BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI CS F213 – Object Oriented Programming ***Lab 3: Unit testing using JUnit Framework***	
Time :	2 hours
Objective :	To learn the basics of unit testing in Java using JUnit, to write and execute tests to ensure code correctness and reliability.
Concepts Covered :	Unit testing, JUnit annotations, assertions, and debugging test failures.
Prerequisites :	Basic knowledge of Java programming and working with packages in Eclipse IDE.
Reflection/Conclusion :	Unit testing is essential for writing reliable programs and helps in identifying and fixing issues effectively.

1. What is Unit Testing ?

Unit testing is a software testing technique that involves testing individual components or units of a program in isolation. Typically, a "unit" refers to a single function, method, or class. The goal is to ensure each unit behaves as expected.

You write **unit tests** for these code units and run them automatically every time you make changes. If a test fails, it helps you quickly find and fix the issue. Unit tests pass input to the code unit you are checking for, and it compares the actual output, with the expected output. These tests check that the code work as expected based on the logic the developer intended. Multiple tests are written for a single unit to cover different possible scenarios (inputs) and these are called **test cases**. While it is ideal to cover all expected behaviors, it is not always necessary to test every scenario.

Unit testing helps detect bugs early in the development cycle, enhances code quality, and reduces the cost and effort of fixing issues later. It also promotes confidence in code changes, especially in larger projects. It is an essential part of **Test-Driven Development**[1], promoting reliable, maintainable, and well-tested code.

1.1 JUnit

JUnit is a popular open-source framework for writing and running unit tests in Java. It simplifies the process of defining, executing, and analyzing tests while promoting clean and maintainable code. With seamless integration into IDEs like Eclipse and IntelliJ, and support for build tools such as Maven and Gradle, JUnit has become an essential tool in modern Java development.

Its simplicity and widespread adoption make it an excellent starting point for developers new to testing. While other testing frameworks like TestNG, Spock, and Mockito exist, this course will focus on JUnit as the foundation for learning unit testing.

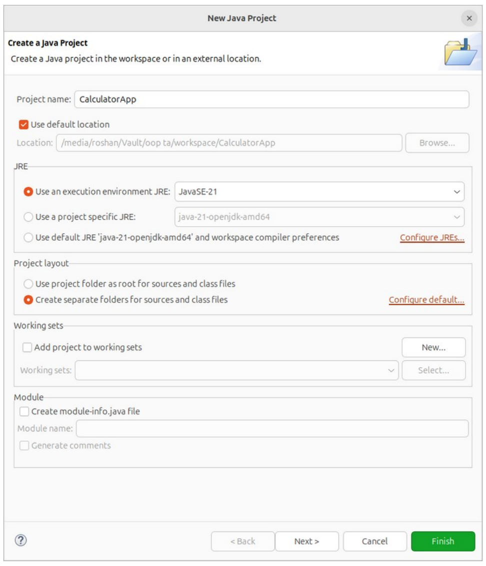
1 **Test Driven Development (TDD)** is a software development practice where developers write automated tests before writing the actual code that needs to be tested. It is an iterative approach combining Programming, Unit Test Creation, and Refactoring.

2. Using JUnit in Eclipse

JUnit comes preinstalled with more recent versions of Eclipse *(eclipse 3.7 and later)*. For earlier versions of eclipse, you will have to manually download the JUnit jar file, and include it in your project.

2.1 Writing tests in JUnit

To start writing tests in eclipse using JUnit, lets create a project called, **CalculatorApp.**



New Java Project

Create a Java Project
Create a Java project in the workspace or in an external location.

Project name: CalculatorApp

[✓] Use default location
Location: /media/roshan/Vault/oop ta/workspace/CalculatorApp

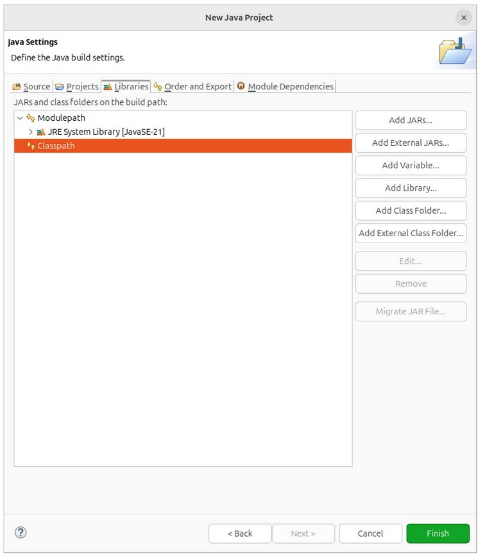
JRE
● Use an execution environment JRE: JavaSE-21
○ Use a project specific JRE: java-21-openjdk-amd64
○ Use default JRE 'java-21-openjdk-amd64' and workspace compiler preferences

Project layout
○ Use project folder as root for sources and class files
● Create separate folders for sources and class files

Working sets
[ ] Add project to working sets
Working sets:

Module
[ ] Create module-info.java file
Module name:
[ ] Generate comments

I have deselected the option to create a module-info.java file to keep this project simple. This file is part of **Java's module system**[2], which introduces a way to encapsulate related packages and classes under a single module. The module-info.java file serves as the module descriptor, storing metadata such as the module's name, the packages it exports, and the other modules it depends on.



New Java Project

Java Settings
Define the Java build settings.

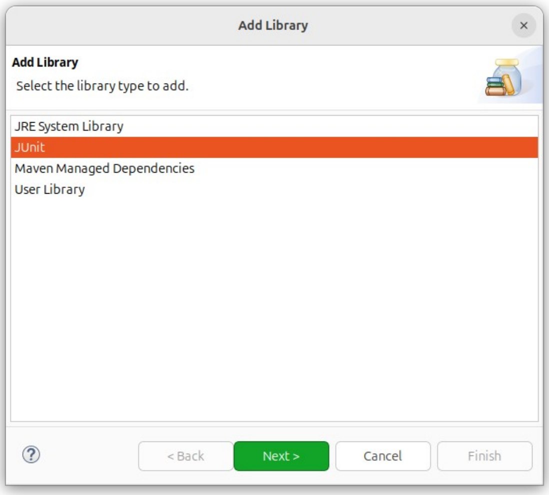
JARs and class folders on the build path:
- Modulepath
  - JRE System Library [JavaSE-21]
- Classpath

Since this project is very basic and aimed solely at demonstrating how to use JUnit in Eclipse, we will not utilize modules. Once you have entered these details, click **Next** to proceed.

2 **Java Platform Module System (JPMS)** was introduced in Java 9. To learn more about modules, check this link.

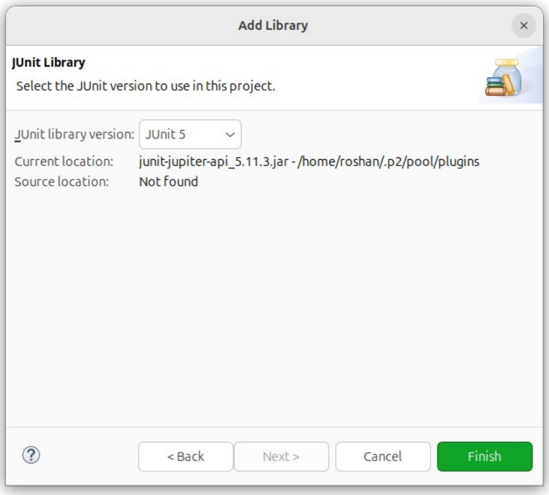
In the next dialog box, we need to add JUnit to the **classpath**[3] to enable its use in our project.

At the top of the dialog, switch to the **Libraries** tab. Click on **Classpath** (make sure that it's selected) , and on the right-hand side, you will see various buttons for adding JARs, external JARs, or libraries. Click the **Add Library** button.


Add Library

Add Library
Select the library type to add.

JRE System Library
JUnit
Maven Managed Dependencies
User Library

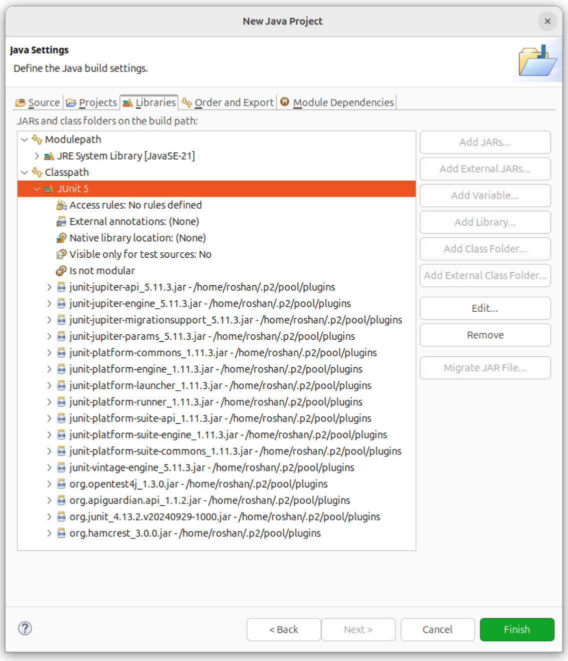

Add Library

JUnit Library
Select the JUnit version to use in this project.

