SWEN 601
Software Construction

Unit Testing & Incremental Development

Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
    a. *The project may already contain some code!*
2. Create a package `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

Do not submit code that ***does not compile***. Comment it out if necessary.

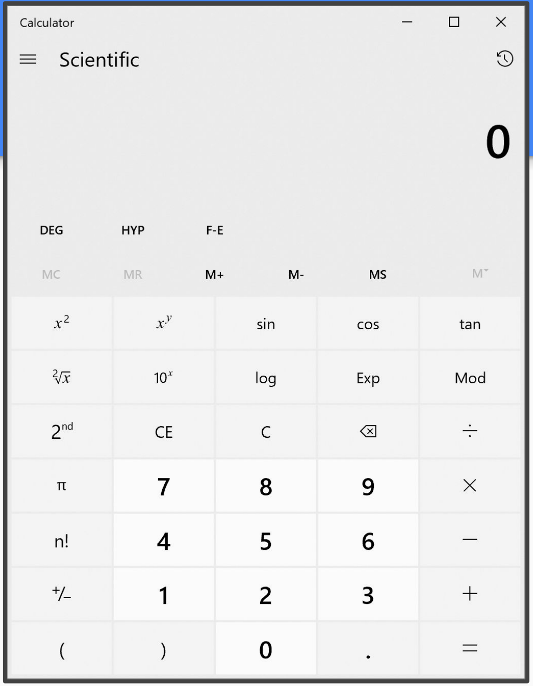
Next Two Weeks

WEEK 05	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #8		Quiz #9		
LECTURE			Interfaces & Abstract Classes		Unit Testing & Incremental Development		
HOMEWORK	Hwk 8 Due (**11:30pm**)		*Hwk 9 Assigned*		*Hwk 10 Assigned*	Hwk 9 Due (**11:30pm**)	

WEEK 04	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #10		Quiz #10		
LECTURE			Recursion & Binary Search		Sorts & Complexity		
HOMEWORK	Hwk 10 Due (**11:30pm**)		*Hwk 11 Assigned*		*Hwk 12 Assigned*	Hwk 11 Due (**11:30pm**)	

Problem: A Simple Calculator

- An *operation* applies an *operator* to some number of *operands* and returns the result.
  - For example, the *sum* operation applies the addition operator to each operand and returns the total.
  - Therefore the *sum* of the operands {1, 2, 3, 4} would be 10 (because 1 + 2 + 3 + 4 = 10).
- A *calculator* comprises zero or more operations.
  - When given an operator and an array of operands, the calculator performs the operation.



Today we are going to write a calculator program using *test driven development* (TDD).

Incremental Development

- ***Incremental Development*** refers to the practice of developing a large program in very small increments.
  - Write a small amount of code.
  - Verify that it is working.
  - Refactor if needed.
  - Repeat.
- This kind of development is an alternative to the way that many novice programmers code:
  - Write large parts of the software.
  - Test it in chunks (maybe) using a main method.
  - Delete the test code once it demonstrates that the code works.

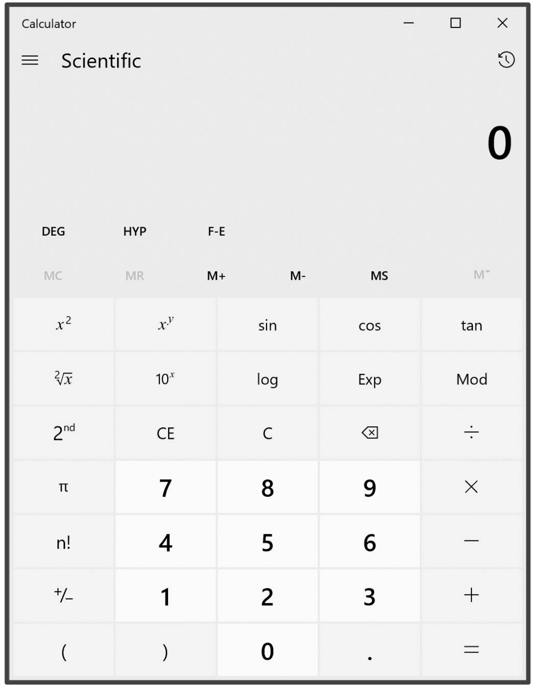
Today, we will use an incremental approach to implement our calculator.

