# Week 2 - TDD using JUnit5 and Mockito - Hands-On

## 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:

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

3. Create a new test class in your project.

Solution Code:

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

java

Copy

Edit

// CalculatorTest.java - The JUnit test class

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calc = new Calculator();

int result = calc.add(2, 3);

assertEquals(5, result);

}

}

### **Maven Dependency (add to pom.xml)**

xml

CopyEdit

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

## Exercise 3: Assertions in JUnit

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

* Steps:

1. Write tests using various JUnit assertions.

Solution Code:

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());  
 }  
}

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

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

* Steps:

1. Write tests using the AAA pattern.

2. Use @Before and @After annotations for setup and teardown methods.

Solution Code:

### **Class to be Tested: Calculator.java**

java

CopyEdit

public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
  
 public int multiply(int a, int b) {  
 return a \* b;  
 }  
}

### **Test Class Using AAA Pattern: CalculatorTest.java**

java

CopyEdit

import org.junit.After;  
import org.junit.Before;  
import org.junit.Test;  
import static org.junit.Assert.\*;  
  
public class CalculatorTest {  
  
 private Calculator calculator;  
  
 // Setup method – runs before each test  
 @Before  
 public void setUp() {  
 calculator = new Calculator();  
 System.out.println("Setup completed.");  
 }  
  
 // Teardown method – runs after each test  
 @After  
 public void tearDown() {  
 calculator = null;  
 System.out.println("Teardown completed.");  
 }  
  
 // Test using AAA pattern for add method  
 @Test  
 public void testAdd() {  
 // Arrange  
 int a = 5;  
 int b = 3;  
  
 // Act  
 int result = calculator.add(a, b);  
  
 // Assert  
 assertEquals(8, result);  
 }  
  
 // Test using AAA pattern for multiply method  
 @Test  
 public void testMultiply() {  
 // Arrange  
 int a = 4;  
 int b = 2;  
  
 // Act  
 int result = calculator.multiply(a, b);  
  
 // Assert  
 assertEquals(8, result);  
 }  
}

## Mockito 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.

* Steps:

1. Create a mock object for the external API.

2. Stub the methods to return predefined values.

3. Write a test case that uses the mock object.

Solution Code:

import static org.mockito.Mockito.\*;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
  
public class MyServiceTest {  
  
 @Test  
 public void testExternalApi() {  
 ExternalApi mockApi = Mockito.mock(ExternalApi.class);  
 when(mockApi.getData()).thenReturn("Mock Data");  
  
 MyService service = new MyService(mockApi);  
 String result = service.fetchData();  
  
 assertEquals("Mock Data", result);  
 }  
}

## Mockito Exercise 2: Verifying Interactions

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

* Steps:

1. Create a mock object.

2. Call the method with specific arguments.

3. Verify the interaction.

Solution Code:

import static org.mockito.Mockito.\*;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
  
public class MyServiceTest {  
  
 @Test  
 public void testVerifyInteraction() {  
 ExternalApi mockApi = Mockito.mock(ExternalApi.class);  
 MyService service = new MyService(mockApi);  
 service.fetchData();  
 verify(mockApi).getData();  
 }  
}