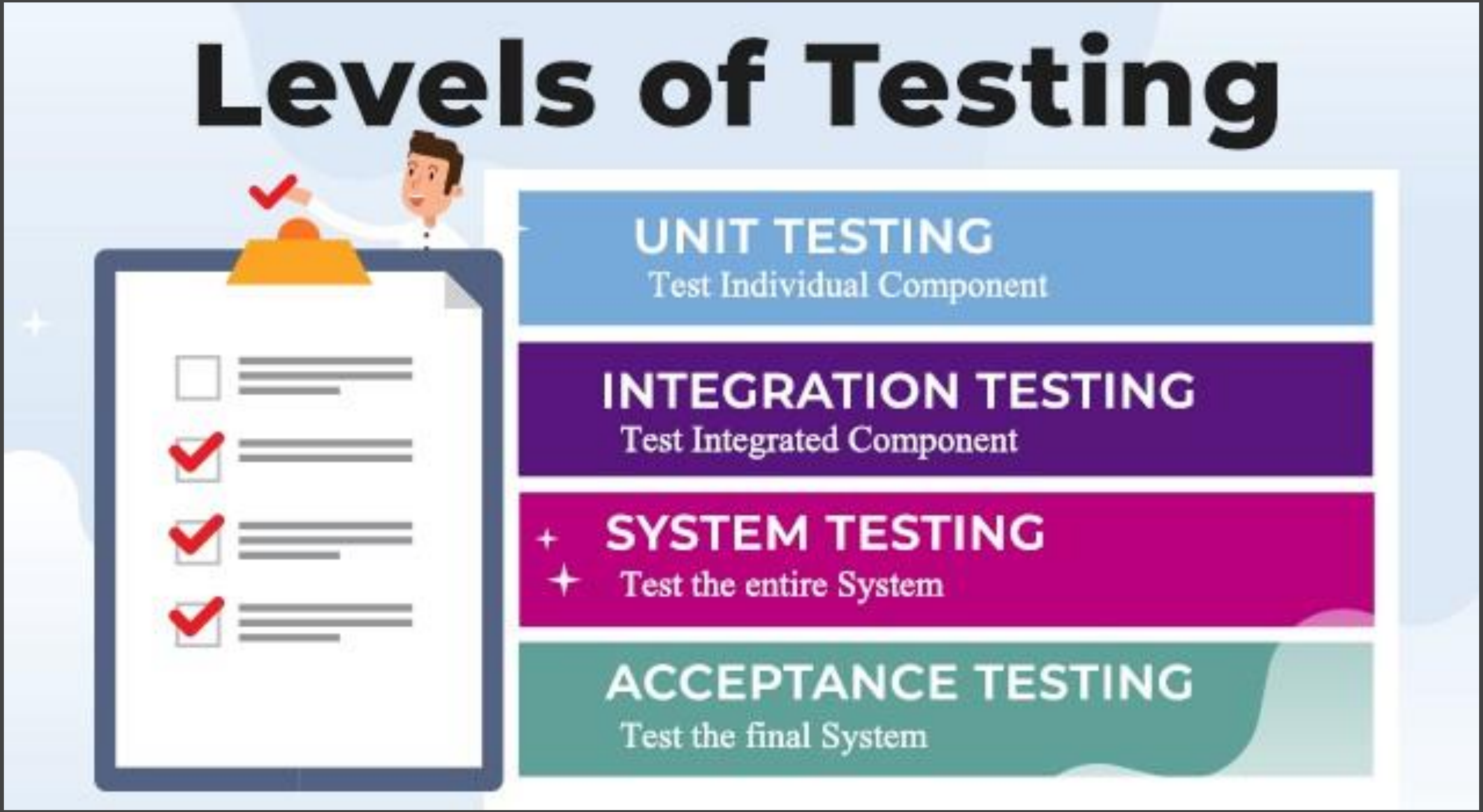
UNIT TESTING

Krishnakripa Jayakumar

WHY TEST?

Why do you need to test your project?

An activity to check whether the actual results match the expected results and to ensure that the software system is Defect free.



Levels of Testing

UNIT TESTING
Test Individual Component

INTEGRATION TESTING
Test Integrated Component

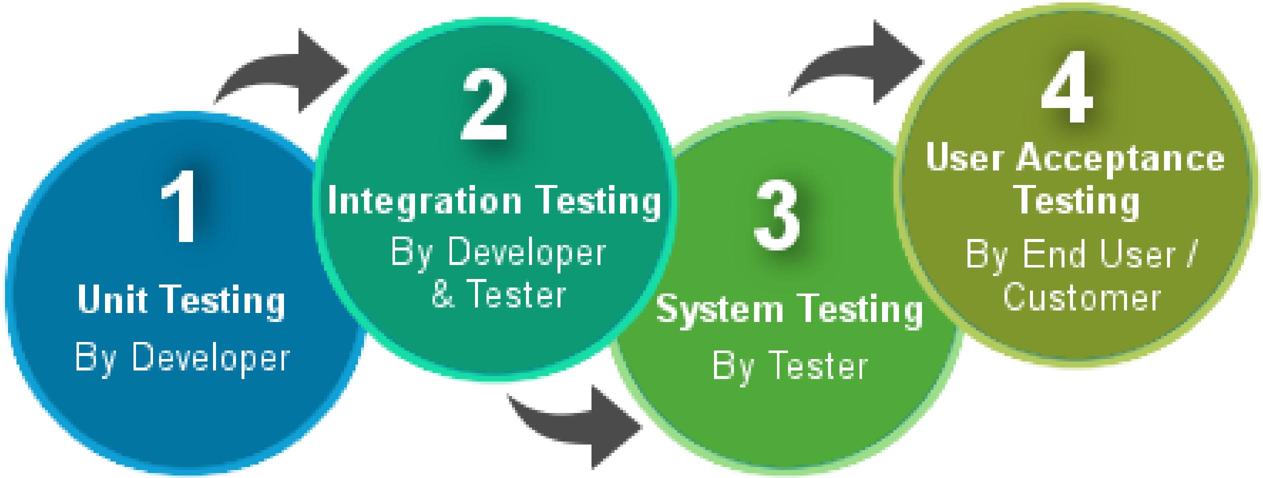
SYSTEM TESTING
Test the entire System

ACCEPTANCE TESTING
Test the final System

WHO IS RESPONSIBLE

FOR THESE TESTS?

Levels of Testing



THE GOAL OF UNIT TESTING?

Segregate each part of the program and test that the individual parts are working correctly.

WHAT IS A UNIT?