We will start by implementing a single operation, and verifying that each method on the operation works.

We will repeat this for several operations, ensuring that each works individually.

Then we will implement the calculator, and make sure that it works with zero or more of our operations.

Activity: An Operation Interface

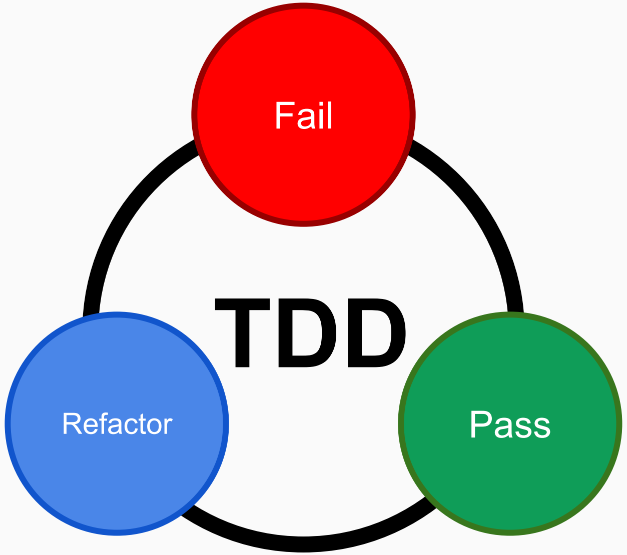


As mentioned in the problem description, an *operation* applies an *operator* to a series of zero or more *operands*. Write an `interface` to represent an operation:

- It should include a method to determine whether or not it `matches` a specific operator, e.g. "+" or "*". The method should accept the operator as a `String` parameter and return `true` if it is the correct operator.
- It should include a method that, given an `double` array of operands, performs the `operation` and returns the result as a `double` value.

Testing

- So far this semester, we have encouraged you to write your code, and then test it.
    - This kind of development is called ***test last development***; you write new code ***first***, and then write the tests ***last***.
- Today we are going to talk about ***test driven development*** (***TDD***).
    - You write the test ***first***, and then write the code ***last*** to make the test pass.
    - ***TDD*** is one kind of incremental development.
    - Many professional developers write code this way.



Test vs. Production

- ***Production Code*** is the “real code” that you are writing to solve your problem, e.g. your solution for a homework assignment.
- ***Test Code*** is the code that makes sure your production code is working properly.
- By creating a separate folder for test source, we can keep the two separated.
  - This makes it easier to deliver one without the other.
- Separating tests from production code is a common practice.
  - Even though your tests are in a different folder, they will be in the same package as your production code, and so will have access to `protected` or *package-private* fields and methods.

Keeping test code separate from production code has lots of advantages.

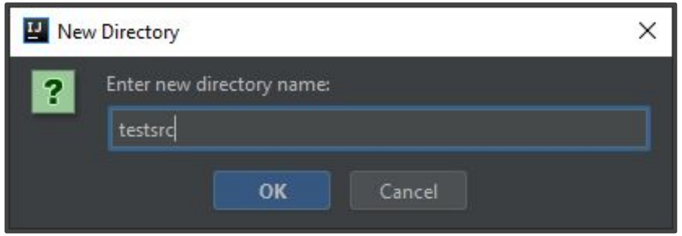
Not only can you deliver your production code without cluttering it with your tests...

...but you can run your test code whenever you’d like to without having to change your `main` function.

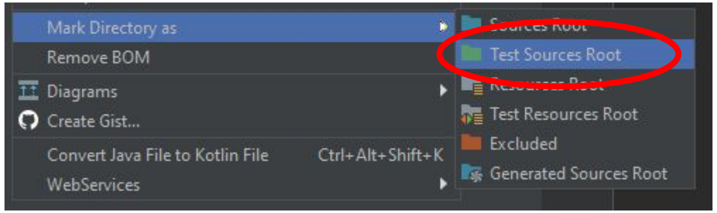
Activity: Create a Test Sources Root

You will want to keep your ***production code***, the software that you intend to deliver to your customer, separated from your ***test code***, the code that validates that your production code is working.

