# TDD using JUnit5 and Mockito

## Exercise 1: Setting Up JUnit

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

Steps:

1. Create a new Java project in your IDE (e.g., IntelliJ IDEA, Eclipse).
2. Add JUnit dependency to your project. If you are using Maven, add the following to your
   pom.xml:
3. Create a new test class in your project.

### Step 1: Create a Java Project

- Open your IDE (e.g., IntelliJ IDEA or Eclipse).
- Create a new **Maven** project.
- Set the project name as **JUnitExample**.

### Step 2: Add JUnit Dependency

- Open **pom.xml** file of your Maven project.
- Add the following dependency inside the <dependencies> tag:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.13.2</version>
  <scope>test</scope>
</dependency>
```

- Save the `pom.xml` file. Maven will download the required JUnit library.

## Step 3: Create a Java Class to be Tested

Create a class named `Calculator.java` in `src/main/java`.

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }
}
```

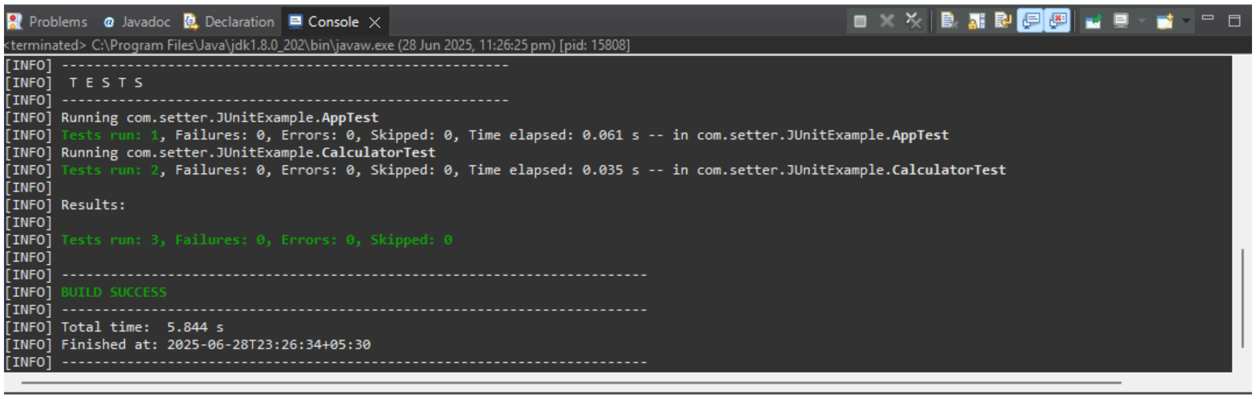
## Step 4: Create a JUnit Test Class

Create a test class named `CalculatorTest.java` in `src/test/java`.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calc = new Calculator();
        assertEquals(5, calc.add(2, 3));
    }

    @Test
    public void testSubtract() {
        Calculator calc = new Calculator();
        assertEquals(1, calc.subtract(3, 2));
    }
}
```

## Step 5: Run the Test

*Output:*



Problems Javadoc Declaration Console

```
<terminated> C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (28 Jun 2025, 11:26:25 pm) [pid: 15808]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.setter.JUnitExample.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.061 s -- in com.setter.JUnitExample.AppTest
[INFO] Running com.setter.JUnitExample.CalculatorTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.035 s -- in com.setter.JUnitExample.CalculatorTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  5.844 s
[INFO] Finished at: 2025-06-28T23:26:34+05:30
[INFO] ------------------------------------------------------------------------
```

## Exercise 3: Assertions in JUnit

### Scenario:

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

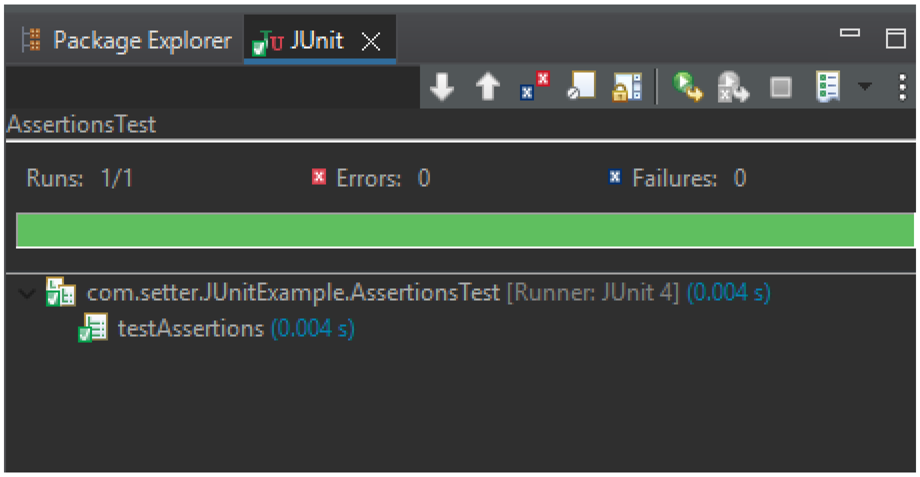
### Steps:

- Create a new Java class named `AssertionsTest` in your test folder.
- Add test methods using different assertions.
- Run the test class to verify all assertions pass.

### Solution Code:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class AssertionsTest {

    @Test
    public void testAssertions() {
        // Assert equals
        assertEquals(5, 2 + 3);

        // Assert true
        assertTrue(5 > 3);

        // Assert false
        assertFalse(5 < 3);

        // Assert null
        assertNull(null);

        // Assert not null
        assertNotNull(new Object());
    }
}
```