JUnit library version: JUnit 5
Current location: junit-jupiter-api_5.11.3.jar - /home/roshan/.p2/pool/plugins
Source location: Not found

In the **Add Library** dialog box, select **JUnit** from the list and click **Next**. Then, choose **JUnit 5** from the dropdown menu and click the **Finish** button.

Now verify whether the classpath has the **JUnit5** library. Click on **Finish** to create the java project.


New Java Project

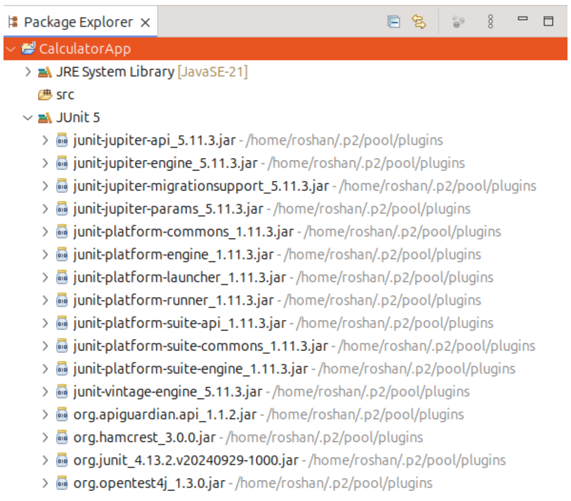
Java Settings
Define the Java build settings.

JARs and class folders on the build path:

```
Modulepath
  JRE System Library [JavaSE-21]
Classpath
  JUnit 5
      Access rules: No rules defined
      External annotations: (None)
      Native library location: (None)
      Visible only for test sources: No
      Is not modular
      junit-jupiter-api_5.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-jupiter-engine_5.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-jupiter-migrationsupport_5.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-jupiter-params_5.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-platform-commons_1.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-platform-engine_1.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-platform-launcher_1.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-platform-runner_1.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-platform-suite-api_1.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-platform-suite-engine_1.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-platform-suite-commons_1.11.3.jar - /home/roshan/.p2/pool/plugins
      junit-vintage-engine_5.11.3.jar - /home/roshan/.p2/pool/plugins
      org.opentest4j_1.3.0.jar - /home/roshan/.p2/pool/plugins
      org.apiguardian.api_1.1.2.jar - /home/roshan/.p2/pool/plugins
      org.junit_4.13.2.v20240929-1000.jar - /home/roshan/.p2/pool/plugins
      org.hamcrest_3.0.0.jar - /home/roshan/.p2/pool/plugins
```

Once your project is created, the directory structure should look something like this,

3 The **classpath** in Java is a parameter that specifies the location of classes and packages required by the Java Virtual Machine (JVM) at runtime. It tells the JVM where to look for compiled .class files and libraries needed to execute a Java program.

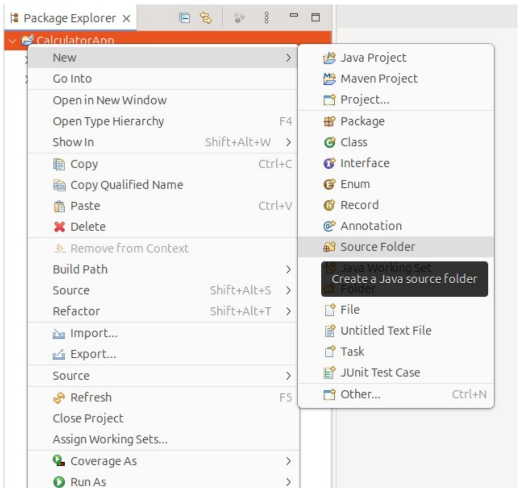


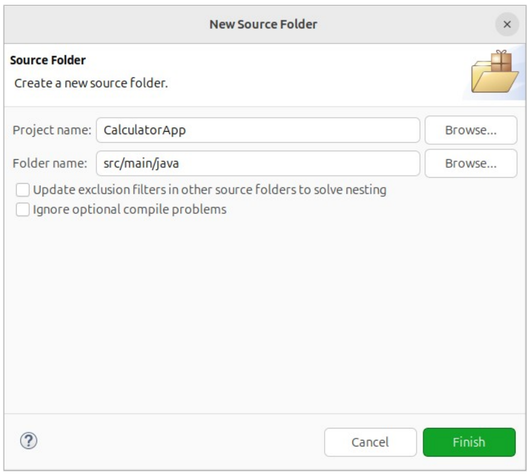
```
Package Explorer
CalculatorApp
  JRE System Library [JavaSE-21]
  src
  JUnit 5
    junit-jupiter-api_5.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-jupiter-engine_5.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-jupiter-migrationsupport_5.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-jupiter-params_5.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-platform-commons_1.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-platform-engine_1.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-platform-launcher_1.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-platform-runner_1.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-platform-suite-api_1.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-platform-suite-commons_1.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-platform-suite-engine_1.11.3.jar - /home/roshan/.p2/pool/plugins
    junit-vintage-engine_5.11.3.jar - /home/roshan/.p2/pool/plugins
    org.apiguardian.api_1.1.2.jar - /home/roshan/.p2/pool/plugins
    org.hamcrest_3.0.0.jar - /home/roshan/.p2/pool/plugins
    org.junit_4.13.2.v20240929-1000.jar - /home/roshan/.p2/pool/plugins
    org.opentest4j_1.3.0.jar - /home/roshan/.p2/pool/plugins
```

Now that our project is set up, let's begin by writing the main program. Before we dive into coding, let's first discuss how to structure the project effectively.

For this project and the subsequent examples in this lab sheet, we'll be adopting **Maven's standard project layout**. Maven is an open-source build automation and project management tool commonly used in Java development. (*While we won't be using Maven itself, we'll follow its recommended directory structure for organizing our project.*)

Let's start by deleting the src folder. Then create a new **source folder** by right clicking on the project name, **New → Source Folder**. Name the folder **src/main/java**[4].





New Source Folder

Source Folder

Create a new source folder.

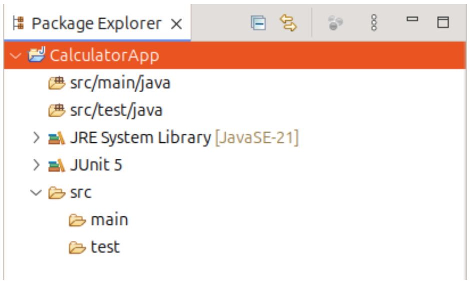
Project name: CalculatorApp

Folder name: src/main/java

☐ Update exclusion filters in other source folders to solve nesting

☐ Ignore optional compile problems

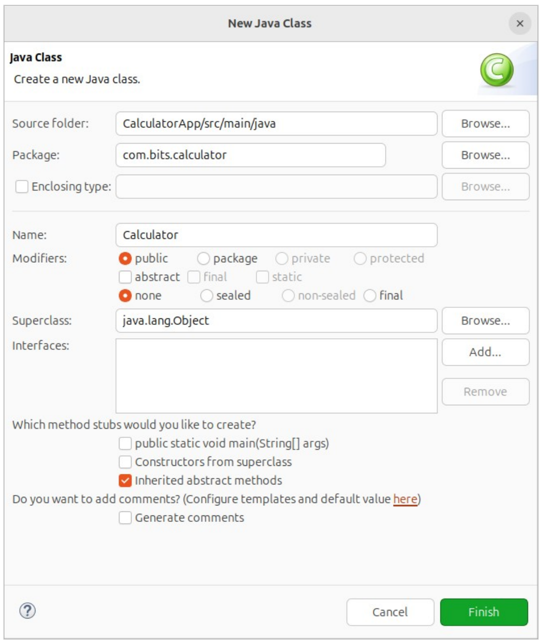
Similarly, create another source folder inside the project and call it, **src/test/java**. Your project directory should look something like this.



```
Package Explorer
CalculatorApp
  src/main/java
  src/test/java
  JRE System Library [JavaSE-21]
  JUnit 5
  src
    main
    test
```

4 Why a deep folder structure ? What do all these folder mean ? Read this.

Let's write the code for our calculator. Inside the **src/main/java** folder, create a new Java class named **Calculator**. This class should be enclosed within **com.bits.calculator**[5] package.