1. Right-click on your project folder and create a new Directory. Name it "`testsrc`".



2. Right-click on the folder and choose *mark Directory As → Test Sources Root.*



Your ***production code*** will go into the `src` folder, and your ***test code*** will go into the `testsrc` folder.

This will keep your tests separate from your production code. Your releases will contain both (for this class).

JUnit

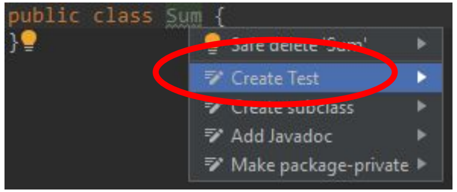
- A ***testing framework*** is a set of tools designed to make testing your code easier.
  - A good testing framework will support ***test driven development*** by making the creation and execution of tests fast and easy.
- This semester, we will be using the JUnit 5 testing framework, support for which is integrated into IntelliJ IDEA.
- JUnit supports the creation of ***unit tests***.
  - You will typically write one unit test per class in your program.
  - A unit test includes one or more tests for each non-trivial method in the class (e.g. you usually would not write unit test for accessors or mutators).

Before we can start using ***test driven development***, we need to first create a class and its empty unit test.

Let's do that now...

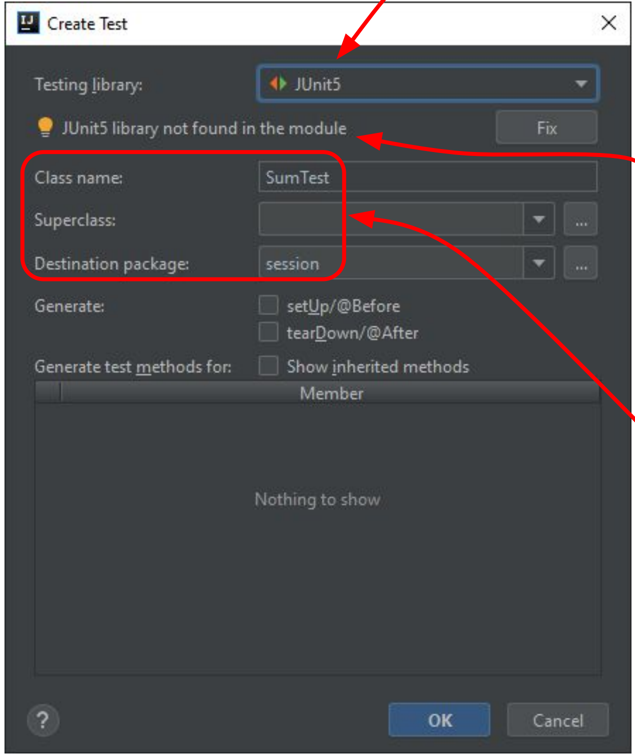
Activity: Creating Your First Unit Test

Begin by creating a new class for the Sum operation. ***Do not add any code*** to the class.



```
public class Sum {
}
```

Next, click your cursor in the class name and press Alt-Enter to open the context menu. Choose the *Create Test* option.



Create Test

Testing library: JUnit5

JUnit5 library not found in the module

Class name: SumTest

Superclass:

Destination package: session

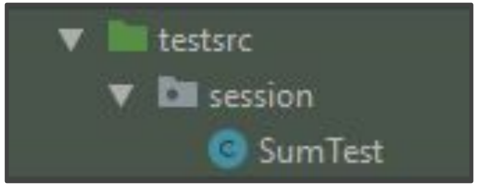
Generate: setUp/@Before, tearDown/@After

Generate test methods for: Show inherited methods

In the *Create Test* dialog, choose JUnit5 from the drop-down menu.

The first time you create a test, you will see this warning because JUnit5 is not yet a part of your project. Click *Fix* and *OK* in the dialog that opens.

Notice that the dialog already has values for the test name and package that match your class. Accept these by clicking *OK*.



