**Module 2 : TDD using JUnit5 and Mockito**

**JUnit\_Basic Testing Exercises**

**Exercise 1: Setting Up JUnit**

**Scenario:** You need to set up JUnit in your Java project to start writing unit tests.

**Description**: Before writing any unit tests, we need to correctly set up JUnit in your Java project. JUnit is one of the most widely used testing frameworks in Java and is essential for test-driven development (TDD).

**Objective:**

* Create a Java project.
* Configure JUnit as a test dependency.
* Prepare the environment to start writing test cases.

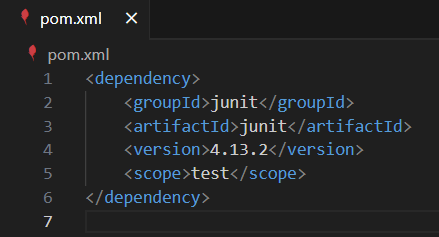
**Solution:**

**Step 1: Create a New Java Project**

* Open an IDE ( say VS Code )
* Create a new project called **JUnitDemo** or any name of your choice.

**Step 2: Add JUnit Dependency**

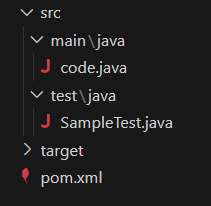
* Using **Maven**, include JUnit in your **pom.xml** file under **<dependencies>:**



**Step 3: Create a Sample Test Class**

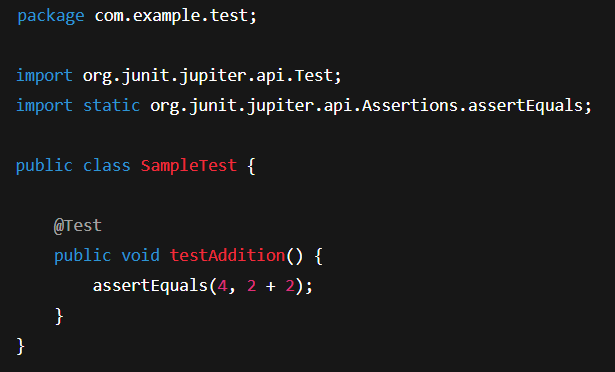
* Create a folder structure like **src/main/java** for code and **src/test/java** for test files.

The Folder Structure should look like this:



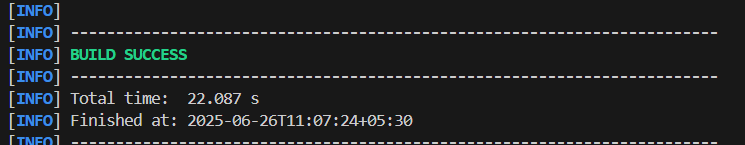
* In **src/test/java**, create a new class **SampleTest.java** with:

**Test Code**:

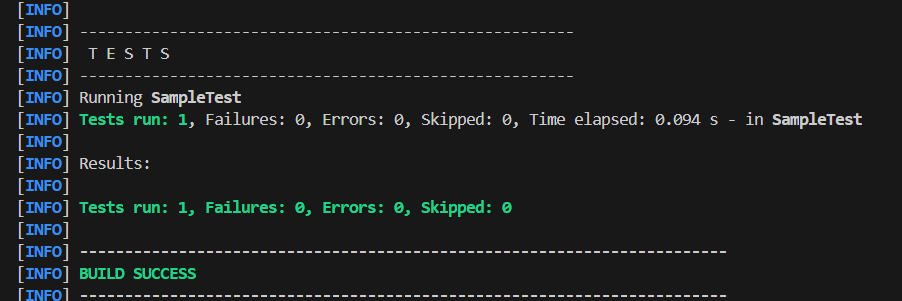


**Step 4: Run the below commands to execute**

1. **mvn clean compile → ( It must show a build success message )**

****

1. **mvn test → ( It must show a build success message )**

****

**Exercise 3: Assertions in JUnit**

**Scenario:** You need to use different assertions in JUnit to validate your test results**.**

**Description**: Assertions are the backbone of unit testing. We need to validate the outputs of the methods using different types of assertions provided by the JUnit framework.

**Objective:**

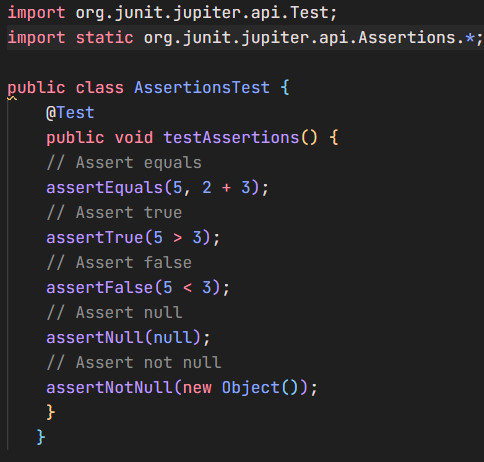
* Understand and use various assertion methods like
  + **assertEquals**
  + **assertTrue**
  + **assertFalse**
  + **assertNull**
  + **assertNotNull.**