- A unit is the smallest testable part of any software.
- It usually has one or a few inputs and usually a single output.
- In procedural programming, a unit may be an individual program, function, procedure, etc.
- In object-oriented programming, the smallest unit is a method.

THE GOAL OF UNIT TESTING?

- Determines whether the ‘unit’ behaves exactly as you expect.
- It allows automation of the testing process, reduces difficulties of discovering errors contained in more complex pieces of the application
- Enhances test coverage because attention is given to each unit.

01. BENEFITS OF UNIT TESTING

What benefits can it bring you?

1. MAKES THE PROCESS AGILE

- When you add more and more features to a software, you sometimes need to change old design and code.
- However, changing already-tested code is both risky and costly. If we have unit tests in place, then we can proceed for refactoring confidently.
- In other words, unit tests facilitate safe refactoring.

2. QUALITY OF CODE

- Unit testing improves the quality of the code.
- It identifies every defect that may have come up before code is sent further for integration testing
- Writing tests before actual coding makes you think harder about the problem.
- It exposes the edge cases and makes you write better code.

3. FINDING BUGS EARLY

- Issues are found at an early stage.
- Since unit testing is carried out by developers who test individual code before integration, issues can be found very early and can be resolved then and there without impacting the other pieces of the code.
- This includes both bugs in the programmer' s implementation and flaws or missing parts of the specification for the unit.

4. REDUCES COST

- Since the bugs are found early, unit testing helps reduce the cost of bug fixes.
- Just imagine the cost of a bug found during the later stages of development, like during system testing or during acceptance testing.
- Of course, bugs detected earlier are easier to fix because bugs detected later are usually the result of many changes, and you don’t really know which one caused the bug.

BUT IS UNIT TESTING FOR YOU?

02. UNIT TESTING IN JAVA

TERMINOLOGY

TEST SUITE

The test suit is a group of test cases combined tests a certain functionality or module. The relation between test cases and test suit is many to many, as one test case can be part of multiple test suits.

CODE COVERAGE

Code Coverage represents the amount of the code covered by unit testing.

FRAMEWORKS & JUnit

- Unit testing have a lot of frameworks that help simplify the process of unit testing and help in testing automation.
- JUnit is a simple framework to write repeatable tests.
- JUnit is open source project can easily be used and automated.
- JUnit is the most used unit testing framework.
- Other Java Unit Testing alternatives :

INSTALLATION OF JUnit

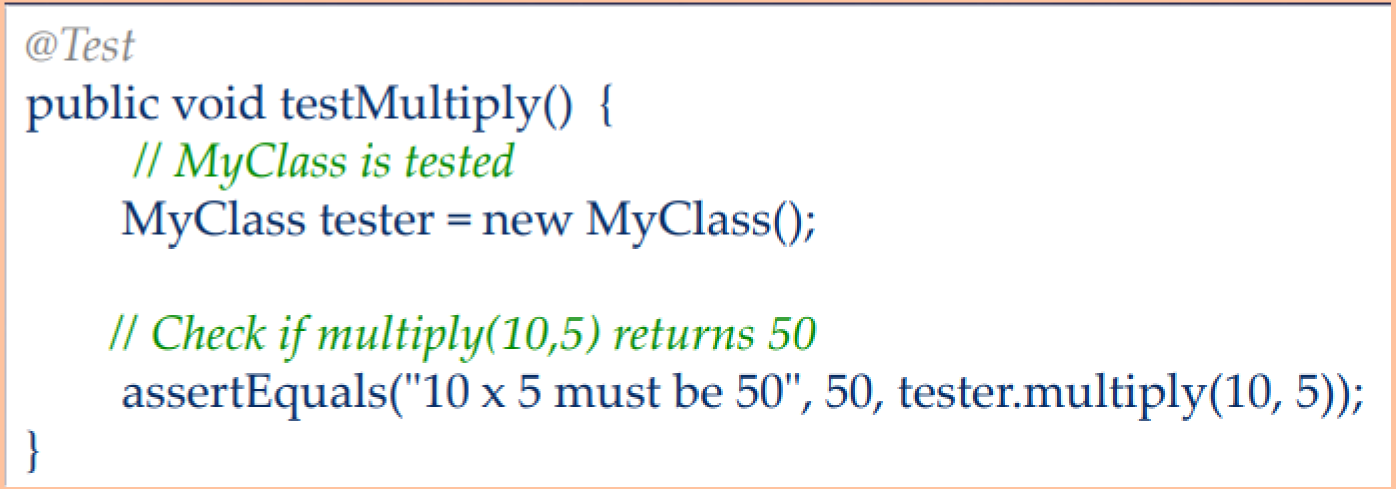
- The only thing you need to do is to add 2 JARs
    - junit.jar
    - hamcrest-core.jar

CREATING TEST UNIT

- Unit test class is not required to inherit or extend any other class or interface.
- Only the test methods need to be annotated with "@Test" annotation.
- JUnit assumes that all test methods can be executed in an arbitrary order. Therefore tests should not depend on other tests.
- Adding test methods (fail, or asserts).