**Output:**



```
AssertionsTest
Runs: 1/1    Errors: 0    Failures: 0
com.setter.JUnitExample.AssertionsTest [Runner: JUnit 4] (0.004 s)
    testAssertions (0.004 s)
```

## Exercise 4: AAA Pattern, Test Fixtures, Setup and Teardown in JUnit

### Objective

To understand and implement the **Arrange-Act-Assert (AAA)** pattern in unit tests and utilize **@Before** and **@After** annotations in JUnit for test setup and teardown operations.

### Concepts Covered

- **AAA Pattern**: Structure for writing clean and understandable test cases.
  - **Arrange**: Set up test data and preconditions.
  - **Act**: Invoke the method being tested.
  - **Assert**: Verify the result.
- **Test Fixtures**: Shared setup data for multiple tests.
- **@Before**: Executed before each test method (test setup).
- **@After**: Executed after each test method (test teardown).

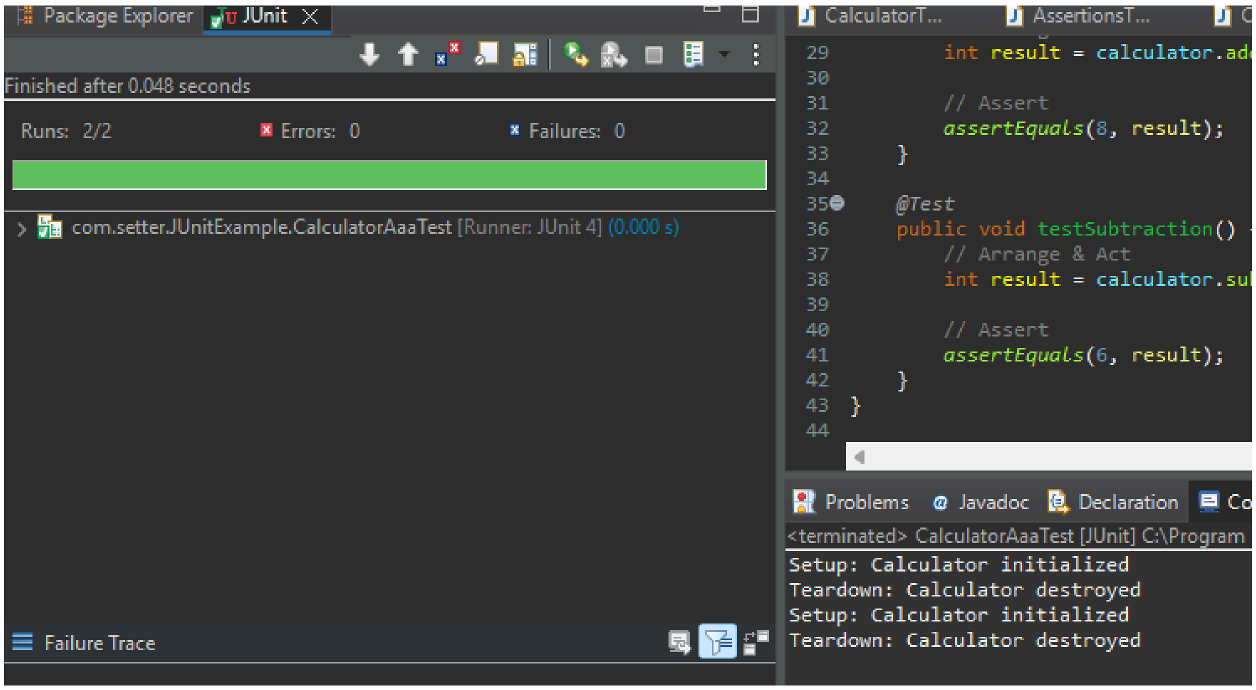
### Step 1: Create Logic Class `CalculatorAaa.java`

```
public class CalculatorAaa {
    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }
}
```