**Common Assertions and there use:**

| **METHOD** | **DESCRIPTION** |
| --- | --- |
| **assertEquals()** | Checks if two values are equal |
| **assertTrue()** | Asserts that a condition is true |
| **assertFalse()** | Asserts that a condition is false |
| **assertNull()** | Asserts that a value is null |
| **assertNotNull()** | Asserts that a value is not null |

**Solution:**

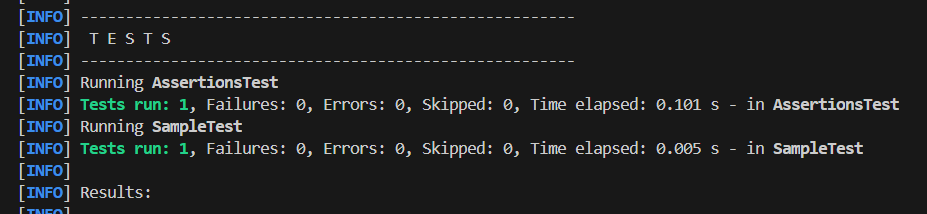
**Code:**

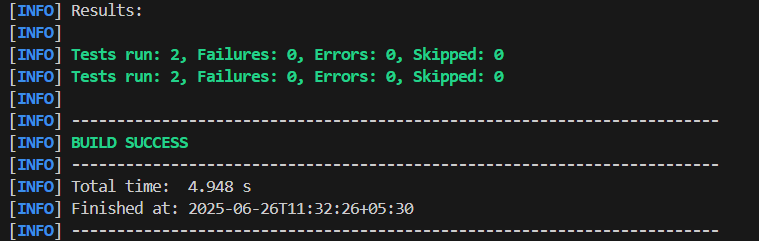
****

**Run the following commands execute:**

* **mvn clean compile → ( It must show a build success message )**
* **mvn clean test→ ( It must show a build success message )**

**Output:**

****

****

**Exercise 4: Arrange-Act-Assert (AAA) Pattern, Test Fixtures, Setup and**

**Teardown Methods in JUnit**

**Scenario:** You need to organize your tests using the Arrange-Act-Assert (AAA) pattern and use setup and teardown methods.

**Objective:**

* Use the **Arrange-Act-Assert (AAA) pattern.**
* Utilize **@Before** and **@After** annotations to manage test setup and teardown.

The **Arrange-Act-Assert (AAA) pattern:**

* **Arrange** - Initialize variables, objects, test data.
* **Act** - Call the method/function you want to test.
* **Assert** - Verify the result/output using assertions.

**Setup and Teardown**

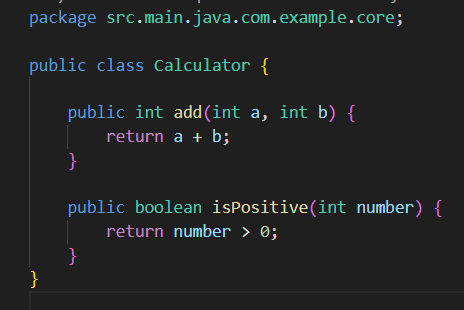
* **@Before**: Runs before every test method – used for common setup.
* **@After**: Runs after every test method – used for cleanup activities.

**Solution:**

**Step 1:** Create a **Core Class** file 9 (say **Calculator.java**) and put it in the directory:

**src/main/java/com/example/core/Calculator.java**

with the following code:



**Step 2:** Create a **JUnit Test Class** File ( say **CalculatorTestWithSetupTest.java** ) in the following directory:

**src/test/java/com/example/test/CalculatorTestWithSetupTest.java**

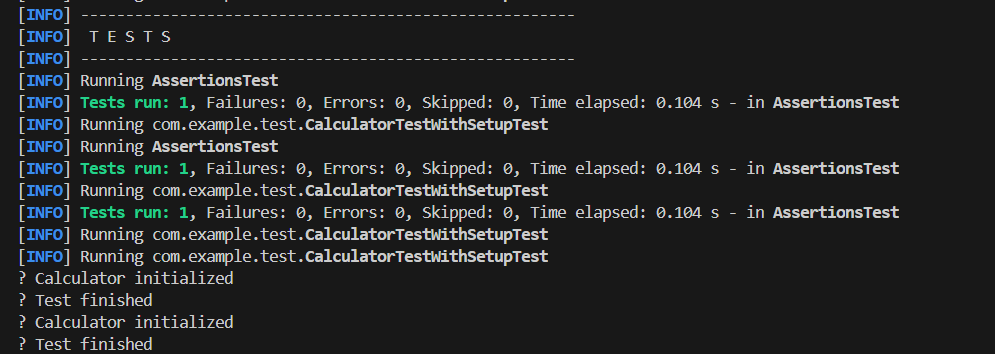
**with the following code:**

****

**Step 3: Run the Tests with the following command:**

* **mvn clean test**

**Step 4: Output**

****

**Mockito Exercises**

**Exercise 1: Mocking and Stubbing**

**Scenario:** You need to test a service that depends on an external API. Use Mockito to mock the external API and stub its methods.