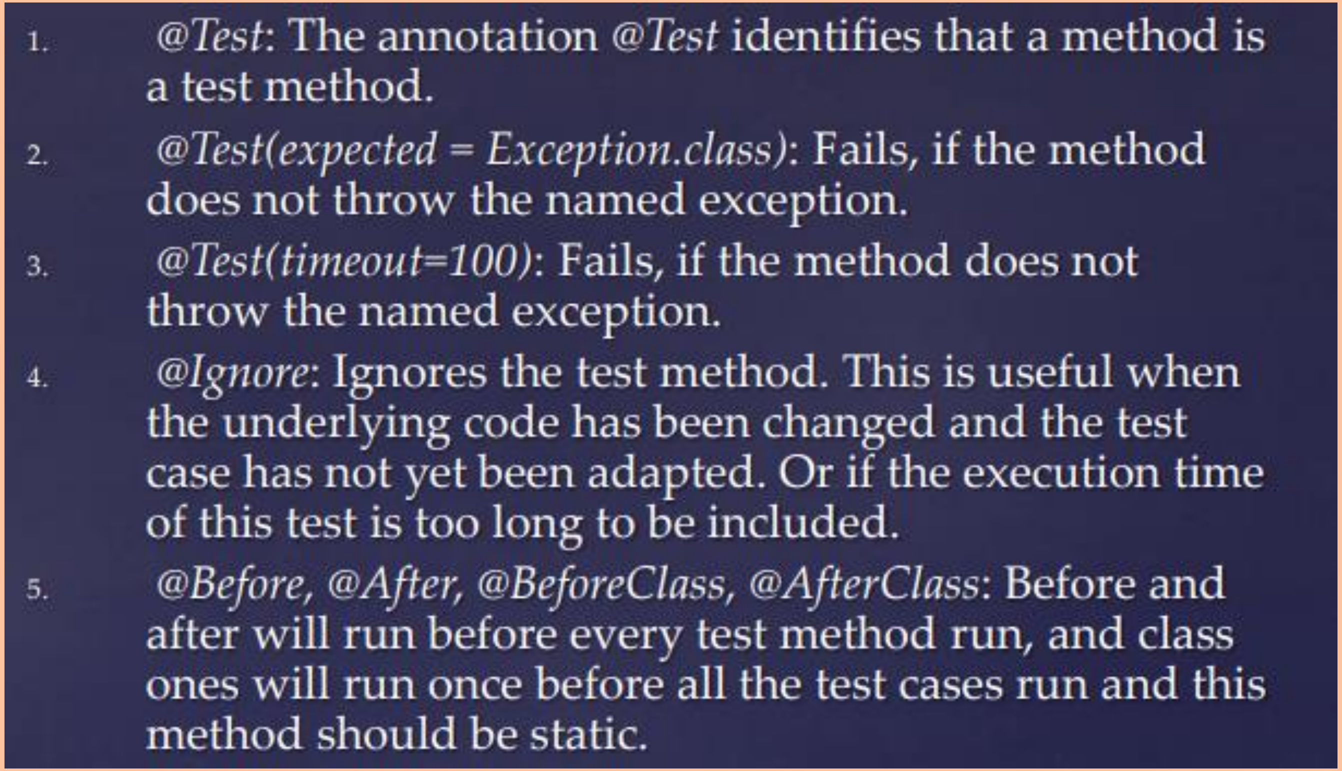
CREATING TEST UNIT

Example



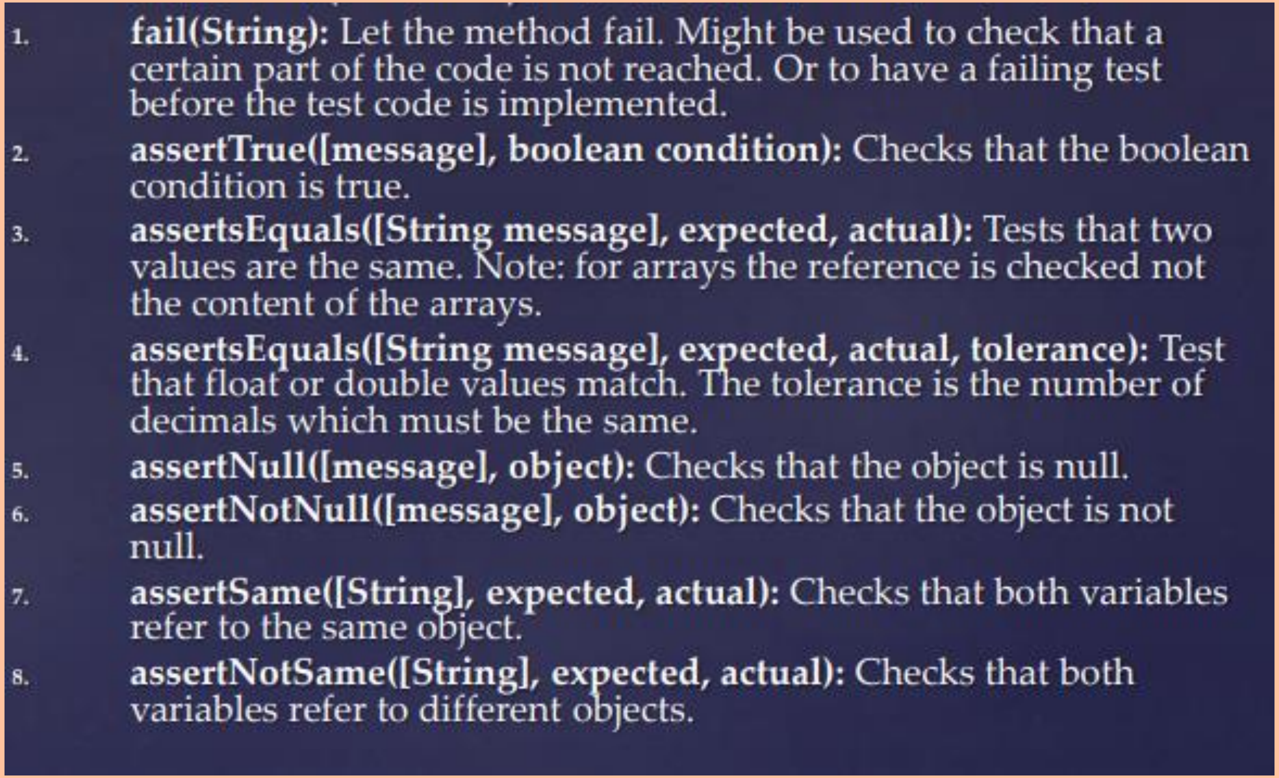
```
@Test
public void testMultiply()  {
        // MyClass is tested
        MyClass tester = new MyClass();

       // Check if multiply(10,5) returns 50
        assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
}
```

LIST OF JUnit Annotations



1. *@Test*: The annotation *@Test* identifies that a method is a test method.
2. *@Test(expected = Exception.class)*: Fails, if the method does not throw the named exception.
3. *@Test(timeout=100)*: Fails, if the method does not throw the named exception.
4. *@Ignore*: Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.
5. *@Before, @After, @BeforeClass, @AfterClass*: Before and after will run before every test method run, and class ones will run once before all the test cases run and this method should be static.