New Java Class

Java Class
Create a new Java class.

Source folder: CalculatorApp/src/main/java
Package: com.bits.calculator
Enclosing type: [ ]
Name: Calculator
Modifiers: [✓ public] package private protected
[ ] abstract [ ] final [ ] static
[✓ none] sealed non-sealed final
Superclass: java.lang.Object
Interfaces:

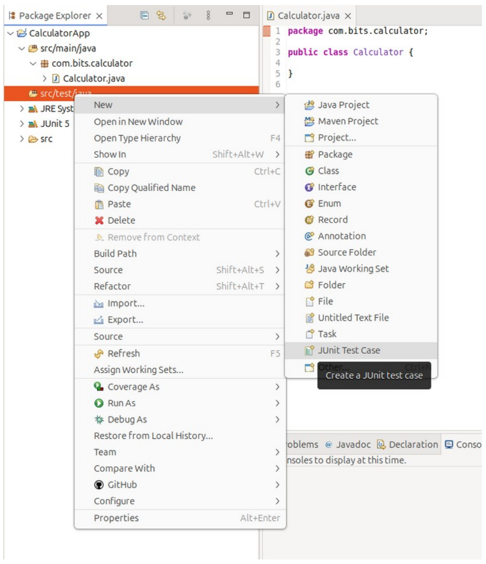
Which method stubs would you like to create?
[ ] public static void main(String[] args)
[ ] Constructors from superclass
[✓] Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
[ ] Generate comments

Inside the **Calculator.java** file, we will write the following code :

```java
package com.bits.calculator;

public class Calculator {
    // Method to add two numbers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to subtract two numbers
    public int subtract(int a, int b) {
        return a - b;
    }

    // Method to multiply two numbers
    public int multiply(int a, int b) {
        return a * b;
    }

    // Method to divide two numbers
    public double divide(int a, int b) {
        if (b == 0) {
            throw new ArithmeticException("Division by zero is not allowed.");
        }
        return (double) a / b;
    }
}
```

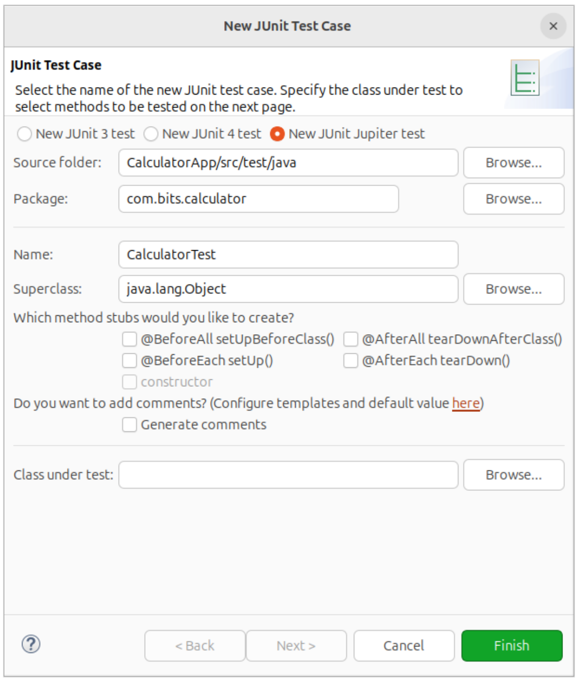
5 Why such a package name ? What package naming convention to follow ? Read this.

Now, let's begin writing tests for our program. Inside the **src/test/java** folder, create a new **JUnit Test Case**. (*src/test/java folder* → *New* → *JUnit Test Case*)



```
package com.bits.calculator;

public class Calculator {

}
```

A new dialog box will appear where you can enter the details for the test case class. In this dialog, specify the test class name as **CalculatorTest,** and enclose it within the **com.bits.calculator** package (same package as the main program files).



New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. Specify the class under test to select methods to be tested on the next page.

New JUnit 3 test [✓ New JUnit Jupiter test] New JUnit 4 test

Source folder: CalculatorApp/src/test/java

Package: com.bits.calculator

Name: CalculatorTest

Superclass: java.lang.Object

Which method stubs would you like to create?

☐ @BeforeAll setUpBeforeClass() ☐ @AfterAll tearDownAfterClass()

☐ @BeforeEach setUp() ☐ @AfterEach tearDown()

☐ constructor

Do you want to add comments? (Configure templates and default value here)

☐ Generate comments

Class under test:

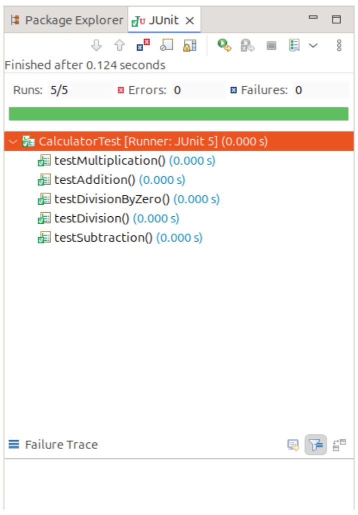
The main program files and test files should lie in the **same package**, there are mainly two reasons for this,

- It helps maintain a logical and coherent structure. It makes it clear which tests correspond to which parts of the codebase.
- It allows access to **protected** and **default** (package-private) members, enabling thorough testing without exposing these members as public.

Inside the **CalculatorTest.java** file we write the following tests,

```
package com.bits.calculator;

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

class CalculatorTest {

    private final Calculator calculator = new Calculator();

    @Test
    void testAddition() {
        assertEquals(5, calculator.add(2, 3), "2 + 3 should equal 5");
        assertEquals(-1, calculator.add(2, -3), "2 + (-3) should equal -1");
    }

    @Test
    void testSubtraction() {
        assertEquals(1, calculator.subtract(3, 2), "3 - 2 should equal 1");
        assertEquals(5, calculator.subtract(2, -3), "2 - (-3) should equal
5");
    }

    @Test
    void testMultiplication() {
        assertEquals(6, calculator.multiply(2, 3), "2 * 3 should equal 6");
        assertEquals(-6, calculator.multiply(2, -3), "2 * (-3) should equal -
6");
    }

    @Test
    void testDivision() {
        assertEquals(2.0, calculator.divide(6, 3), "6 / 3 should equal 2.0");
        assertEquals(-2.0, calculator.divide(6, -3), "6 / (-3) should equal -
2.0");
    }

    @Test
    void testDivisionByZero() {
        Exception exception = assertThrows(ArithmeticException.class, () ->
calculator.divide(1, 0));
        assertEquals("Division by zero is not allowed.",
exception.getMessage());

   }

}
```

To run the test in Eclipse, right-click on the project name and select **Run As → JUnit Test**. Eclipse will then automatically build and execute the JUnit test for you.

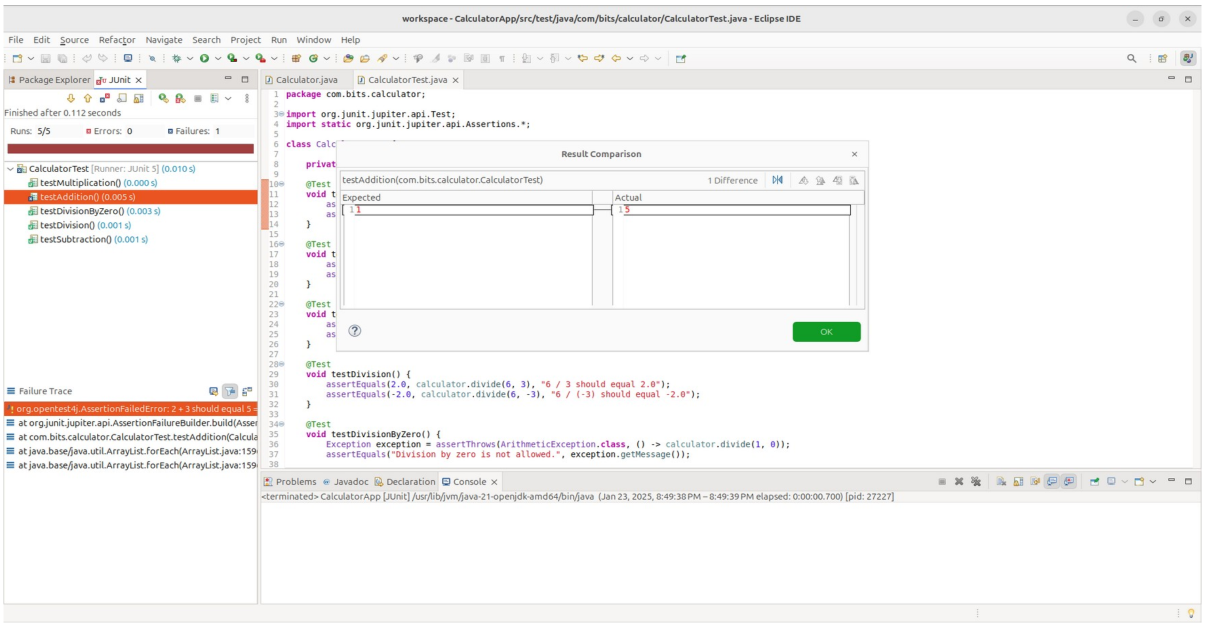


```
Finished after 0.124 seconds
Runs: 5/5    Errors: 0    Failures: 0

CalculatorTest [Runner: JUnit 5] (0.000 s)
    testMultiplication() (0.000 s)
    testAddition() (0.000 s)
    testDivisionByZero() (0.000 s)
    testDivision() (0.000 s)
    testSubtraction() (0.000 s)
```