Unit Tests

- A ***unit test*** contains all of the tests for a single class.
- Each test is implemented as a separate method in the class.
  - To start, your test will be empty.
  - We will change this soon.
- A good test method has two characteristics:
  - ***small*** - it tests only one thing (i.e. one method of your class).
  - ***fast*** - it should run very quickly (i.e. in a fraction of a second).

```
package session;

import static org.junit.jupiter.api.Assertions.*;

class SumTest {

}
```

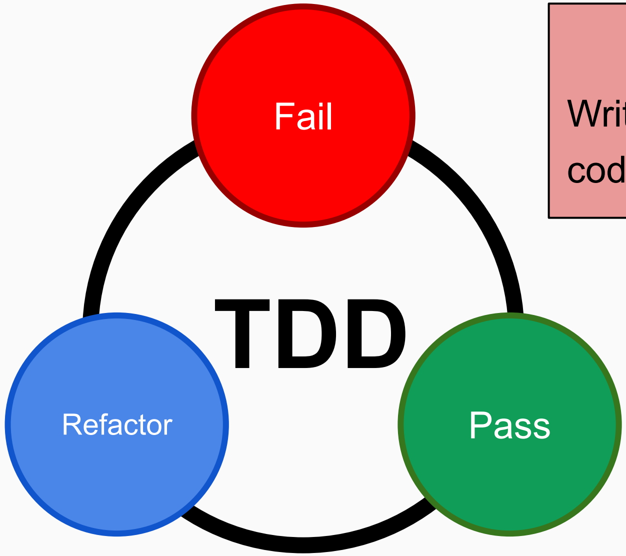
IntelliJ IDEA has done a lot to help us get started, but before we start writing tests, let’s talk a bit more about ***test driven development***.

Test Driven Development (TDD)

3. REFACTOR

Refactor the code to eliminate redundancy/clean up. Run your tests again to make sure that you didn't break anything!

Once you verify that your refactored code is still working (because ***all*** of the tests pass), you push it to your SCM server!



1. FAIL

Write a test that ***fails*** (because the code doesn't exist yet).

2. PASS

Write the *minimum* code to make the test ***pass***.

Step 1: FAIL

- The first step of TDD is to write a test that ***fails***.
- Every good unit test will have three basic parts:
  - ***Setup*** - create any variables needed for the test, including an instance of the class under test.
  - ***Invoke*** - invoke the method that is being tested using values created during setup.
  - ***Assert*** - validate that the method worked as intended and returned the correct result.

Before we can even run the test, it must compile, but as written, there will be a number of compilation problems with this test.

```
class SumTest {
  @Test
  void operationSingle() {
    // setup
    Operation sum = new Sum();
    double[] input = { 1.0d };
    double expected = 1.0d;

    // invoke
    double result = sum.operation(input);

    // assert
    assertEquals(expected, result);
  }
}
```

We will use IntelliJ IDEA’s code generation to quickly fix the problems so that we can run the test.

Anatomy of a Unit Test

The `@Test` annotation is used by JUnit to determine which methods are tests.

When a unit test is executed, JUnit will execute any method marked as such.

The ***invoke*** part of the test runs the code that is being tested. Often this is just a single method call.

```
class SumTest {
  @Test
  void operationSingle() {
    // setup
    Operation sum = new Sum();
    double[] input = { 1.0d };
    double expected = 1.0d;

    // invoke
    double result = sum.operation(input);

    // assert
    assertEquals(expected, result);
  }
}
```

Each test should have a descriptive name (and should include the name of the method that it is testing).

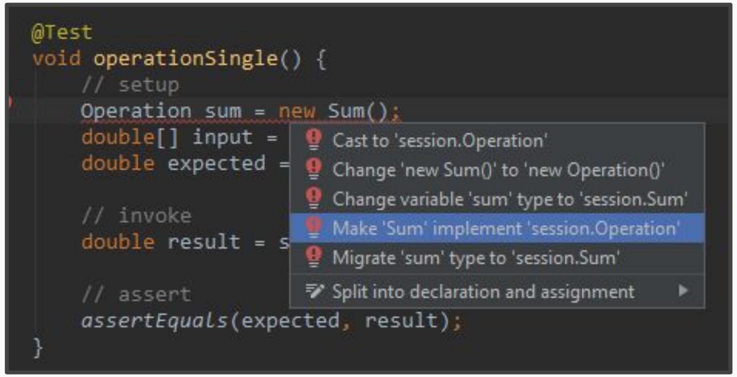
The ***setup*** part of the test creates any values needed for the test, including the object under test.

The ***assert*** part of the test verifies that the expected results were achieved, e.g. the correct value is returned.