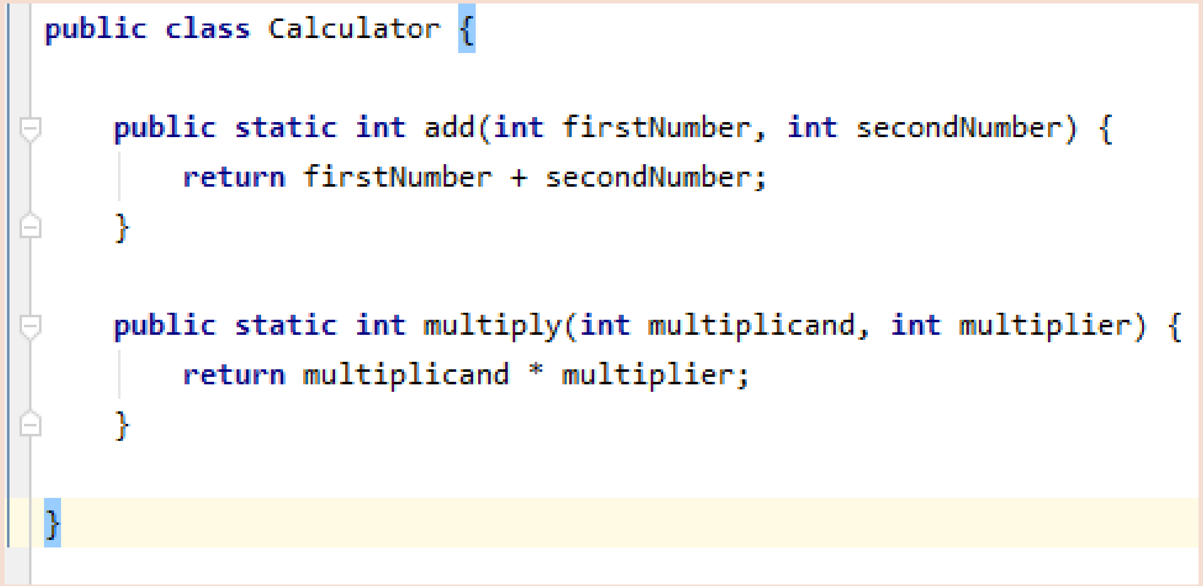
ASSERT STATEMENTS (METHODS) LIST



1. **fail(String):** Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have a failing test before the test code is implemented.
2. **assertTrue([message], boolean condition):** Checks that the boolean condition is true.
3. **assertsEquals([String message], expected, actual):** Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
4. **assertsEquals([String message], expected, actual, tolerance):** Test that float or double values match. The tolerance is the number of decimals which must be the same.
5. **assertNull([message], object):** Checks that the object is null.
6. **assertNotNull([message], object):** Checks that the object is not null.
7. **assertSame([String], expected, actual):** Checks that both variables refer to the same object.
8. **assertNotSame([String], expected, actual):** Checks that both variables refer to different objects.

Create an empty Java Project

1. Create an empty project ‘CalculatorDemo’ from IntelliJ
2. Create a class called ‘Calculator.java’ inside the src folder

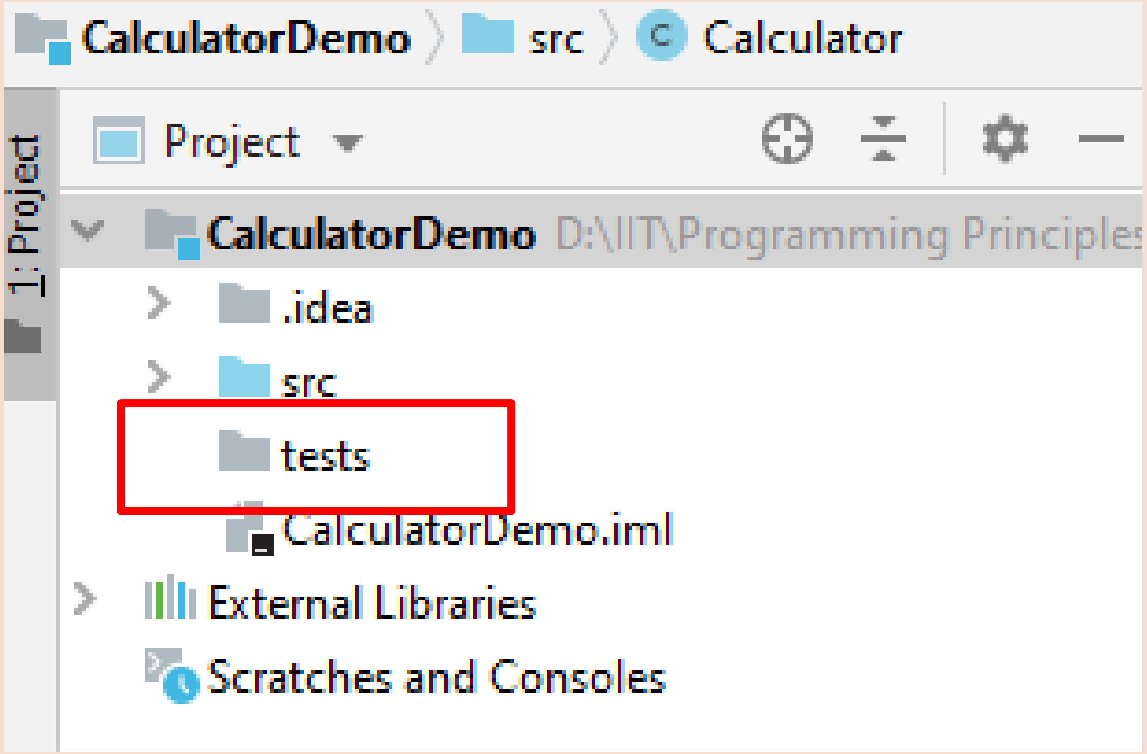


```java
public class Calculator {

    public static int add(int firstNumber, int secondNumber) {
        return firstNumber + secondNumber;
    }

    public static int multiply(int multiplicand, int multiplier) {
        return multiplicand * multiplier;
    }

}
```

Setting Up Unit Tests in a Project

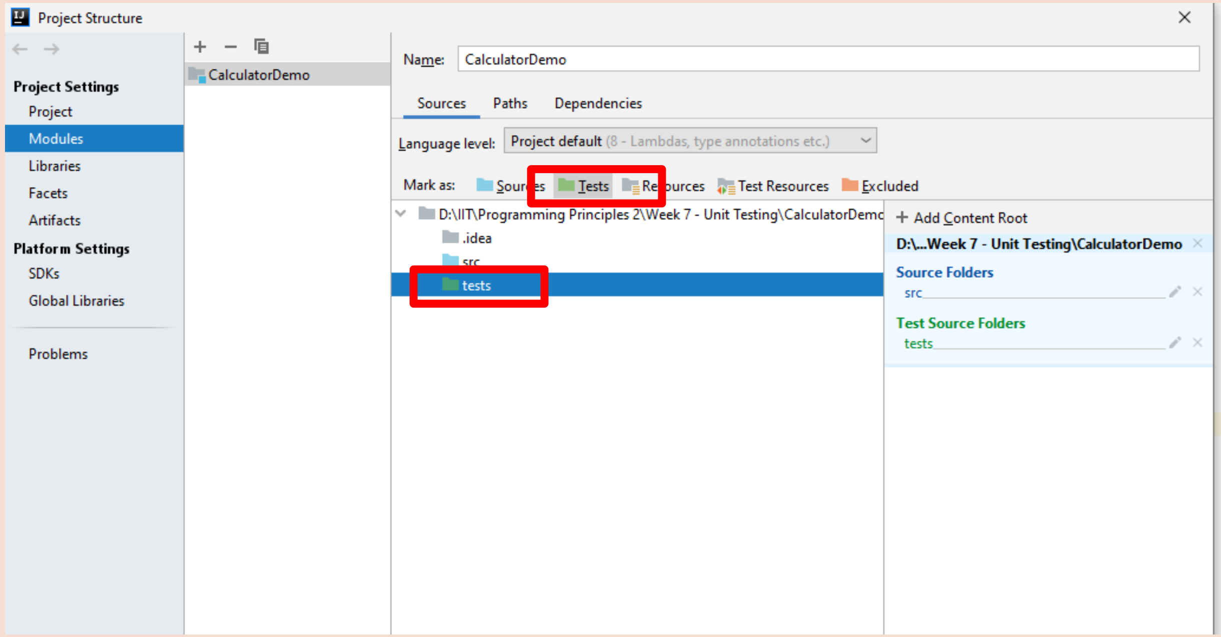
1. Create a new directory, ‘tests’ inside the Project