A new tab should appear on the left side of your screen, indicating that all of our tests have passed. Now, let's observe what happens if one of our test cases fails. In the **CalculatorTest.java** file, change the expected value of **testAddition()**, from 5 to 1 and see the result.

```
// Test addition
    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        assertEquals(1, calculator.add(2, 3), "2 + 3 should equal 5");
    }
```

We should see something like this,



```
Finished after 0.112 seconds
Runs: 5/5    Errors: 0    Failures: 1

CalculatorTest [Runner: JUnit 5] (0.010 s)
    testMultiplication() (0.000 s)
    testAddition() (0.005 s)
    testDivisionByZero() (0.003 s)
    testDivision() (0.001 s)
    testSubtraction() (0.001 s)

Failure Trace
org.opentest4j.AssertionFailedError: 2 + 3 should equal 5 =
at org.junit.jupiter.api.AssertionFailureBuilder.build(Asser
at com.bits.calculator.CalculatorTest.testAddition(Calcula
at java.base/java.util.ArrayList.forEach(ArrayList.java:159
at java.base/java.util.ArrayList.forEach(ArrayList.java:159
```

```
package com.bits.calculator;

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

class Calc

    privat

    @Test
    void t
        as
        as
    }

    @Test
    void t
        as
        as
    }

    @Test
    void t
        as
        as
    }

    @Test
    void testDivision() {
        assertEquals(2.0, calculator.divide(6, 3), "6 / 3 should equal 2.0");
        assertEquals(-2.0, calculator.divide(6, -3), "6 / (-3) should equal -2.0");
    }

    @Test
    void testDivisionByZero() {
        Exception exception = assertThrows(ArithmeticException.class, () -> calculator.divide(1, 0));
        assertEquals("Division by zero is not allowed.", exception.getMessage());
```

```
Result Comparison
testAddition(com.bits.calculator.CalculatorTest)    1 Difference
Expected        Actual
1 1             1 5
```

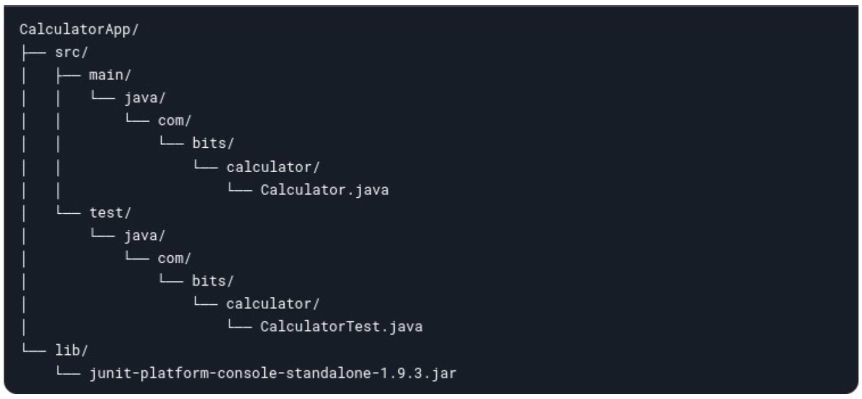
```
<terminated> CalculatorApp [JUnit] /usr/lib/jvm/java-21-openjdk-amd64/bin/java (Jan 23, 2025, 8:49:38 PM – 8:49:39 PM elapsed: 0:00:00.700) [pid: 27227]
```

The JUnit panel indicates that one of our tests have failed. If a test fails, the **Failure Trace** section shows detailed information about the failure, including the specific error message and the stack trace. This trace highlights the sequence of method calls that led to the failure, making it easier to identify the exact cause of the issue.

2.2 Running JUnit on the command line

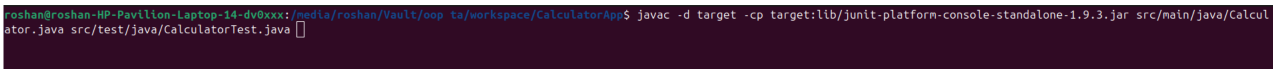
JUnit5 comes with a standalone console jar file, which allows you to run JUnit from the command line. You can download the console application from this link. (*At the time of making this lab sheet, the latest version of the console application was 1.9.3. Thus, navigate to* ***1.9.3 → junit-platform-console-standalone-1.9.3.jar***[6]*. It is possible that newer versions may have come out, by the time you are reading this lab sheet. Navigate similarly.*)

You should have downloaded *junit-platform-console-standalone-1.9.3.jar.* Next, create a folder named **lib**[7] within your project directory and place the downloaded JAR file inside it. Your project directory structure should now resemble the following:



```
CalculatorApp/
├── src/
│   ├── main/
│   │   └── java/
│   │       └── com/
│   │           └── bits/
│   │               └── calculator/
│   │                   └── Calculator.java
│   └── test/
│       └── java/
│           └── com/
│               └── bits/
│                   └── calculator/
│                       └── CalculatorTest.java
└── lib/
    └── junit-platform-console-standalone-1.9.3.jar
```

Next, we'll use the JUnit console JAR file to execute tests from the command line. Start by typing the following command on the command line to compile all your program and test files:

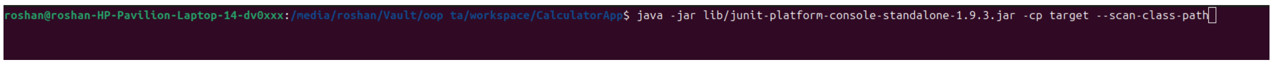


```
roshan@roshan-HP-Pavilion-Laptop-14-dv0xxx:/media/roshan/Vault/oop ta/workspace/CalculatorApp$ javac -d target -cp target:lib/junit-platform-console-standalone-1.9.3.jar src/main/java/Calcul
ator.java src/test/java/CalculatorTest.java
```

Let's breakdown the command,

- *javac : The Java compiler command, used to compile .java source files into .class bytecode files.*
- *d target : -d flag specifies the destination directory for the compiled .class files. target folder is where the compiled files will be placed.*
- *-cp : Stands for "classpath". It specifies the location of additional classes or libraries that the program needs for compilation. Followed by target folder and JAR file, separated by a colon.*

And the rest of it is the java files required in our project. Now to run the tests, type in the below command,



```
roshan@roshan-HP-Pavilion-Laptop-14-dv0xxx:/media/roshan/Vault/oop ta/workspace/CalculatorApp$ java -jar lib/junit-platform-console-standalone-1.9.3.jar -cp target --scan-class-path
```

Let's breakdown the command again :