JUnit provides many built-in assertion functions to validate results.

Fixing Simple Compilation Problems

- Sure, the goal is to write a *failing test*, but it doesn't help if the program won't compile at all.
- In this case, there are several problems with the code:
  - `Sum` does not implement `Operation`.
  - There is no `operation()` method to call on `Sum`.
- We can fix this by using IntelliJ IDEA to quickly generate code to fix each problem.
- This is a common practice in ***test driven development*** and can save experienced developers a lot of time.



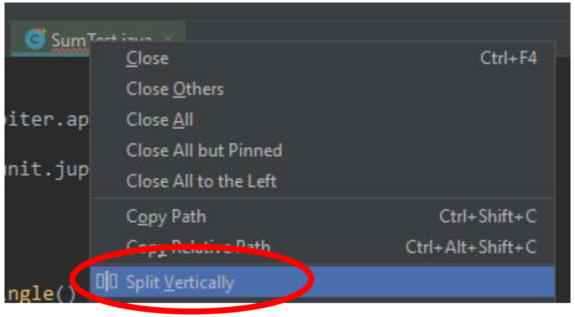
```
@Test
void operationSingle() {
    // setup
    Operation sum = new Sum();
    double[] input =
    double expected =

    // invoke
    double result = s

    // assert
    assertEquals(expected, result);
}
```

Cast to 'session.Operation'
Change 'new Sum()' to 'new Operation()'
Change variable 'sum' type to 'session.Sum'
Make 'Sum' implement 'session.Operation'
Migrate 'sum' type to 'session.Sum'
Split into declaration and assignment

We will fix the errors one at a time using IntelliJ IDEA's nearly *magical* Alt-Enter key and the context menu.

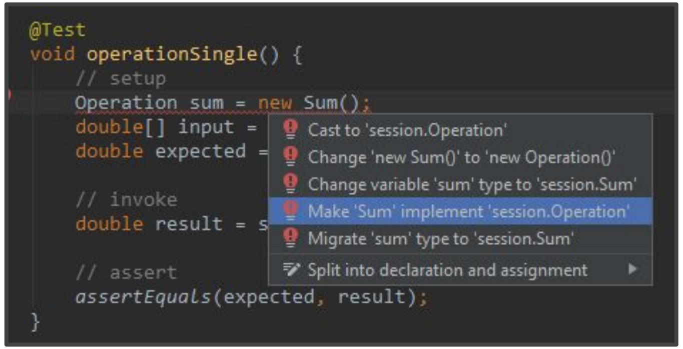
Activity: Fixing Errors With Code Generation

It is ***very*** helpful to view your test and production code side-by-side in your IDE.



Right-click on the tab in the editor and choose *Split Vertically* from the popup menu.

With the test and class under test open side-by-side, click your cursor into the first error and press Alt-Enter to open the context menu.

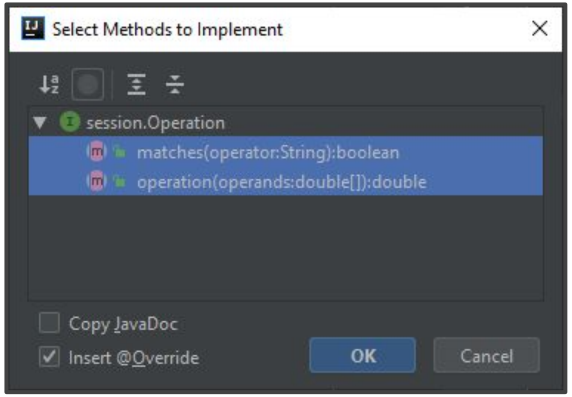


```
@Test
void operationSingle() {
    // setup
    Operation sum = new Sum();
    double[] input =
    double expected =

    // invoke
    double result = s

    // assert
    assertEquals(expected, result);
}
```

Cast to 'session.Operation'
Change 'new Sum()' to 'new Operation()'
Change variable 'sum' type to 'session.Sum'
Make 'Sum' implement 'session.Operation'
Migrate 'sum' type to 'session.Sum'
Split into declaration and assignment

Choose the option to *Make 'Sum' implement 'session.Operation'* using the arrow keys and press Enter.

