Tutorial

Test-Driven Development

As you learned from the pre-reading, test-driven development (TDD) is a development process that follows a specific sequence of steps designed to ensure that production code is thoroughly covered by unit tests and that the resulting production code is modular, flexible, and extensible. In this lab you will create some simple methods that make up part of a Calculator class. The functionality of the calculator class is simple. The purpose of this lab is to teach you the TDD process by using it to write something that is easy to understand, allowing you to focus on learning the TDD process.

Setup

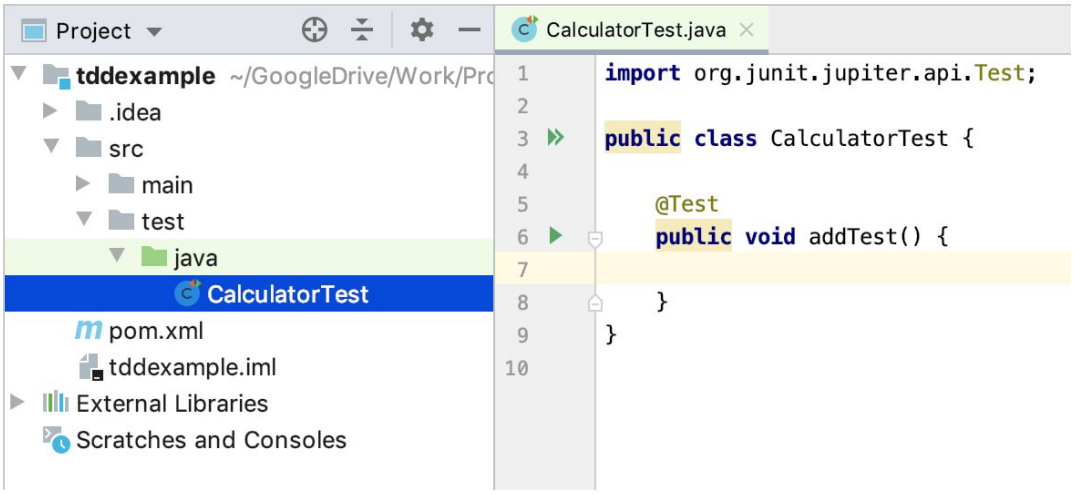
1. Create a new Maven or Gradle project and name it tdd-example.
    a. Use whatever group Id you want.
2. We will be using JUnit 5 for this tutorial. Add the dependency for the latest production version of **junit-jupiter-api** to your Maven **pom.xml** or Gradle **build.gradle** file. Refer to the dependency management tools tutorial if you need help with this step.

Tutorial

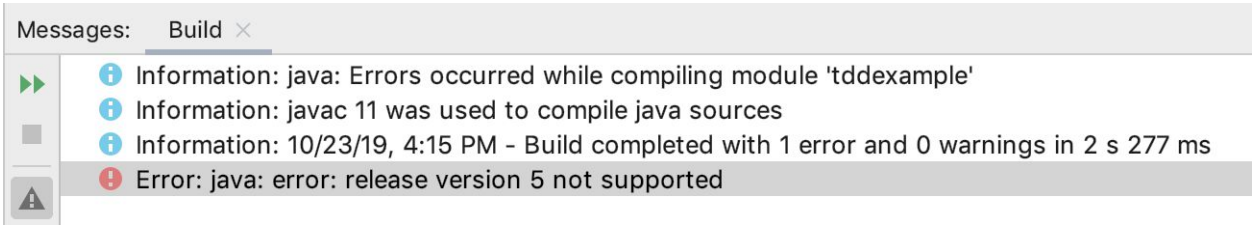
1. We will be creating a partially functional, simple Calculator in this lab while strictly following the TDD process. The steps will seem tedious for such a simple example, but they are required to ensure we are getting the benefits of TDD. The TDD process requires that we always have a failing test that can only pass by writing production (non-test) code before we write production code. TDD also requires that we always write the simplest code that will allow the test to pass and that we only make it more complex when required to do so by a failing test. This ensures that our code is fully covered by tests.