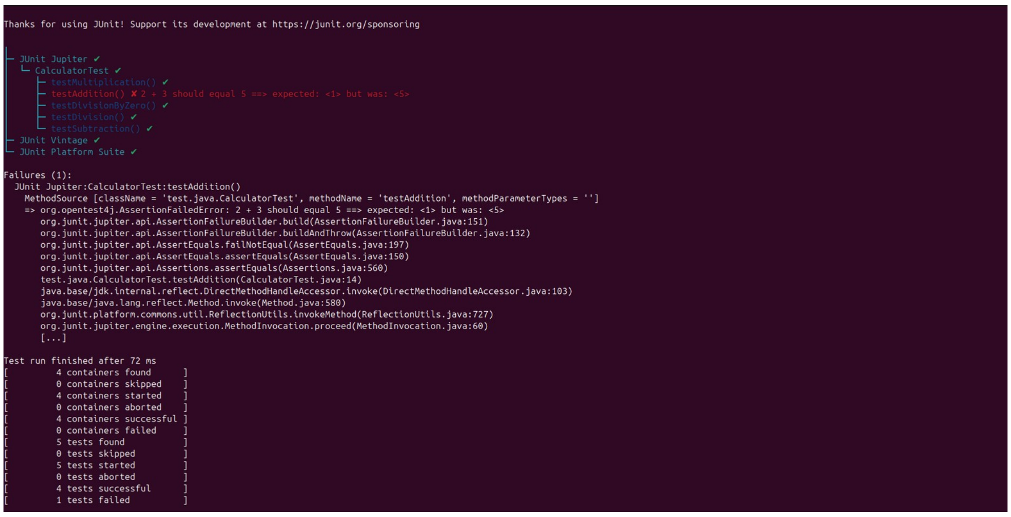
- *java : The Java Runtime Environment (JRE) command for running Java applications.*
- *-jar : Indicates that a JAR file is being executed. Followed by the jar file.*
- *-cp : Sets the classpath, which specifies where to look for compiled classes and other resources needed during execution. Followed by target folder, where we are storing our class files.*

6 *Hint : look for the jar file with the most downloads.*

7 In a standard Maven project, dependencies are managed using the *pom.xml* file, and Maven automatically downloads them from the central repository.

- *--scan-class-path : A **JUnit-specific option**[8] instructing the console launcher to scan the provided classpath for test classes.*

On running the command, you should see the following output,



```
Thanks for using JUnit! Support its development at https://junit.org/sponsoring

├─ JUnit Jupiter ✔
│  └─ CalculatorTest ✔
│     ├─ testMultiplication() ✔
│     ├─ testAddition() ✘ 2 + 3 should equal 5 ==> expected: <1> but was: <5>
│     ├─ testDivisionByZero() ✔
│     ├─ testDivision() ✔
│     └─ testSubtraction() ✔
├─ JUnit Vintage ✔
└─ JUnit Platform Suite ✔

Failures (1):
  JUnit Jupiter:CalculatorTest:testAddition()
    MethodSource [className = 'test.java.CalculatorTest', methodName = 'testAddition', methodParameterTypes = '']
    => org.opentest4j.AssertionFailedError: 2 + 3 should equal 5 ==> expected: <1> but was: <5>
       org.junit.jupiter.api.AssertionFailureBuilder.build(AssertionFailureBuilder.java:151)
       org.junit.jupiter.api.AssertionFailureBuilder.buildAndThrow(AssertionFailureBuilder.java:132)
       org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:197)
       org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:150)
       org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:560)
       test.java.CalculatorTest.testAddition(CalculatorTest.java:14)
       java.base/jdk.internal.reflect.DirectMethodHandleAccessor.invoke(DirectMethodHandleAccessor.java:103)
       java.base/java.lang.reflect.Method.invoke(Method.java:580)
       org.junit.platform.commons.util.ReflectionUtils.invokeMethod(ReflectionUtils.java:727)
       org.junit.jupiter.engine.execution.MethodInvocation.proceed(MethodInvocation.java:60)
       [...]

Test run finished after 72 ms
[         4 containers found      ]
[         0 containers skipped    ]
[         4 containers started    ]
[         0 containers aborted    ]
[         4 containers successful ]
[         0 containers failed     ]
[         5 tests found           ]
[         0 tests skipped         ]
[         5 tests started         ]
[         0 tests aborted         ]
[         4 tests successful      ]
[         1 tests failed          ]
```

Oh no! Why did it fail ? It failed because we had changed the **testAddition()** expected value in the previous example, to demonstrate what happens when a test fails. Now change the expected value, back to the correct value and run the command again.

3. Writing tests in JUnit

Let's look at a simple test, and understand each of it's components,

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {

    @Test
    void testAdd() {
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        assertEquals(5, result);
    }
}
```

The *import org.junit.jupiter.api.Test;* statement brings the **@Test** annotation into your code, allowing you to mark methods as test cases. This annotation tells JUnit to treat the method as a test and execute it during testing.

The import static *org.junit.jupiter.api.Assertions.*;* statement is a **static import** that brings all assertion methods (e.g., assertEquals, assertTrue, assertNull) from the Assertions class into your code. The assertEquals method is used to **compare two values** in a test. It checks if the **expected value** (the first argument) matches the **actual value** (the second argument). If they are equal, the test passes; if not, the test fails, and JUnit reports the mismatch.

8 *The JUnit Console Launcher has plenty of other options you can explore. Check out the official documentation, or just run: java -jar junit-platform-console-standalone-1.9.3.jar.*

Static imports allow you to use these methods directly without prefixing them with **Assertions.**, making your test code cleaner and more readable. For example, instead of **Assertions.assertEquals(5, result)**, you can simply write **assertEquals(5, result)**. This simplifies writing assertions in your tests.

Let's look at annotations and assertions in more detail.

3.1 Annotations

Annotations in JUnit are special markers that provide metadata about your test methods and classes. The can be imported from the ***org.junit.jupiter.api*** package. They help JUnit understand how to execute and manage your tests. Here are some key annotations,

1. **@Test** : Marks a method as a **test case**. JUnit executes methods annotated with @Test as individual tests.

2. **@BeforeEach** : Marks a method to run **before each test method**. Used for setup tasks like initializing objects.

3. **@AfterEach** : Marks a method to run **after each test method**. Used for cleanup tasks like releasing resources.

4. **@BeforeAll** : Marks a method to run **once before all tests** in the class. Must be **static**[9].

5. **@AfterAll** : Marks a method to run **once after all tests** in the class. Must be **static**.

6. **@DisplayName** : Provides a custom name for a **test class** or **method**, making test reports more readable.

7 **@Disabled** : Temporarily **disables** a test method or class.

JUnit provides a rich set of annotations beyond the basics, such as **@RepeatedTest**, **@ParameterizedTest**, **@TestFactory**, and **@Nested**, to handle advanced testing scenarios. These annotations enable features like repeating tests, running tests with multiple inputs, creating dynamic tests, and organizing tests into nested groups. For a comprehensive list and detailed usage, explore the official JUnit documentation.

3.2 Assertions

An *assertion* is a statement that enables you to test your assumptions about your program. They allow you to **validate the expected behavior** of your code by comparing actual results with expected outcomes. If an assertion fails, the test is marked as failed, and JUnit provides detailed feedback about the mismatch. Here are some common assertions,

1. **assertEquals(expected, actual)** : Checks if the **expected value** matches the **actual value**.

2. **assertTrue(condition)** : Verifies that a condition is **true**.

9 Why should the method be static ? It is to do with lifetime of tests and the test class. We'll explore this later in this labsheet.

3. **assertNull(object)** : Checks if an object is **null**.

4. **assertAll(groupedAssertions)** : Groups **multiple assertions** and ensures all are executed, even if some fail.

5. **assertArrayEquals(expectedArray, actualArray)** : Compares **two arrays** to check if they have the same **length** and same elements in the same **order**. Works for all types of arrays.

Like annotations, JUnit provides a **rich set of assertions** to handle diverse testing scenarios. For example, **assertThrows** ensures a specific exception is thrown, while **assertTimeout** verifies that code completes within a specified time limit. Additionally, **assertIterableEquals** compares the contents of iterables, and **assertLinesMatch** checks if two lists of strings match line by line. Feel free to go through the JUnit documentation to read more about different assertions.

Let's understand these concepts by going through an example.

3.3.1 Example

We will explore a **String Utility class** called **StringUtils**. This class provides helpful methods for common string operations, such as checking if a string is a palindrome (**isPalindrome**), reversing a string (**reverse**), and counting the number of vowels in a string (**countVowels**). It also handles edge cases, like null inputs.

The **StringUtilsTest** class demonstrates how to test these methods using JUnit annotations like **@Test** and **@DisplayName**, along with assertions like **assertTrue**, **assertFalse**, **assertEquals**. This example shows how to write focused tests for common string operations.