This directory will hold all the unit tests for the Calculator project



Mark the directory you created as the Tests directory in the project

1. Click on File-> Project Structure
2. In the window, select 'Modules'
3. Under 'Sources' tab, click on the 'tests' directory you created
4. Also click on 'Tests' marked in green



Name: CalculatorDemo

Language level: Project default (8 - Lambdas, type annotations etc.)

D:\IIT\Programming Principles 2\Week 7 - Unit Testing\CalculatorDemo
.idea
src
tests

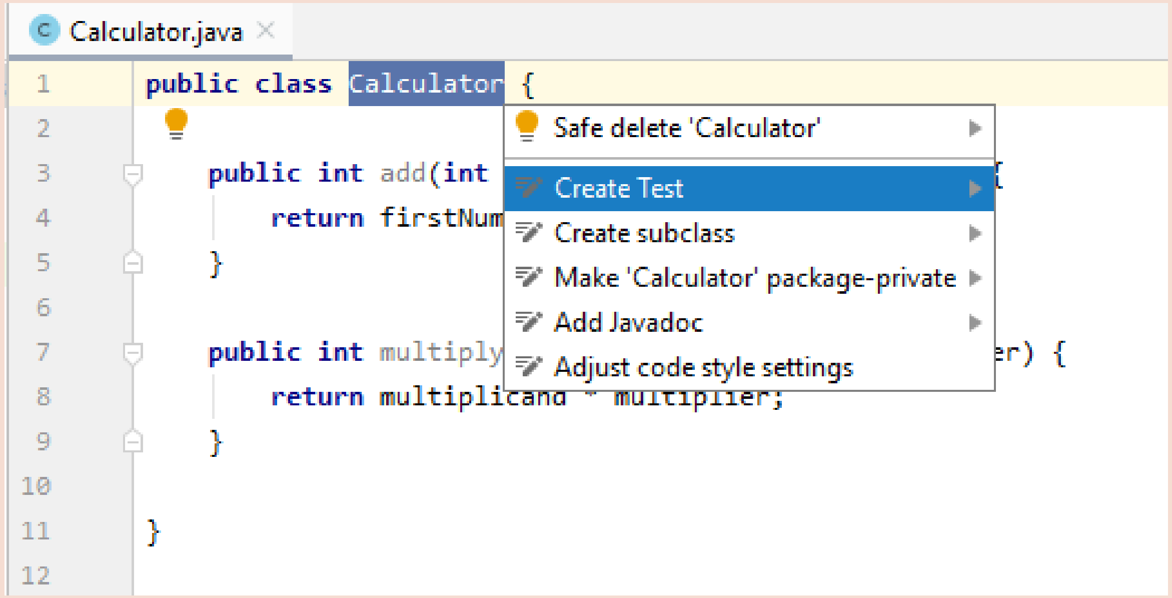
D:\...Week 7 - Unit Testing\CalculatorDemo

Source Folders
src

Test Source Folders
tests

Create a Test class for the java file

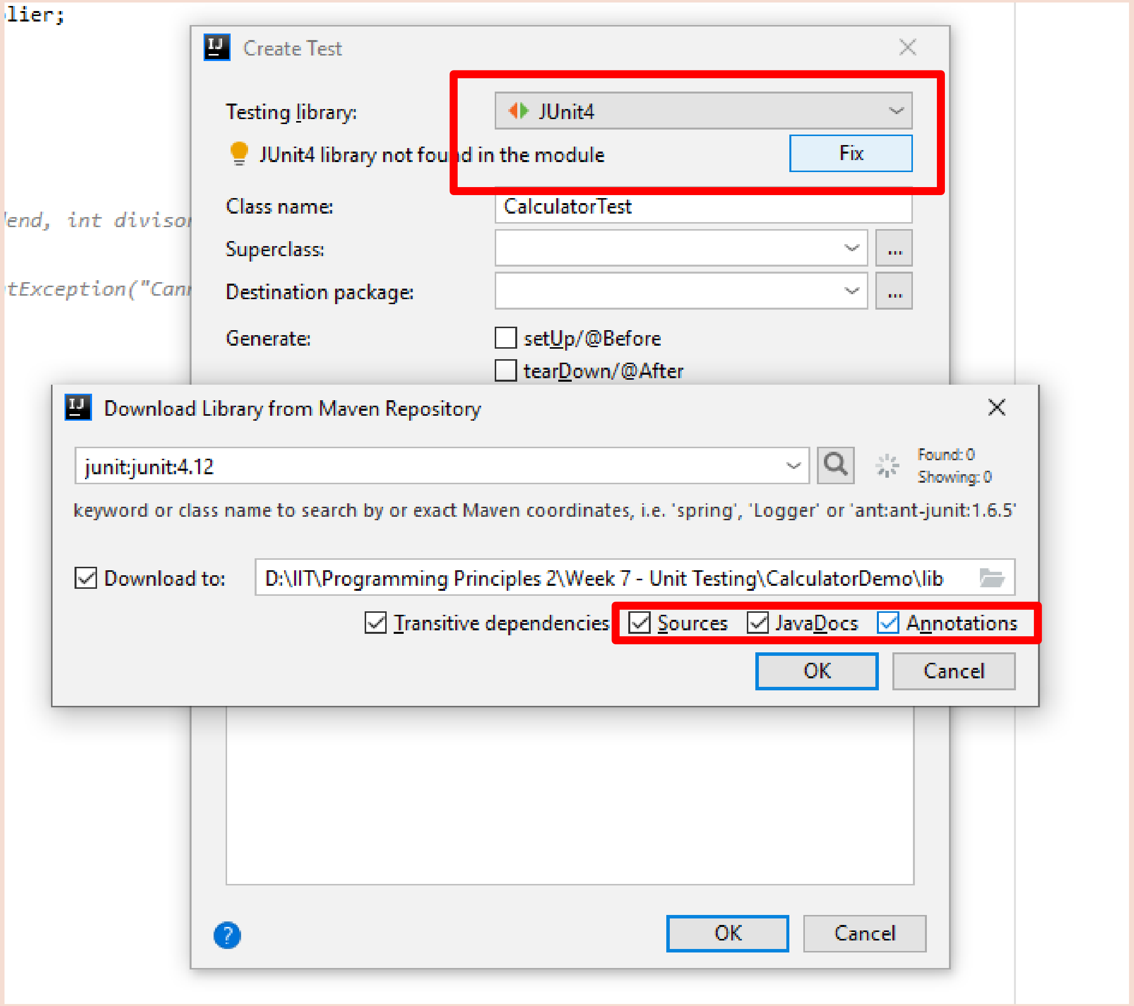
1. Click on the class name and right click -> Show context Actions
2. Click on 'Create Test Option'



```
public class Calculator {

    public int add(int
        return firstNum
    }

    public int multiply                    er) {
        return multiplicand * multiplier;
    }

}
```