In the dialog, make sure that all of the `Operation` methods are selected and press Enter.



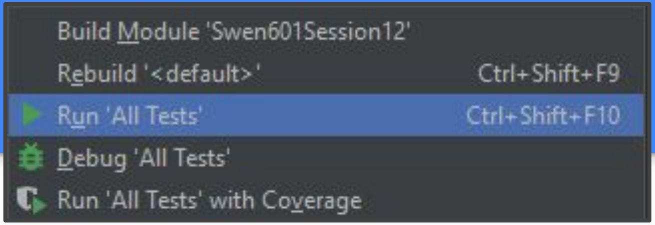
Select Methods to Implement
session.Operation
matches(operator:String):boolean
operation(operands:double[]):double
Copy JavaDoc
Insert @Override
OK
Cancel

Stubbed methods will appear automatically in the Sum class!

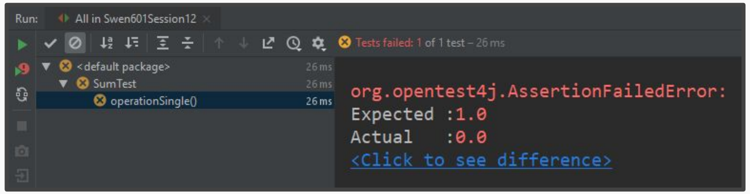
Running a Test

- Now that the test is compiling, we can get back to ***failing***.
- There are several ways to run a test.
  - Run the individual test method.
  - Run the entire unit test (i.e. all of the methods marked with the @Test annotation in the entire class).
  - Run all of your tests (i.e. every unit test in the test sources root).
- It is good to get into the habit of running all of your tests every time.

Right click on your test sources root and choose the *Run 'All Tests'* option from the popup menu.


Build Module 'Swen601Session12'
Rebuild '<default>' Ctrl+Shift+F9
Run 'All Tests' Ctrl+Shift+F10
Debug 'All Tests'
Run 'All Tests' with Coverage

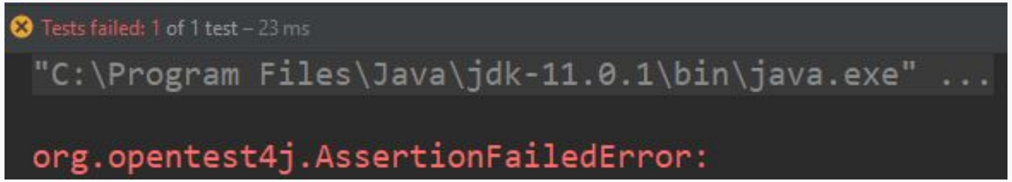
At the bottom of the window, you will see lots of feedback that the test failed.


Run: All in Swen601Session12
Tests failed: 1 of 1 test – 26 ms
<default package> 26 ms
SumTest 26 ms
operationSingle() 26 ms

```
org.opentest4j.AssertionFailedError:
Expected :1.0
Actual   :0.0
<Click to see difference>
```

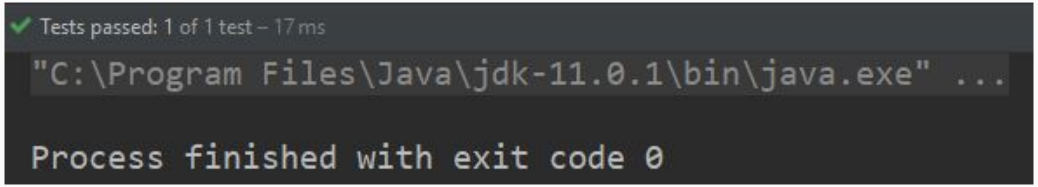
When a test fails, JUnit gives you a lot of feedback in **red**. For this reason we refer to the ***fail*** step as *"going red."*

Step 2: Pass



```
Tests failed: 1 of 1 test – 23 ms
"C:\Program Files\Java\jdk-11.0.1\bin\java.exe" ...

org.opentest4j.AssertionFailedError:
```

- Now that the test is running (and failing), the next step is to write the ***minimum code*** to make the test ***pass***.
- Then, rerun the test to make sure that it passes.