```java
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class StringUtilsTest {

    @Test
    @DisplayName("Test if string is palindrome")
    void testIsPalindrome() {
        StringUtils stringUtils = new StringUtils();
        assertTrue(stringUtils.isPalindrome("madam")); // Check palindrome
        assertFalse(stringUtils.isPalindrome("hello")); // Check non-
palindrome
    }

    @Test
    @DisplayName("Test reverse of a string")
    void testReverseString() {
        StringUtils stringUtils = new StringUtils();
        assertEquals("olleh", stringUtils.reverse("hello")); // Verify reverse
        assertEquals("", stringUtils.reverse("")); // Verify empty string
    }

}
```

3.3.2 Exercise – 1

Write JUnit tests for the **countVowels** method in the **StringUtils** class. This method counts the number of vowels (a, e, i, o, u) in a string, ignoring case. Validate the method by testing it against various inputs, such as strings with only vowels, no vowels, mixed characters, uppercase and lowercase vowels, an empty string, and null. Ensure the test method has a descriptive display name and covers all edge cases. *(Check the StringUtilities project)*

Write your test methods in the **StringUtilsTest.java** file.

(*BONUS PROBLEM: Write a test case to verify that the reverse method in the StringUtils class throws an IllegalArgumentException when the input string is null. Ensure the test checks both the exception type and its message ("Input cannot be null"). Hint* [10] *in the footnotes.*)

3.3.3 Exercise – 2

The **UtilityFunctions** project contains two utility classes: **MathUtils** for mathematical operations and **DateUtils** for date-related tasks. Following is the description of each of the classes:

1. **MathUtils**

- **int factorial(int n)** : The **factorial** method calculates the factorial of a non-negative integer n. It returns the product of all positive integers less than or equal to n. For edge cases, it returns 1 if n is 0 or 1, and throws an **IllegalArgumentException** for negative inputs.
- **boolean isPrime(int n)** : The **isPrime** method checks if a number n is prime. A prime number is greater than 1 and has no divisors other than 1 and itself. It returns false for numbers less than or equal to 1 and efficiently handles large values of n.
- **int[] generateFibonacci(int n)** : The **generateFibonacci** method generates the first n numbers in the Fibonacci sequence, starting with 0 and 1. It returns an array of integers representing the sequence. For edge cases, it returns an empty array if n is 0 and throws an **IllegalArgumentException** for negative inputs. While the recursive implementation works well for smaller inputs, it becomes highly inefficient for larger values of n due to repeated calculations.
- **int gcd(int a, int b)** : The **gcd** method computes the greatest common divisor (GCD) of two integers a and b. It uses the **Euclidean algorithm** to find the largest integer that divides both a and b without a remainder. If b is 0, it returns a.

2. **DateUtils**

- **boolean isLeapYear(int year)** : The **isLeapYear** method checks if a given year is a leap year. A leap year is divisible by 4 but not by 100, unless it is also divisible by 400. It correctly handles edge cases like the year 2000 (leap year) and 1900 (not a leap year).
- **int getDayOfWeek(int year, int month, int day)** : The **getDayOfWeek** method returns the day of the week for a given date, where 0 represents Sunday, 1 represents Monday, and so on. It throws a **DateTimeException** for invalid dates, such as February 30th.

10 You can use assertThrows to check if the exception is thrown, and IllegalArgumentException.class along with .getMessage() to verify the exception type and message.

- **String formatDate(int year, int month, int day)** : The **formatDate** method formats a date as a string in the "**YYYY-MM-DD**" format. It uses the LocalDate class to validate and format the date. If the date is invalid, it throws a **DateTimeException**.

Your task is to write **unit tests** for the methods in the MathUtils and DateUtils classes. First, create the test files **MathUtilsTest.java** and **DateUtilsTest.java** in the appropriate packages under the **src/test/java** directory. Follow the **proper package naming convention** and **project structure.** For each method, write test cases to verify its correctness. Ensure your tests cover:

1. **Valid inputs** and expected outputs.
2. **Edge cases** (e.g., smallest/largest inputs, empty inputs).
3. ***Error handling*** *(e.g., invalid inputs).* (*Bonus Question!*)

Use **assertions** like assertEquals, assertTrue, assertArrayEquals, and assertThrows to validate the results. Provide descriptive test names using **@DisplayName**.

3.3.3.1 – Exercise 2.1

In the previous task, you wrote tests to validate the methods in the MathUtils class. Now, add the following test case to the **MathUtilsTest** file to test the performance of the generateFibonacci method using **assertTimeout**:

```
 @Test
    @DisplayName("Test recursive Fibonacci for n = 50 with timeout (should
fail)")
    void testRecursiveFibonacciLarge() {
        MathUtils mathUtils = new MathUtils();
        assertTimeout(Duration.ofMillis(100), () -> {
            mathUtils.generateFibonacci(50); // This will likely exceed the
timeout
        });
    }
```

The **assertTimeout** function in JUnit verifies whether a block of code completes execution within a specified time limit. If the code takes longer than the allowed time, the test fails. This is particularly useful for testing the performance of methods and ensuring they meet efficiency requirements.

In the provided test case, the **generateFibonacci** function is tested with a timeout of **1 second**. The test will fail because the recursive implementation is inefficient for **large inputs**, causing it to exceed the time limit.

Your task is to **optimize the generateFibonacci method** by replacing the recursive approach with an iterative one. Once optimized, re-run the test to confirm it completes within the time limit and passes successfully.

3.4 Test Lifecycle and Hooks

In JUnit, tests follow a specific **lifecycle** that determines the order in which test methods are executed and how resources are managed.

One key aspect of this lifecycle is that JUnit **creates a new instance of the test class before running each test method**. This ensures that tests are **isolated** from one another, preventing side effects or shared state between tests. By maintaining this isolation, JUnit helps you write reliable and independent tests.

```java
public class SharedStateTest {

       // Instance variable shared between tests
       private int sharedValue = 0;

       @Test
       void testA() {
           sharedValue = 10; // Test A modifies the shared variable
           assertEquals(10, sharedValue); // This assertion passes
       }

       @Test
       void testB() {
           // Test B relies on the shared variable, expecting it to be 10
           assertEquals(10, sharedValue); // This assertion fails
       }
   }
```

In this example, we have a test class with two test methods, **testA** and **testB**, and an instance variable **sharedValue** that is shared between them. In testA, the variable sharedValue is set to 10, and an assertion verifies that the value is correct. In testB, another assertion checks if sharedValue is still 10. However, this test will fail because JUnit creates a **new instance of the test class** before running each test method. This means that sharedValue is reset to 0 for testB, even though testA modified it.

This behavior highlights why sharing state between tests is a **bad idea**. JUnit enforces **test isolation** by creating a new instance of the test class for each test method, ensuring that tests do not interfere with one another.

While JUnit's approach of creating a new instance for each test ensures **test isolation**, it also means that you cannot rely on instance variables to share state between tests. Instead, JUnit provides **hooks**—special methods annotated with **@BeforeEach**, **@AfterEach**, **@BeforeAll**, and **@AfterAll**—to manage setup and teardown operations at specific points in the test lifecycle. These hooks allow you to initialize resources, reset state, and clean up efficiently, without violating the principle of test isolation.

1. **@BeforeEach** : This method runs **before each test**. It's used to set up test-specific resources or reset the state, ensuring each test starts with a clean environment. For example, initialize objects or prepare test data here.

2. **@AfterEach** : This method runs **after each test**. It's used to clean up resources or reset the state, ensuring no leftover data affects other tests. For example, close files or clear temporary data here.

3. **@BeforeAll** : This method runs **once before all tests**. It's ideal for expensive, shared setup tasks like initializing a database or starting a server. Since it runs only once, it improves efficiency.

4. **@AfterAll** : This method runs **once after all tests**. It's used to release shared resources or perform final cleanup, such as closing a database connection or stopping a server.

3.4.1 Example

This example simulates a scenario where you're testing a **DatabaseService** that requires setup and teardown operations.