## Step 2: Create JUnit Test Class `CalculatorAaaTest.java`

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.After;
import org.junit.Test;

public class CalculatorAaaTest {

    private CalculatorAaa calculator;

    // This runs before every test method
    @Before
    public void setUp() {
        calculator = new CalculatorAaa();
        System.out.println("Setup: Calculator initialized");
    }

    // This runs after every test method
    @After
    public void tearDown() {
        calculator = null;
        System.out.println("Teardown: Calculator destroyed");
    }
    @Test
    public void testAddition() {
        // Arrange & Act
        int result = calculator.add(5, 3);

        // Assert
        assertEquals(8, result);
    }

    @Test
    public void testSubtraction() {
        // Arrange & Act
        int result = calculator.subtract(10, 4);

        // Assert
        assertEquals(6, result);
    }
}
```

## Output:



```
Finished after 0.048 seconds
Runs: 2/2      Errors: 0      Failures: 0
com.setter.JUnitExample.CalculatorAaaTest [Runner: JUnit 4] (0.000 s)
```

```
        int result = calculator.ad

        // Assert
        assertEquals(8, result);
    }

    @Test
    public void testSubtraction() 
        // Arrange & Act
        int result = calculator.su

        // Assert
        assertEquals(6, result);
    }
}
```

```
<terminated> CalculatorAaaTest [JUnit] C:\Program
Setup: Calculator initialized
Teardown: Calculator destroyed
Setup: Calculator initialized
Teardown: Calculator destroyed
```

## Exercise 1: Mocking and Stubbing with Mockito

**Scenario:**

You need to test a service ( `MyService` ) that depends on an external API ( `ExternalApi` ).

To properly isolate unit testing, use **Mockito** to:

- Create a mock object for the external API
- Stub the methods to return predefined values
- Write a test case that uses the mock object

**Steps Followed:**

- Created a Maven Java Project in Eclipse named `MockitoExample`
- Added JUnit 5 and Mockito dependencies in `pom.xml`
- Created a interface and two classes: `ExternalApi` , `MyService` , and `MyServiceTest`

### 1. pom.xml Configuration

Make sure your `pom.xml` includes the following dependencies:

```
<dependencies>
  <!-- JUnit 5 -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.9.3</version>
    <scope>test</scope>
  </dependency>

  <!-- Mockito -->
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>3.12.4</version> <!-- Compatible with Java 8 -->
    <scope>test</scope>
  </dependency>
</dependencies>
```

### 2. Class: ExternalApi.java

```
public class ExternalApi {
    public String getData() {
        // Simulate fetching from API
        return "Real Data";
    }
}
```

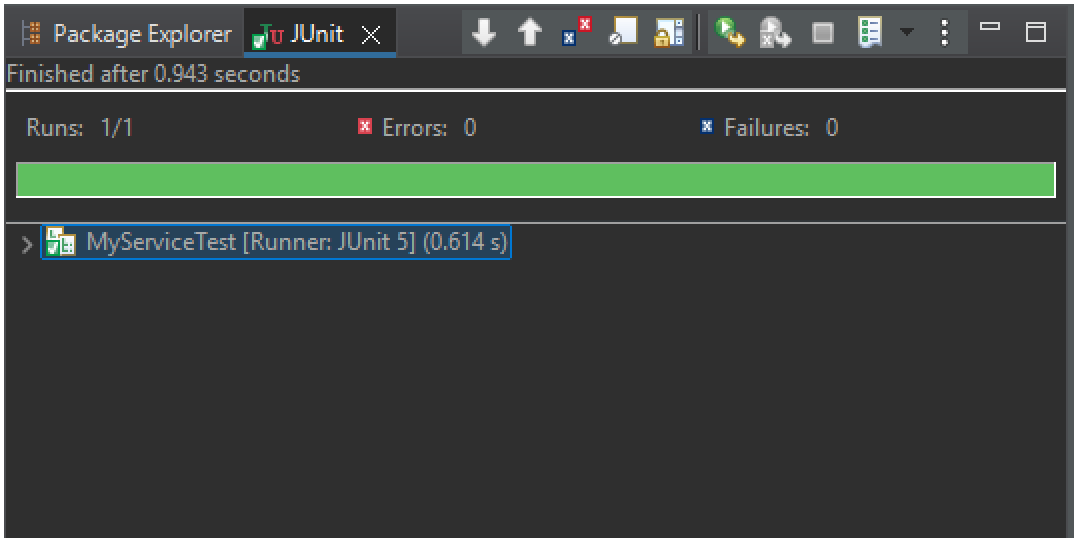
### 3. Class: MyService.java