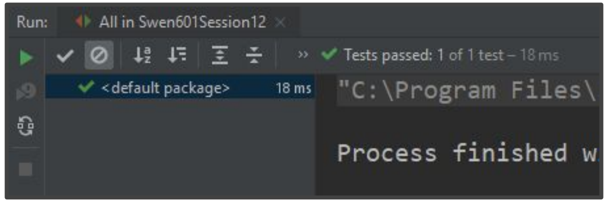
```
Tests passed: 1 of 1 test – 17 ms
"C:\Program Files\Java\jdk-11.0.1\bin\java.exe" ...

Process finished with exit code 0
```

The *assertion failed error* tells us what went wrong.

```
org.opentest4j.AssertionFailedError:
Expected :1.0
Actual   :0.0
```

The test expected the `operation()` method to return `1`, but it returned `0`.

The next step is to write the *minimal* code to get the test to pass.



Run: All in Swen601Session12

Tests passed: 1 of 1 test – 18 ms

<default package> 18 ms

```
"C:\Program Files\

Process finished w
```

When all of your tests pass, you will see lots of green feedback from JUnit.

For this reason, we often refer to the ***pass*** step in TDD as *"going green."*

Activity: Pass

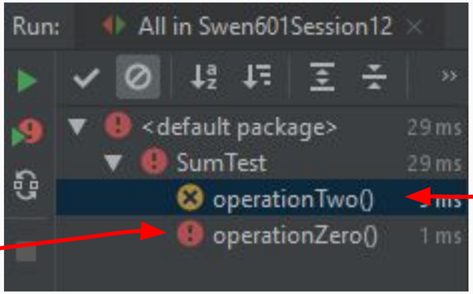
The `SumTest` unit test is currently failing because the `operation()` method is not returning the expected result.

1. Fix your `operation()` method by writing the *minimal code* to handle an array with a single value.
   a. *Hint: just return the first value in the array.*
2. Then rerun your test to see that it passes.
   a. *Hint: Shift-F10 will rerun the last thing that you executed.*
3. Once your test is passing, commit & push your code to GitHub.
   a. You will do this ***every time*** you make a test pass.

TDD & Incremental Development

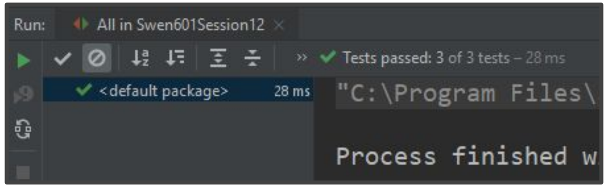
- ***Incremental development*** refers to building up a large software application in very small increments.
- Test Driven Development is one approach to incremental development.
  - Write one ***small*** test.
  - Write the ***minimal*** code to make the test pass.
  - ***Repeat*** until the application is finished.
- Let's incrementally develop the `Sum` class by writing a few more tests.

Some of your tests will fail with *assertion errors*. These will be **yellow** in your list of tests.



Others may fail with *exceptions* (that would normally crash your program). These will be **red**.

Activity: Fail, Then Pass



```
Tests passed: 3 of 3 tests – 28 ms
"C:\Program Files\

Process finished w
```

If you write a new test and you don't need to write any code to make it pass, you should consider whether or not you need that test.

For example, if you have a test for an array of 2 input values, do you need to test 3? Or 4?

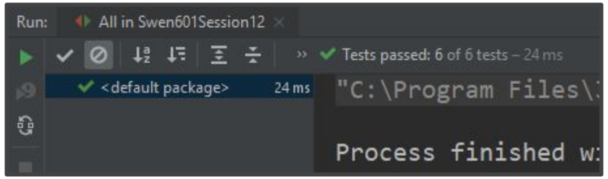
Take a moment to think of another, simple test case that your `operation()` method should handle.

1. Write a failing test (either because the wrong result is returned, or an exception is generated).
2. Write the minimal code to make the test pass.
3. Commit and push your code.
4. Repeat this until you run out of things to test.
   a. *Hint: This will be when you don't have any more code left to write. You should have written about three tests.*

Activity: Fail, Then Pass *Again*

So far, we have only tested the `operation()` method on the `Sum` class. What about the `matches()` method?! It should return `true` for the "+" operator, and `false` for any other string.