Create a Test class for the java file

1. Select the Testing Library as Junit4
2. Click on the 'Fix' button
3. In the newly opened window, click on the option to 'Download to'
4. Select all check boxes underneath it



```
lier;
end, int divisor
tException("Cann
```

Create Test

Testing library: JUnit4

JUnit4 library not found in the module [Fix]

Class name: CalculatorTest

Superclass:

Destination package:

Generate: ☐ setUp/@Before ☐ tearDown/@After

Download Library from Maven Repository

junit:junit:4.12

Found: 0
Showing: 0

keyword or class name to search by or exact Maven coordinates, i.e. 'spring', 'Logger' or 'ant:ant-junit:1.6.5'

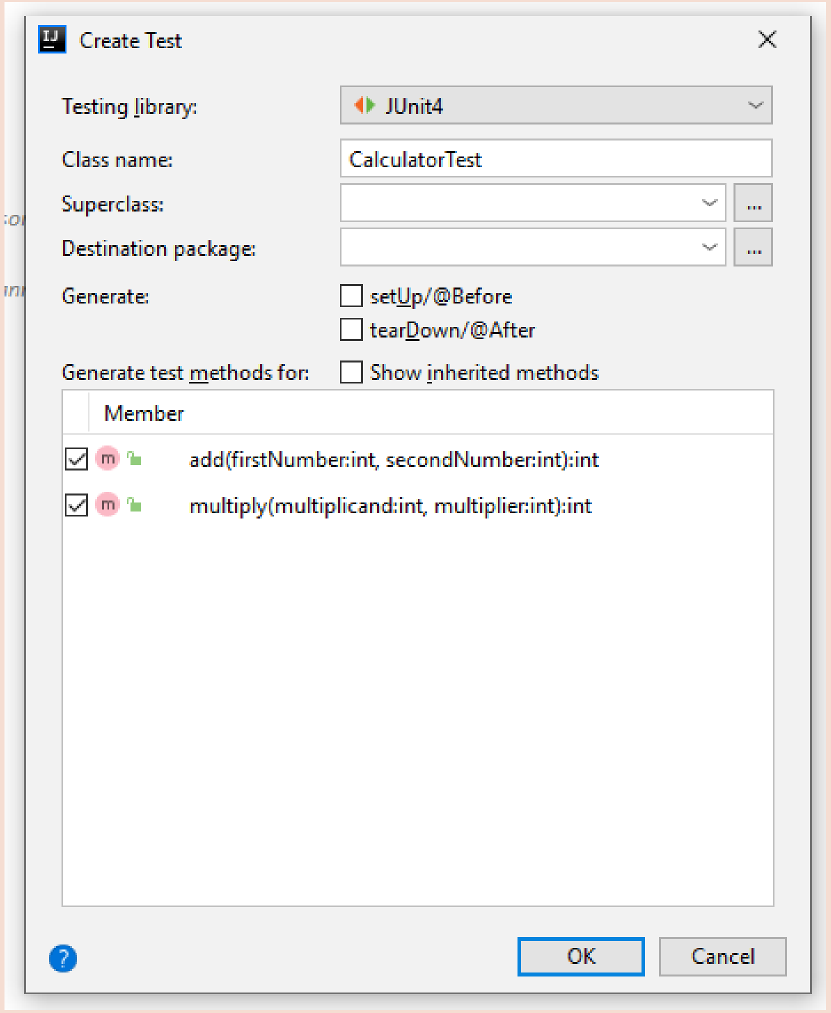
☑ Download to: D:\IIT\Programming Principles 2\Week 7 - Unit Testing\CalculatorDemo\lib

☑ Transitive dependencies ☑ Sources ☑ JavaDocs ☑ Annotations

Create a Test class for the java file

1. Select all methods you want to create tests for
2. Click OK
3. Check inside the tests folder

A new test file (CalculatorTest.java) should be created for you.



Testing library: JUnit4

Class name: CalculatorTest

Superclass:

Destination package:

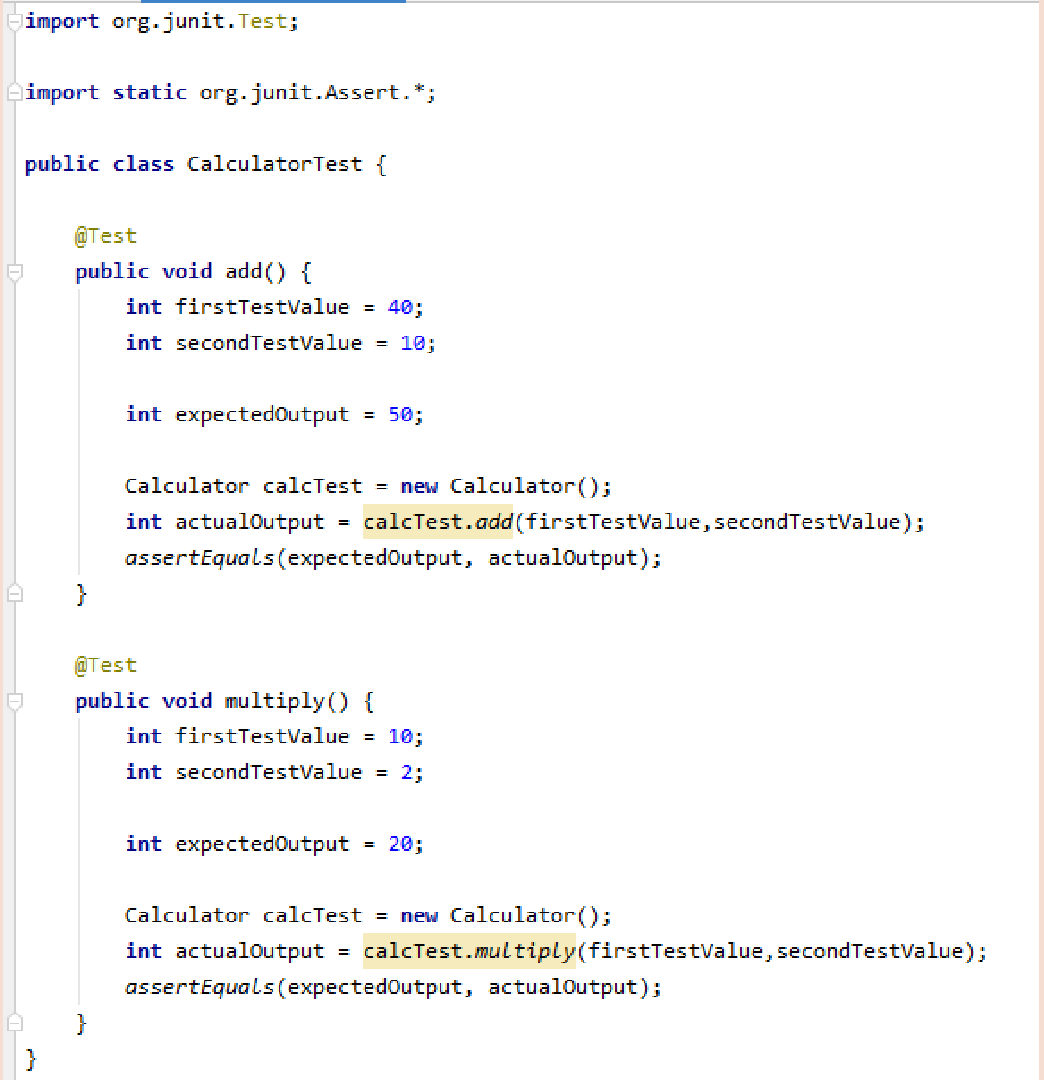
Generate: ☐ setUp/@Before ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

	Member
☑	add(firstNumber:int, secondNumber:int):int
☑	multiply(multiplicand:int, multiplier:int):int

Create unit tests for add() and multiply() of the Calculator class

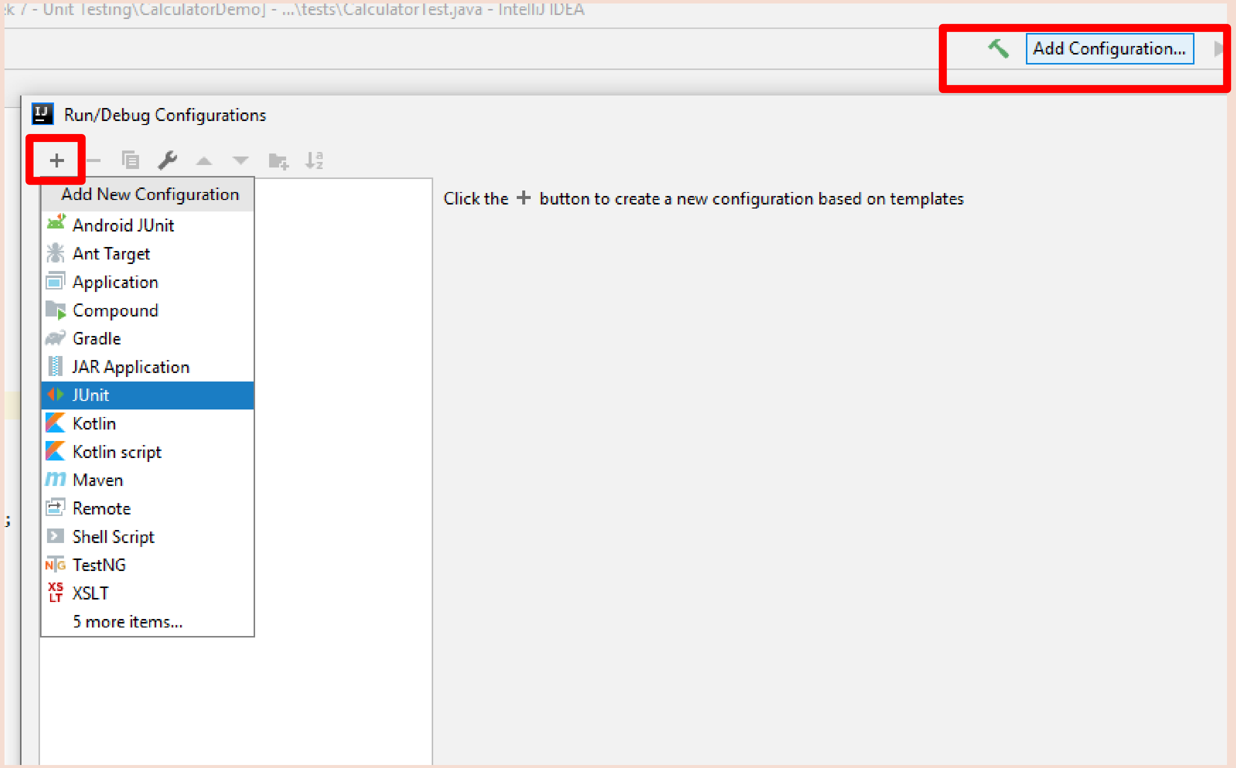
1. Inside CalculatorTest.java fill the unit tests



```java
import org.junit.Test;

import static org.junit.Assert.*;

public class CalculatorTest {

    @Test
    public void add() {
        int firstTestValue = 40;
        int secondTestValue = 10;

        int expectedOutput = 50;

        Calculator calcTest = new Calculator();
        int actualOutput = calcTest.add(firstTestValue,secondTestValue);
        assertEquals(expectedOutput, actualOutput);
    }

    @Test
    public void multiply() {
        int firstTestValue = 10;
        int secondTestValue = 2;

        int expectedOutput = 20;

        Calculator calcTest = new Calculator();
        int actualOutput = calcTest.multiply(firstTestValue,secondTestValue);
        assertEquals(expectedOutput, actualOutput);
    }
}
```

