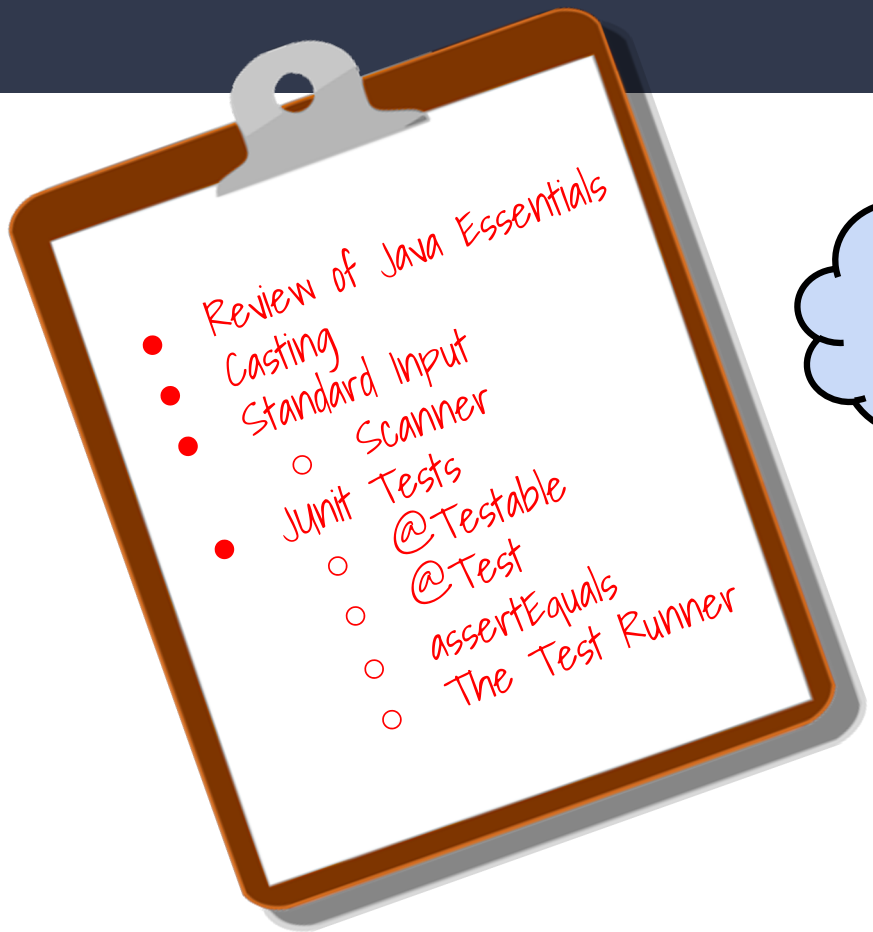
More JUnit Tests

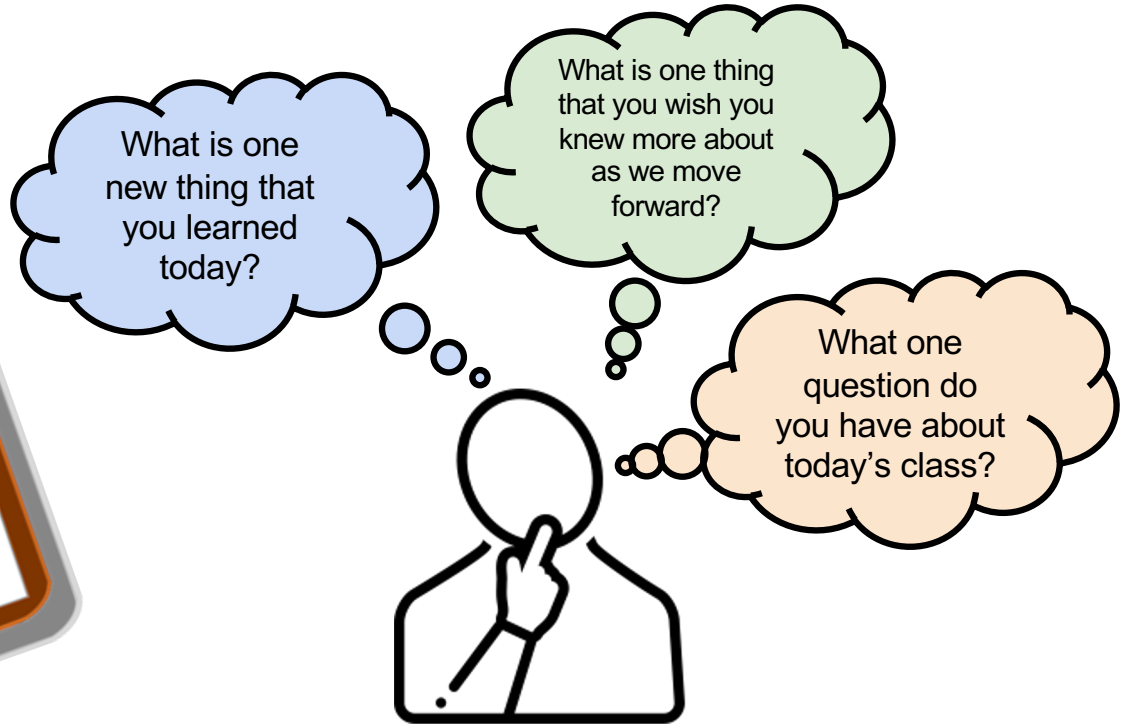
We will be writing lots of JUnit tests this semester to verify that our code is working properly. Let's get some more practice by writing tests for some of the other functions for the `Calculon` five-function calculator. We will want at least one test for each function.

```
@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.

Summary & Reflection





Please answer the questions above in your notes for today.