```
public class MyService {
    private ExternalApi api;
    public MyService(ExternalApi api) {
        this.api = api;
    }

    public String fetchData() {
        return api.getData();
    }
}
```

**4. Test Class: MyServiceTest.java**

```
import static org.mockito.Mockito.*;
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

public class MyServiceTest {

    @Test
    public void testExternalApi() {
        // Arrange
        ExternalApi mockApi = Mockito.mock(ExternalApi.class);
        when(mockApi.getData()).thenReturn("Mock Data");

        MyService service = new MyService(mockApi);

        // Act
        String result = service.fetchData();

        // Assert
        assertEquals("Mock Data", result);
    }
}
```

**Output:**



Finished after 0.943 seconds

Runs: 1/1 Errors: 0 Failures: 0

MyServiceTest [Runner: JUnit 5] (0.614 s)

## Exercise 2: Verifying Interactions using Mockito

### Scenario

You need to ensure that a method is called with specific arguments. This is useful for verifying that your service interacts correctly with its dependencies.

Steps:

1. Create a mock object.
2. Call the method with specific arguments.
3. Verify the interaction.

### Code Implementation

### ExternalApiV2.java

```
public interface ExternalApiV2 {
    String getData();
}
```

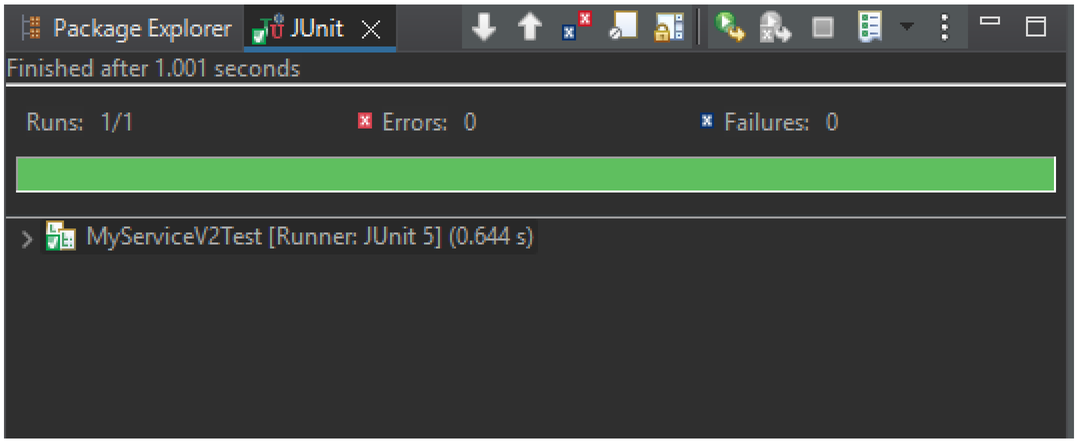
### MyServiceV2.java

```
public class MyServiceV2 {
    private ExternalApiV2 api;

    public MyServiceV2(ExternalApiV2 api) {
        this.api = api;
    }

    public String fetchData() {
        return api.getData();
    }
}
```

### MyServiceV2Test.java

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

public class MyServiceV2Test {

    @Test
    public void testVerifyInteraction() {
        ExternalApiV2 mockApi = mock(ExternalApiV2.class);
        when(mockApi.getData()).thenReturn("Mock Data");

        MyServiceV2 service = new MyServiceV2(mockApi);
        String result = service.fetchData();

        assertEquals("Mock Data", result);
        verify(mockApi).getData(); // Verifying interaction
    }
}
```

## Output:



Package Explorer JUnit

Finished after 1.001 seconds

Runs: 1/1 Errors: 0 Failures: 0

MyServiceV2Test [Runner: JUnit 5] (0.644 s)