Creating a Run configuration to run the unit tests

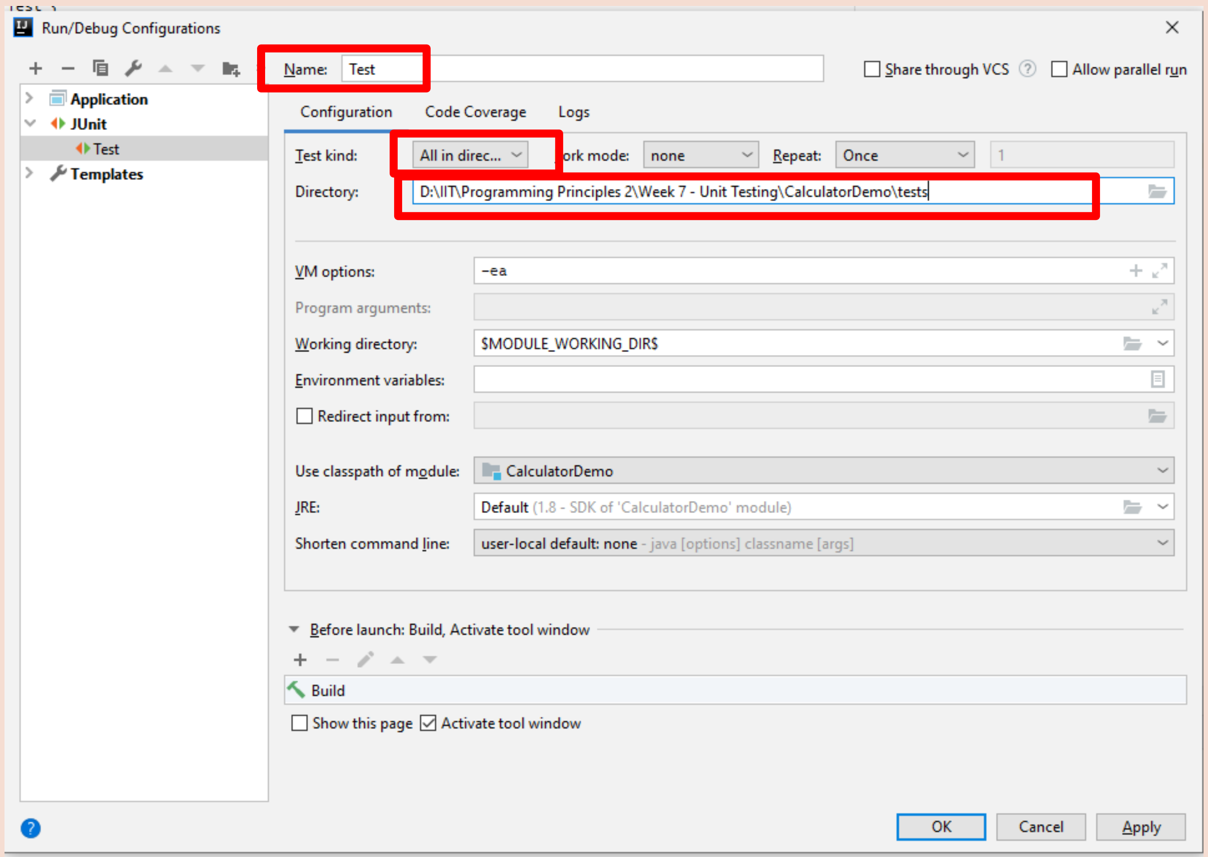
1. Click on 'Add configuration'
2. In the window that opens up click on the + button on the left
3. In the dropdown select 'JUnit'



Click the + button to create a new configuration based on templates

Creating a Run configuration to run the unit tests

1. Give a name for this configuration
2. In the dropdown for 'Test kind' select 'All in directory'
3. Give the path to the 'tests' directory you have created in the project



Name: Test

Test kind: All in direc...   Fork mode: none   Repeat: Once   1

Directory: D:\IIT\Programming Principles 2\Week 7 - Unit Testing\CalculatorDemo\tests

VM options: -ea

Program arguments:

Working directory: $MODULE_WORKING_DIR$

Environment variables:

Redirect input from:

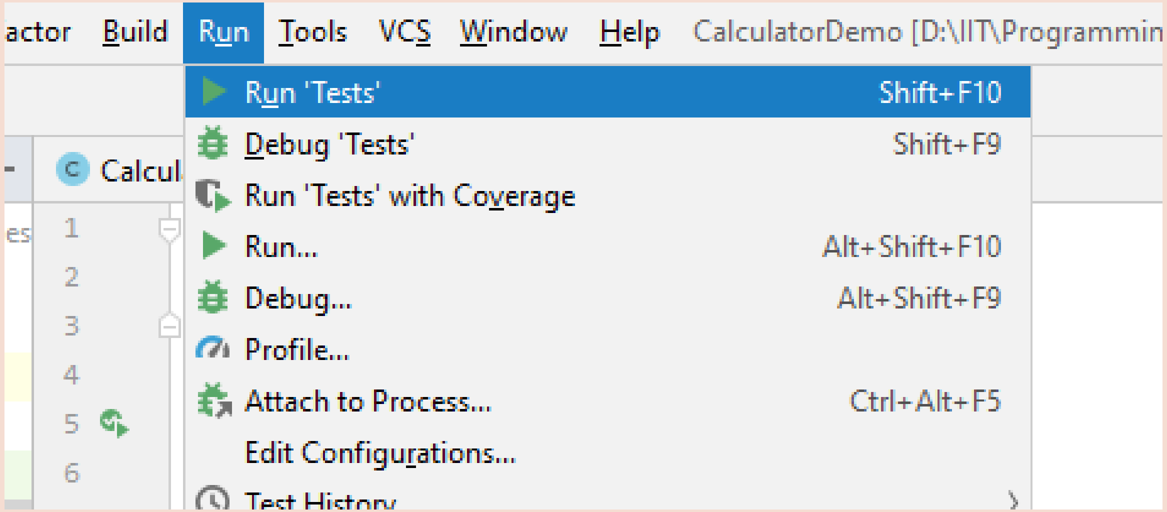
Use classpath of module: CalculatorDemo

JRE: Default (1.8 - SDK of 'CalculatorDemo' module)

Shorten command line: user-local default: none - java [options] classname [args]

Run the unit tests

1. Click on ‘Run’ from the menu bar
2. Click on ‘Run ‘’Tests’ to execute all the unit tests inside the CalculatorTest.java


actor
Build
Run
Tools
VCS
Window
Help
CalculatorDemo [D:\IIT\Programmin
Calcul
Run 'Tests' Shift+F10
Debug 'Tests' Shift+F9
Run 'Tests' with Coverage
Run... Alt+Shift+F10
Debug... Alt+Shift+F9
Profile...
Attach to Process... Ctrl+Alt+F5
Edit Configurations...
Test History