2. Start by creating a CalculatorTest class in your ‘src/test/java’ folder.

3. Add an empty addTest method annotated with the @Test annotation as shown below:



4. Run the test to see that it passes. You may see the following error:



If you do, fix it by following the instructions for the following two issues (in this order) in the "Setup and Test Running Issues in Intellij" document.

1. Intellij Reverts to An Earlier Java Version After you Specify a Latter One
2. Error:java: error: release version 5 not supported

Do not continue until the test is passing.

5. With TDD, we do not write production code until a test requires it in order to pass. Since our only test is passing, we will continue to write test code. Add code to the addTest() method to create an instance of the Calculator class (which doesn't exist yet, so the line of code you write won't compile). Your method should look like this:

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

public class CalculatorTest {

    @Test
    public void addTest() {
        Calculator calculator = new Calculator();
    }
}
```

Having a test method that does not compile counts as having a failing test method. It can't pass until we write production code so now we can write the minimum production code that will allow the test to pass.

6. Create a Calculator class in the 'src/main/java' directory.

```
public class Calculator {
}
```

7. Now your test will pass again. Confirm this by running it and do not proceed until you see it pass.

8. There's nothing to refactor (the next step in the TDD process), so we continue by writing test code. Add the following line to the end of your addTest() method:
`Assertions.assertEquals(5, calculator.add(2, 3));`

9. Now we have a failing test that can only be made to pass by writing production code in the Calculator class. We write the simplest code that will allow the test to pass and only make it more complex when required by a test, so add an 'add' method that takes two int parameters and simply returns 5 as follows:

```
public class Calculator {
    public int add(int x, int y) {
        return 5;
    }
}
```

It may seem silly to write such a simple implementation of the method that clearly isn't correct, but it is the simplest code that will allow the test to pass. By strictly following this rule, we end up with a complete set of tests that test every code path and will eventually require the code to be correct in order to pass. In other words, our tests drive the implementation, which is why the process is called Test-Driven Development.

Confirm that your test is once again passing before continuing.

10. Since our code clearly isn't correct, we need to add more test code that will require it to be correct or at least require it to be more correct. When the new test code is failing, and cannot be made to pass without improving our production code, we will improve it.

    Add the following statement to the end of your addTest() method and confirm that the test fails when you run it: `Assertions.assertEquals(9, calculator.add(4, 5));`

11. Now the add() method in our Calculator class needs to return 5 when it gets parameters of 2 and 3 and it needs to return 9 when it gets parameters of 4 and 5. There are several ways we can make this happen, but the simplest code is finally the correct code. Replace the line that returns 5 to return the result of adding x and y as follows, and confirm that the test now passes:

```
public class Calculator {
    public int add(int x, int y) {
        return x + y;
    }
}
```

12. Our only test is now passing, and we don't need to add any more tests for the add() method. We also don't need to refactor the code in Calculator or CalculatorTest. They are both about as clean and free of code duplication as they can be, so we are done with the add() method. However, a calculator that can only add isn't very useful. Let's make it subtract, starting with adding a failing subtract test in Calculator. We keep writing test code until it cannot pass without writing production code, so we end up with something like this:

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

public class CalculatorTest {

    @Test
    public void addTest() {
        Calculator calculator = new Calculator();
        Assertions.assertEquals( expected: 5, calculator.add( x: 2,  y: 3));
        Assertions.assertEquals( expected: 9, calculator.add( x: 4,  y: 5));
    }

    @Test
    public void subtractTest() {
        Calculator calculator = new Calculator();
        Assertions.assertEquals( expected: 3, calculator.subtract(7, 4));
    }
}
```

13. As before, write the simplest production code that will allow the test to pass. This will be a subtract method in the Calculator class with a hard-coded return value of 3. **Confirm that all tests are passing before continuing.**