```
public class DatabaseServiceTest {

    private static DatabaseService databaseService;

    @BeforeAll
    static void setUpClass() {
        // Initialize the database connection once before all tests
        databaseService = new DatabaseService();
        databaseService.connect();
        System.out.println("Database connected: BeforeAll");
    }

    @BeforeEach
    void setUp() {
        // Clear the database before each test to ensure a clean state
        databaseService.clear();
        System.out.println("Database cleared: BeforeEach");
    }

    @Test
    void testInsertRecord() {
        // Test inserting a record into the database
        databaseService.insert("User1");
        assertEquals(1, databaseService.getRecordCount());
        System.out.println("Test: Inserted record");
    }

    @Test
    void testDeleteRecord() {
        // Test deleting a record from the database
        databaseService.insert("User2");
        databaseService.delete("User2");
        assertEquals(0, databaseService.getRecordCount());
        System.out.println("Test: Deleted record");
    }

    @AfterEach
    void tearDown() {
        // Log the state of the database after each test
        System.out.println("Records in database: " +
databaseService.getRecordCount());
        System.out.println("Database state logged: AfterEach");
    }

    @AfterAll
    static void tearDownClass() {
        // Close the database connection after all tests
        databaseService.disconnect();
        System.out.println("Database disconnected: AfterAll");
    }
}
```

A few things to keep in mind,

- The **@AfterAll** and **@BeforeAll** method will always be static. Why ? JUnit creates a **new instance of the test class** for each test method to ensure **test isolation**. This means each test runs in its own isolated environment. If **@BeforeAll** and **@AfterAll** were not static, they would require an instance of the test class to run. However, at the time these methods are executed, no test instances exist yet (for @BeforeAll) or all test instances have been discarded (for @AfterAll). Making them static allows them to run independently of test instances.
- In the **3.3.1 example** (and all other problems we've worked on so far), how can we avoid manually initializing the class in every test ? By using the **@BeforeEach** method. This hook automatically initializes the class *(the one which is being tested)* for us before each test, ensuring that every test starts with a fresh instance. This approach not only reduces code duplication but also maintains **test isolation**, as each test runs with its own independent instance of the class.
- The full **JUnit**[11] **Life Cycle** has 3 phases:

  I. **Setup phase**: Setup the test infrastructure (i.e., test fixtures). Two levels of setup are available:
     i. **Class-Level** Setup (using @BeforeAll annotation): such as a database connection.
     ii. **Test-Level** Setup (using @BeforeEach annotation): before running each test case.

  II. **Test execution**: Run the tests and verifying the result (using @Test annotation).

  III. **Clean up phase**: Perform cleanup after all post-test executions. Similar to setup phase, there is a corresponding **class-level** clean up (using @AfterAll annotation) and **test-level** cleanup (using @AfterEach annotation) after each test execution.

11 A good resource for learning more about JUnit5 , it's earlier version and other testing frameworks.

3.4.2 Exercise – 3

You are tasked with writing unit tests for a **BankAccount** class, inside the **Bank project**, that simulates a simple bank account. A brief description of each method and constructor is given below :

1. **BankAccount()** : Initializes the bank account with a specified starting balance.

2. **deposit(double amount)** : This method adds the specified amount to the account balance. The amount must be positive; otherwise, the balance remains unchanged.

3. **withdraw(double amount)** : This method deducts the specified amount from the account balance. If the amount is greater than the current balance, the method throws an **InsufficientFundsException**.

4. **getBalance()** : This method returns the current balance of the account.

The **InsufficientFundsException** is a custom exception that is thrown when a withdrawal attempt exceeds the available balance. It ensures that the account balance cannot go negative.

Your task is to write unit tests for the **BankAccount** class. Create a test file named **BankAccountTest.java** in the appropriate package under **src/test/java**, following standard naming conventions and project structure. Write test cases for each method, ensuring coverage of:

1. **Valid inputs and expected outputs** (e.g., deposit/withdraw with valid amounts).
2. **Edge cases** (e.g., smallest/largest amounts, boundary conditions).
3. ***Error handling*** *(e.g., withdrawing more than the balance or depositing negative amounts). (Bonus!)*

Use JUnit **hooks** to:

- Initialize a BankAccount object with a balance of 1000.0 before each test (**@Before**).
- Print the final balance after each test (**@After**).
- Print messages at the start (**@BeforeClass**) and end (**@AfterClass**) of the test suite.

Ensure your tests are comprehensive, well-structured, and use assertions to validate outcomes. Provide descriptive test names using **@DisplayName**.

3.5 Parameterized tests

Let's say we want to test a function which checks weather a number is even or not. Below is the test for it,

```java
public class EvenNumberTest {

    @Test
    void testIsEven2() {
        assertTrue(isEven(2));
    }

    @Test
    void testIsEven4() {
        assertTrue(isEven(4));
    }

    @Test
    void testIsEven6() {
        assertTrue(isEven(6));
    }

    @Test
    void testIsEven8() {
        assertTrue(isEven(8));
    }

    @Test
    void testIsEven10() {
        assertTrue(isEven(10));
    }

}
```

When you write a separate test method for each input or scenario, you end up duplicating code and introducing unnecessary complexity. Each test method is nearly identical, with the only difference being the **input** value. This repetition violates the **DRY (Don't Repeat Yourself)** principle, making the code harder to maintain and more prone to errors. For instance, if you need to update the test logic—such as modifying an assertion or adding new checks—you'll have to make the same change across every single test method. Additionally, a test class filled with dozens or even hundreds of nearly identical methods becomes cluttered and difficult to read, obscuring the clarity of what's being tested and making it harder to navigate and understand the test suite as a whole.

Why didn't we put all the assertions into one test ? Store all the inputs into an array and use a for loop to go through each input ? When you use a **for loop** to run multiple assertions in a single test method, all assertions are treated as part of **one test case**. If one assertion fails, the entire test stops, and the remaining assertions won't execute, leaving you unaware of whether the other inputs would have passed or failed. This **lack of isolation** makes it harder to identify and fix issues, as you don't get a complete picture of the test results. Additionally, debugging becomes more challenging because when a test fails, it's difficult to pinpoint the exact input that caused the failure, as all assertions are grouped together in a single test method.

A better solution is **parameterized tests**. Unlike separate test methods for each input, they eliminate duplication, making tests cleaner and easier to maintain. Each input is an **isolated test case**—if one fails, others still run, providing full feedback and making debugging easier, as failures are tied to specific inputs. They improve readability by organizing inputs and outcomes in one place and simplify maintenance—updating the input list adjusts test cases. Here's the updated code:

```java
@ParameterizedTest
@ValueSource(ints = {2, 4, 6}) // Input values to test
void testIsEven(int number) {
    assertTrue(isEven(number), number + " is not even");
}
```

We use the **@ParameterizedTest** annotation to tell JUnit that this function is a parameterized test. The **@ValueSource** annotation provides the inputs for the test case—in this example, we supply integer values *(ints)*. Similarly, we can provide values of other data types, such as *doubles* for decimals or *strings* for text. The **testIsEven** function now takes a parameter, number, which receives each value from the provided inputs. For each input, it runs the **assertTrue** assertion to verify if the number is even. If the assertion fails for any input, JUnit reports the failure, clearly indicating which input caused the issue.

When we need to test methods with **multiple inputs and an expected output**, we use the **@CsvSource** annotation. This annotation allows us to provide **comma-separated values (CSV)** as inputs, where each line represents a set of inputs and the corresponding expected output. For example, if we're testing a method that adds two numbers, we can use @CsvSource to supply pairs of inputs and their expected sums, like this:

```java
@ParameterizedTest
@CsvSource({
    "2, 3, 5",  // Input: 2, 3 → Expected Output: 5
    "4, 5, 9",  // Input: 4, 5 → Expected Output: 9
    "10, -2, 8" // Input: 10, -2 → Expected Output: 8
})
void testAdd(int a, int b, int expected) {
    assertEquals(expected, utility.add(a, b));
}
```

Here, the test method **testAdd** takes three parameters: a, b, and expected. The **@CsvSource** annotation provides the input values and expected outputs, and the assertEquals statement verifies that the method produces the correct result for each set of inputs.

3.5.1 Exercise – 4

You are building a **PasswordValidator** class that checks if a password meets the following criteria:

1. At least 8 characters long.
2. Contains at least one uppercase letter.
3. Contains at least one lowercase letter.
4. Contains at least one digit.
5. Contains at least one special character (e.g., !@#$%^&*).

Your task is to write **parameterized tests** for **isValidFunction** in the PasswordValidator class using JUnit. Use the **@CsvSource** annotation to test passwords that **meet all the criteria** as well as those that **fail one or more criteria**. The PasswordValidator class is located in the **Validator project** inside the src/main/java directory. Create your test files in the src/test/java directory, ensuring they are in the **same package** as the main files.

3.5.2 Exercise – 5

In addition to validating whether a password meets the basic criteria, you are now tasked with evaluating the **strength** of a password. The strength is determined as follows:

1. **Strong**: The password meets all the criteria (length, uppercase, lowercase, digit, special character).
2. **Medium**: The password meets the length requirement and has at least one uppercase and one lowercase letter, but lacks a digit or special character.
3. **Weak**: The password does not meet the above conditions.

Your task is to write **parameterized tests** for **getPasswordStrength** function in the PasswordValidator class using JUnit. Use the @CsvSource annotation to test passwords that **meet all the criteria** as well as those that **fail one or more criteria**. For example, you can test passwords that are too short, lack uppercase or lowercase letters, miss digits, or don't include special characters.

3.6 Repeated Tests

Imagine you have a method that simulates rolling a six-sided die. The method should return a random number between 1 and 6. How do you ensure that:

1. The method always returns a valid number (between 1 and 6)?
2. The method behaves consistently over multiple runs?

If you write a single test, it might pass once but fail in subsequent runs due to randomness. This is where **repeated tests** come in handy.

Here's a method that simulates rolling a die :