1. Repeat the TDD process for the `matches()` method.
   a. Write a failing test to test one possible value (e.g. the "+" operator). *Hint: How do you compare two strings?*
   b. Write the *minimal* code to get the test to pass.
   c. ***Don't forget to <u>commit and push</u>.***
2. Stop when you think that you are done.



```
"C:\Program Files\
Process finished w
```

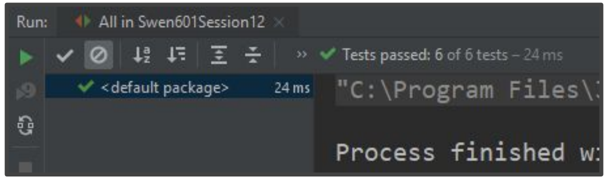
You should have a total of about 6 tests when you are finished.

If you have fewer than that, it is possible that there is something that you did not consider. Ask!

Activity: Difference

Now we are going to repeat the entire process to implement a `Difference` operation.

- It should match the "-" operator.
- It should subtract all of the input values in order
  - e.g. {10, 3, 2} would return 5 because 10 - 3 - 2 = 5.



```
"C:\Program Files\

Process finished w
```

As you practice TDD more and more, you will get better and faster at the process of creating and writing tests.

Make sure to always run every test to make sure that your new code didn't break anything!

1. Begin by creating the new class in your `src` folder named `Difference`.
2. Then create the empty unit test.
3. Then write your first failing test!
   a. Use the `SumTest` unit test as a reference.

Step 3: REFACTOR

- Up to this point we have skipped the third TDD step: ***refactor***.
  - This step is *optional* and you only need to consider it if your code needs to be cleaned up.
- Refactoring refers to cleaning up your code to improve code quality.
  - Often by eliminating unnecessarily duplicated code.
  - This includes your test code as well as your production code.

You should notice that the `matches()` methods in your `Sum` and `Difference` classes are ***very*** similar...

```
@Override
public boolean matches(String operator) {
      return operator != null &&
         operator.equals("+");
}
```

```
@Override
public boolean matches(String operator) {
      return operator != null &&
         operator.equals("1");
}
```

...but the `operation()` methods are different. What should we use if we can implement ***some*** of the behavior in an interface, but not ***all*** of it?

Safely Refactoring

- Refactoring refers to cleaning up your code to improve code quality.
- Refactoring *does not change* the behavior of the code.
  - That means that, given the same input, your algorithms will produce the same output before and after the refactoring.
- There are many advantages to TDD, but one of the biggest is that it gives you a ***test harness***.
  - This means that it is safe to refactor your code.
  - Make small changes, and rerun your tests.
  - If they pass, great!
  - If they break, roll back the changes.

Activity: Abstract Operation

The next step will be to ***refactor*** one of your operations to extend the abstract class.

You can do this safely by re-running your tests to make sure that nothing broke.

It looks like our setup could benefit from an `abstract` class that implements `Operation` and the `matches()` method. Thanks to the ***test harness*** that we have written, it is safe to refactor the code to use a new class.

- This will avoid having to write similar matches() implementations in all of our future operations.
- Because the class is `abstract`, there won't be a need to implement the `operation()` method. That will be left up to the child classes.

1. Begin by creating a new `abstract` class named `AbstractOperation` that `implements` `Operation`.
2. Add a field for the operator and a constructor to set the value of the field.
3. Implement the matches() method to match the field.
4. Commit and push your code.

Activity: Refactor Sum

It is time to safely refactor the Sum class.

1. Modify the class so that it extends `AbstractOperation`.
   a. This will cause a compilation problem that you will need to fix.
2. Before doing anything else, re-run your tests to make sure that they pass.
3. Commit and push your code.
4. Now, delete the no-longer-needed `matches()` method from the class.
5. Re-run your tests to make sure that they pass.
6. Commit and push your code.

By re-running your tests each time you make a change to the code, you verify that your recent changes did not break what was already working.

This is the safety provided by your ***test harness***.

Reasons not to Test

- You think writing tests takes too long.
  - How much time do you spend trying to figure out why your code isn't working?
- You are lazy.
  - Testing actually significantly reduces development time, so...
- You like losing points on labs/homework.
- You never refactor, so you don't need to do it safely.
- You don't want to impress interviewers with your TDD experience.
  - Unit testing is industry standard.