14. The next step is 'refactor'. We do have something to refactor because both of our test methods have the same first line of code. This is code duplication and code duplication is bad. Eliminate the duplication by adding a setup method annotated with

@BeforeEach. The setup method should initialize a ‘calculator’ instance variable and the ‘calculator’ variables should be removed from the two tests as follows:

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

public class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    public void setup() {
        calculator = new Calculator();
    }

    @Test
    public void addTest() {
        Assertions.assertEquals( expected: 5, calculator.add( x: 2,  y: 3));
        Assertions.assertEquals( expected: 9, calculator.add( x: 4,  y: 5));
    }

    @Test
    public void subtractTest() {
        Assertions.assertEquals( expected: 3, calculator.subtract( x: 7,  y: 4));
    }
}
```

15. Run the tests to ensure that they still pass after the refactor.

16. Now that our tests are all passing and our tests and production code are as clean as we can make them by refactoring, add the following assert to the subtractTest() method to force an improvement to the subtract code: `Assertions.assertEquals(12, calculator.subtract(20, 8));`

    Confirm that your subtractTest() method is failing before continuing.

17. Write the simplest code that will allow all tests to pass. As with the add() method, this will be the correct code that causes the subtract() method to properly handle subtraction.

18. Now we will write a powerTest() method as the first step to adding a power() method to our Calculator class. As before, write the powerTest() method until you reach a point where the test is failing and can only be made to pass by writing production code in the Calculator class. You should end up with a method that looks like the following:

```
@Test
public void powerTest() {
    Assertions.assertEquals( expected: 16, calculator.power(4, 2));
}
```

19. The test doesn't compile, so it is failing. Now write a power() method in the Calculator class with a hard-coded value and confirm that the tests are all passing again.

20. As before, write an additional assert in the powerTest() method. Make the power value something higher than 2 as follows: `Assertions.assertEquals(27, calculator.power(3, 3));`

    Confirm that the test fails.

21. Now replace the hard-coded return value in the Calculator class' power() method with the simplest code that will allow both assertions to pass. You should end up with a power() method that looks something like this:

```
public int power(int x, int y) {
    int result = 1;

    for (int i = 0; i < y; i++) {
        result *= x;
    }

    return result;
}
```

22. This seems right for positive methods and powers, but what if we raised a value to the 0 power? That should produce a result of 1. Write a test that raises a value to the 0 power. We could add this to our existing powerTest() method, but it seems like we'll be testing something different about the power() method. Each test method should be simple enough that it would only ever fail for one reason, we should write this test as a separate test method, leaving us with two power tests as follows:

```
@Test
public void powerTest() {
    Assertions.assertEquals( expected: 16, calculator.power( x: 4,  y: 2));
    Assertions.assertEquals( expected: 27, calculator.power( x: 3,  y: 3));
}

@Test
public void powerTest_positiveValueZeroPower() {
    Assertions.assertEquals( expected: 1, calculator.power( x: 3,  y: 0));
}
```

23. Run the tests and you will see that the new test passes without changing the code in the Calculator class. The simplest code for the previous tests also works for 0 power cases. This is still a good test case to have though, so we will keep it.

24. This test does bring up a refactoring issue though. It seemed like a good idea to have a test named “powerTest” when there was only one test for the power() method. Now that there is more than one, we should rename powerTest() to something more clear about what it does and does not test. Rename this test to powerTest_positiveValuePositivePower()[1].

25. There clearly are more combinations of positive and negative values and positive and negative powers that we should test. Now that we’ve thought of it, let’s list the remaining combinations and make sure our power() method handles them all properly. We’ve already handled positive value positive power and positive value zero power. Here are the remaining combinations: **Note:** Don’t create these tests yet, we will be creating the rest of these tests throughout the tutorial.
    a. Positive Value Negative Power
    b. Negative Value Positive Power
    c. Negative Value Zero Power
    d. Negative Value Negative Power
    e. Zero Value Positive Power
    f. Zero Value Zero Power
    g. Zero Value Negative Power

26. Add a powerTest_positiveValueNegativePower() test using the values 5 and -2 as follows:

```
@Test
public void powerTest_positiveValueNegativePower() {
    Assertions.assertEquals( expected: 0.04, calculator.power( x: 5,  y: -2));
}
```

[1] The names used for the tests for the power() method follow a naming convention suggested by Roy Osherove in his book “The Art of Unit Testing”.

Confirm that the test fails.

27. This test reveals two problems with our power() method. First it will have to return a floating point result instead of an int to handle negative power cases, and second, it doesn't seem to be doing the right calculation for negative power cases. We'll handle the first problem first. We have a test failing that requires a change in the production code, so change the return type of the power() method to double.

28. Run all of your tests. This won't have fixed the test that was failing before, but it's closer to passing and we want to confirm that we didn't break any tests that were passing before. We should still have four tests passing and one failing.

29. Fix the failing test by writing the simplest code that allows it to pass as follows:

```
public double power(int x, int y) {
    if(y < 0) {
        return 0.04;
    }

    int result = 1;

    for (int i = 0; i < y; i++) {
        result *= x;
    }

    return result;
}
```

30. Obviously this is not correct, so we need to improve our test. Add the following assertion to the powerTest_positiveValueNegativePower() test: `Assertions.assertEquals(0.0625, calculator.power(4, -2));`

Confirm that the test fails.

31. The simplest code that will allow both assertions to pass is probably the correct code now. A negative power (exponent) can be converted to a positive one if we take the inverse of the value being raised to that power. To make this change while still using the for loop to do the power calculation, we will need to convert 'result' to a double and do the calculation on the inverse of x when y is negative. The following changes will fix the code, but then we will have to account for a floating point calculation error in the test. We'll handle the floating point error in the test in the next step. Compare this code to the previous power() method code to see what was changed:

```
public double power(int x, int y) {
    double value = x;

    if(y < 0) {
        value = 1.0 / x;
        y = y * -1;
    }

    double result = 1.0;

    for (int i = 0; i < y; i++) {
        result *= value;
    }

    return result;
}
```

Note: Remember that each TDD iteration ends with a refactoring step where we clean up the code without changing what it does and then confirm that our refactoring didn't cause any test failures. This is a simple example and we are making very small changes, so our code is staying clean as we go, but we should always consider at the end of each TDD iteration whether the production or test code needs to be refactored.

32. Now we can fix the floating point comparison errors in our test. We just need to add a third "delta" parameter to our assertEquals methods in the failing test to tell the assertions that the values don't have to be exactly equal to account for floating point calculation errors. Add .001 as a third parameter to the two asserts in the powerTest_positiveValueNegativePower() method and confirm that all tests are now passing.

33. Now write a powerTest_negativeValuePositivePower() test with the following assertion: Assertions.assertEquals(9, calculator.power(-3, 2)). That should work without any changes to the production code. Run the test to confirm.

34. We should also confirm that the code works when a negative value is raised to a power that results in a negative value being returned. Add the following assertion to the previous test and confirm that the test still passes: Assertions.assertEquals(-27, calculator.power(-3, 3)).

35. Now let's add a test for the negative value, zero power case. It should already work, but it is a test we should have. Add a powerTest_negativeValueZeroPower() test with the following assertion: Assertions.*assertEquals*(1, **calculator**.power(-3, 0)), and confirm that the new test passes.

36. Now create a negative value negative power test. Remember this will produce a floating point number that may be subject to a floating point rounding error so you will need to include a small delta in the assertions. Use the following assertion: Assertions.*assertEquals*(0.04, **calculator**.power(-5, -2), .001). Run the test to confirm that the production code already handles this case correctly.

37. The above test produced a positive result, but raising a negative value to an odd negative power should produce a negative result. Add the following assertion and confirm that the power method handles this case properly as well: Assertions.assertEquals(-0.008, calculator.power(-5, -3), .001).

    Note: One of the benefits of adding these test cases that don't require production code changes is that they add to our regression test suite. These tests will ensure that future changes do not break this functionality. Changes that break previously working functionality are called regressions. A good automated test suite, and especially one produced as a result of a TDD development process, will protect against regressions.

38. We have three more tests to create. Next create a zero value, positive power test case. Your assertion should have zero as the value and any positive number as the power. Assert the expected result to be 0. This test should already produce the correct result. Confirm that by running the tests before continuing.

39. Now create a zero value, zero power test case. There is some debate in the mathematical community about what the expected value should be. Some believe the value should be 1 while others believe it is undefined. We will assume a requirement that the value is undefined, and that the power() method should throw an ArithmeticException in this case. Create an assertThrows() statement in your test case as follows:

```
@Test
public void powerTest_zeroValueZeroPower() {
    Assertions.assertThrows(ArithmeticException.class, () -> calculator.power( x: 0,  y: 0 ));
}
```

    This assertion takes two parameters. The first is the class type of the exception we expect the power() method to throw. The second is a lambda expression that invokes the power() method with the required parameters.

40. Run the test and confirm that it fails.

41. Write the simplest code that will allow this and all previous tests to pass. You should end up with something like the following (new code is highlighted in blue):

```java
public double power(int x, int y) {
    if(x == 0 && y == 0) {
        throw new ArithmeticException("Not a number");
    }

    double value = x;

    if(y < 0) {
        value = 1.0 / x;
        y = y * -1;
    }

    double result = 1.0;

    for (int i = 0; i < y; i++) {
        result *= value;
    }

    return result;
}
```

Run the tests and confirm that they all pass.

42. Now write the final test--a zero value, negative power test. This test should also throw an ArithmeticException. Use an assertThrows with a value of zero and any negative power in the lambda call of the power() method.

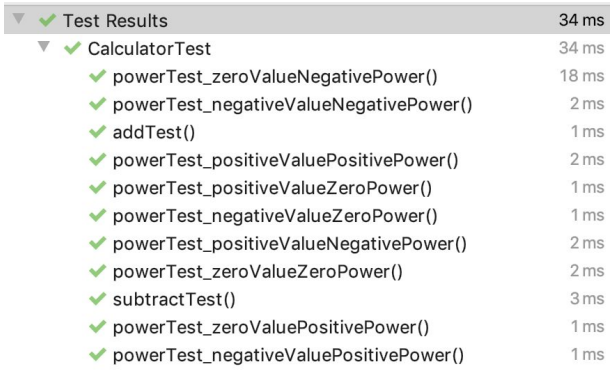
43. Run the tests and confirm that the new test fails.

44. Change the power() code to make the test pass. This is a simple 1 character change. Change the expression `y == 0` to `y <= 0` in the first `if` statement.

45. Run the tests to confirm that they all pass. The code looks clean and free of duplication so we are done with the power() method and its associated tests. If we were building a fully-functional Calculator class we would continue with the next operation, but we will stop here.

Submission

Submit your Calculator.java and CalculatorTest.java files to Canvas. Also submit a screenshot showing the results of your test run with all tests passing. Your screenshot should look something like this:


Test Results 34 ms
CalculatorTest 34 ms
powerTest_zeroValueNegativePower() 18 ms
powerTest_negativeValueNegativePower() 2 ms
addTest() 1 ms
powerTest_positiveValuePositivePower() 2 ms
powerTest_positiveValueZeroPower() 1 ms
powerTest_negativeValueZeroPower() 1 ms
powerTest_positiveValueNegativePower() 2 ms
powerTest_zeroValueZeroPower() 2 ms
subtractTest() 3 ms
powerTest_zeroValuePositivePower() 1 ms
powerTest_negativeValuePositivePower() 1 ms