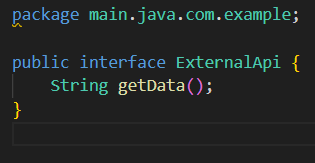
**Solution:**

**Step 1**: **Create a mock object for the external API**

**Use Mockito.mock()**

**Step 2: Stub the methods to return predefined values.**

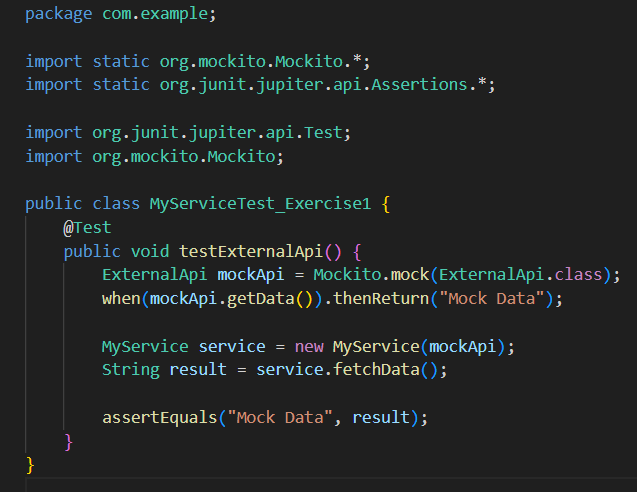
**Use getData() to return a mock data.**

****

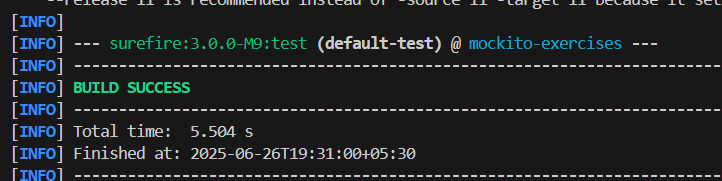
**Step 3: Write a test case that uses the mock object.**

Use **JUnit** that calls the method **fetchData()** from your service and checks the returned value using **assertEquals**.

**Sample Test Code:**



**Output:**

****

**Exercise 2: Verifying Interactions**

**Scenario:** You need to ensure that a method is called with specific arguments.

**Objective:** to verify that your service method interacts correctly with an external dependency, specifically that a method (e.g., getData()) is called on a mocked object. This helps ensure your business logic is actually using the dependency as expected.

**Solution:**

**Step 1: Create a mock object. [ for the external API ]**

**Use Mockito.mock()**

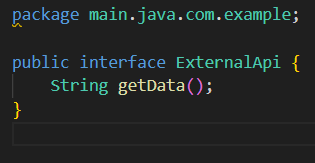
**Step 2: Call the method with specific arguments. [ Internally uses the external API ]**

**Step 3: Use verify() to confirm that the expected method was indeed called.**

**Purpose of Each Component**

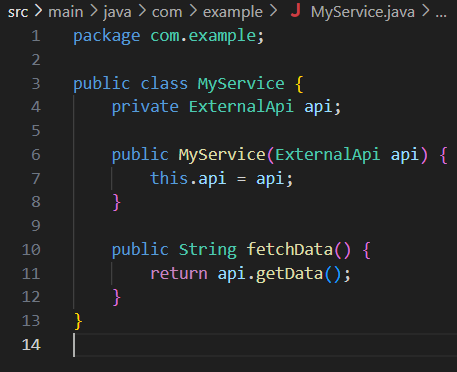
* **ExternalApi:** An interface that simulates an external system.
* **MyService:** Business logic that uses the ExternalApi.
* **MyServiceExercise2Test:** JUnit test class that verifies the method interaction using Mockito.

**Step 4: Create an ExternalAPI.java file as follows:**

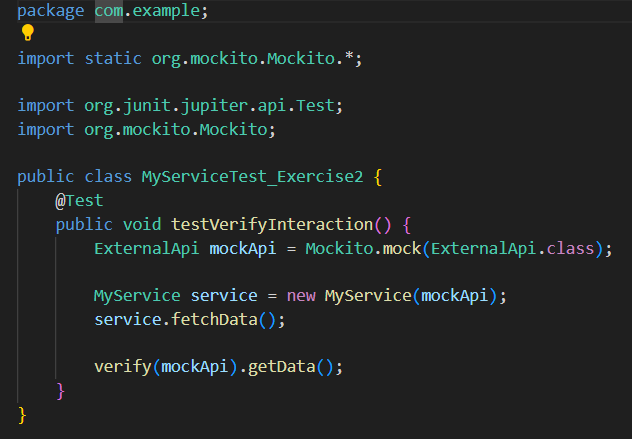
****

**Step 5: Create a service.java file as follows:**

**MyService.java** is your own business logic class, which depends on ExternalApi to get some data. You don't want to call the real external API during tests — so you'll inject a mock of ExternalApi using Mockito.



**Step 6: Finally creating the test file to execute:**

****

**Key Concepts Demonstrated**

* Use of Mockito.mock() to create a test double.
* Dependency Injection: passing the mock into MyService.
* Use of Mockito.verify() to ensure interaction occurred.