```java
public int rollDie() {
    return (int) (Math.random() * 6) + 1;
}
```

If you write a single test, it might look like this :

```java
@Test
void testRollDie() {
    int result = rollDie();
    assertTrue(result >= 1 && result <= 6, "Result should be between 1 and
6");
}
```

Issues with This Approach:

1. **Randomness**: The test might pass once but fail in subsequent runs if the random number falls outside the expected range (though this is unlikely, it's still possible).
2. **Incomplete Validation**: A single test run doesn't provide enough confidence that the method behaves correctly over multiple executions

This is where **repeated tests** come in. By running the same test method multiple times, they are perfect for **testing randomness** (e.g., generating random numbers), **ensuring consistency** (e.g., verifying repeated results), and **simulating stress testing** (e.g., checking reliability under repeated use).

Here's the same test rewritten using **repeated tests**:

```java
@RepeatedTest(value = 10, name = "Test {currentRepetition} of
{totalRepetitions}")
void testRollDie() {
    int result = rollDie();
    assertTrue(result >= 1 && result <= 6, "Result should be between 1 and
6");
}
```

The **@RepeatedTest** annotation tells JUnit that the method is a **repeated test**, meaning it will run multiple times. The **value** attribute specifies how many times the test should be executed, while the **name** attribute (optional) allows you to customize the display name for each repetition. You can use **placeholders** like {currentRepetition} and {totalRepetitions} in the name attribute to dynamically include the current repetition number and the total number of repetitions in the test output.

3.6.1 Exercise – 6

You are building a method that shuffles the characters of a string randomly. Your task is to write a **repeated test** to ensure that:

1. The shuffled string has the **same length** as the original string.
2. The shuffled string contains **all the characters** of the original string.

The **StringShuffler** class is located in the **Shuffler project** inside the src/main/java directory. Create your test files in the **src/test/java directory**, ensuring they are in the **same package** as the main files.

3.6.2 Exercise – 7

Extend the **shuffled string problem** by writing tests for a **shuffleStringDerangement** method that shuffles a string but ensures that **no character remains in its original position**. This is known as a **derangement** of the string. Your task is to:

1. Write a **repeated test** to validate the behavior of the new method.
2. Ensure that the shuffled string:
   - Has the **same length** as the original string.
   - Contains **all the characters** of the original string.
   - Has **no character in its original position**.

3.7 Nested tests

Nested tests is a powerful feature in JUnit 5 that allows you to group related tests together in a **hierarchical** structure. This is useful for:

- Organizing tests for complex classes or methods.
- Grouping tests by functionality, scenarios, or edge cases.
- Improving readability and maintainability of test code.

Let's say we have a **Calculator** class with methods for addition, subtraction, multiplication, and division. We can use nested tests to group tests by operation.

```java
public class CalculatorTest {

    private Calculator calculator = new Calculator();

    @Nested
    class AdditionTests {
        @Test
        void testAddPositiveNumbers() {
            assertEquals(5, calculator.add(2, 3));
        }

        @Test
        void testAddNegativeNumbers() {
            assertEquals(-1, calculator.add(2, -3));
        }
    }

    @Nested
    class SubtractionTests {
        @Test
        void testSubtractPositiveNumbers() {
            assertEquals(2, calculator.subtract(5, 3));
        }

        @Test
        void testSubtractNegativeNumbers() {
            assertEquals(8, calculator.subtract(5, -3));
        }
    }

    @Nested
    class MultiplicationTests {
        @Test
        void testMultiplyPositiveNumbers() {
            assertEquals(6, calculator.multiply(2, 3));
        }

        @Test
        void testMultiplyNegativeNumbers() {
            assertEquals(-6, calculator.multiply(2, -3));
        }
    }

    @Nested
    class DivisionTests {
        @Test
        void testDividePositiveNumbers() {
            assertEquals(2.0, calculator.divide(6, 3));
        }

        @Test
        void testDivideByZero() {
            assertThrows(IllegalArgumentException.class, () →
calculator.divide(6, 0));
        }
    }
}
```

The **@Nested** annotation is used to mark inner classes as **nested test classes**, allowing you to group related tests together in a hierarchical structure. By defining these inner classes, you can organize your tests logically, making it easier to navigate and understand the test suite.

4.1 Exercise – 8

You are building a **TreasureHunter** class that helps adventurers navigate a treasure map. The map is represented as a grid, and the adventurer's position is tracked using coordinates (x, y). Your task is to write tests for the TreasureHunter class to ensure it works correctly. The class has the following functionalities:

1. Move the adventurer in four directions: **North**, **South**, **East**, and **West**.
2. Check if the adventurer has found the treasure (reached a specific location).
3. Validate if a move is within the bounds of the map.
4. Simulate random moves to test the adventurer's path.

Your task is to test the following scenarios :

Test the following scenarios:

1. **move**:
   - Move the adventurer in all four directions and verify the new position. *(use paramterized test)*
   - *Test invalid moves (e.g., moving outside the map bounds). (using assertThrows())*
2. **hasFoundTreasure**:
   - Verify if the adventurer has found the treasure at specific locations.
3. **IsValidMove**: *(use parameterized test)*
   - Test if a move is valid based on the current position.
4. **SimulateRandomMove**: *(use repeated test)*
   - Simulate random moves and ensure they are valid.

Use the **@Nested** annotation to group tests for the move, hasFoundTreasure, and isValidMove methods into **nested test classes.** Leverage hooks like **@BeforeEach** to initialize the TreasureHunter object before each test, ensuring a clean state for every test case. The TreasureHunter class is located in the **Quest project** inside the src/main/java directory. Create your test files in the **src/test/java directory**, ensuring they are in the **same package** as the main files to maintain proper project structure and accessibility.

4.2 Exercise- 9

You are tasked with building a **Library Management System** from scratch. The system will allow users to manage books, track their availability, and view book details. The system consists of two main classes:

1. **Book class** : This class represents a single book in the library system. It stores details such as the title, author, availability, and due date. The class provides methods to borrow a book, return a book, and display book details.

- **Attributes** : (*Ensure all attributes are private and provide* ***getter and setter methods*** *where necessary.)*
  - **title (String)**: The title of the book.
  - **author (String)**: The author of the book.
  - **isAvailable (boolean)**: Indicates whether the book is available for borrowing.
  - **dueDate (String)**: The due date for the book (if borrowed).
- **Methods** :
  - **Book()**: *Constructor that Initializes isAvailable to True value and dueDate to NULL value.*
  - **borrowBook(String dueDate)**: Marks the book as unavailable and sets the due date.
  - **returnBook()**: Marks the book as available and clears the due date.
  - **displayDetails()**: Prints the book's details, including the title, author, availability, and due date (if applicable).
  - **readInput()**[12]: Takes user input by reading values for the book's title, author, and availability.

2. **Library Class** : This class manages a collection of books in the library system. It provides functionality to add books, find books by title, and display all books in the library.

- **Attributes**: *(Ensure all attributes are private and provide* ***getter and setter methods*** *where necessary.)*
  - **books (Book[])**: An array to store books in the library.
  - **count (int)**: Tracks the number of books added to the library.
- **Methods** :
  - ***Library(int size)****: Constructor that initializes the books array with the specified size and sets count to 0.*
  - **addBook(Book book)**: Adds a book to the library if there is space
  - **findBook(String title)**: Searches for a book by its title and returns the book if found.
  - **displayAllBooks()**: Displays details of all books in the library.

12 How does one test a function, that uses external dependencies (in this case I/O) ? Read this.

Your task is to implement the **Book** and **Library** classes, ensuring they work correctly, and write comprehensive unit tests using JUnit. Make use of **JUnit hooks** wherever necessary to set up and clean up the test environment. Test edge cases, such as:

- Borrowing an already borrowed book.
- Returning a book that is already available.
- Adding a book to a full library.
- Searching for a book that doesn't exist.

Use **@ParameterizedTest** to test methods with multiple inputs and Group related tests logically using the **@Nested** annotation. Proper package structure and naming conventions should be followed, with test classes placed in the same package as the main classes but under the **src/test/java** directory.