**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>org.junit.jupiter</groupId>

<artifactId>junit-jupiter</artifactId>

<version>5.9.3</version>

<scope>test</scope>

</dependency>

**Source Code:**

**Pom.xml**

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 <http://maven.apache.org/xsd/maven-4.0.0.xsd>">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>org.example</groupId>  
 <artifactId>cognizant-company</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <properties>  
 <maven.compiler.source>22</maven.compiler.source>  
 <maven.compiler.target>22</maven.compiler.target>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.13.2</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>RELEASE</version>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
  
</project>

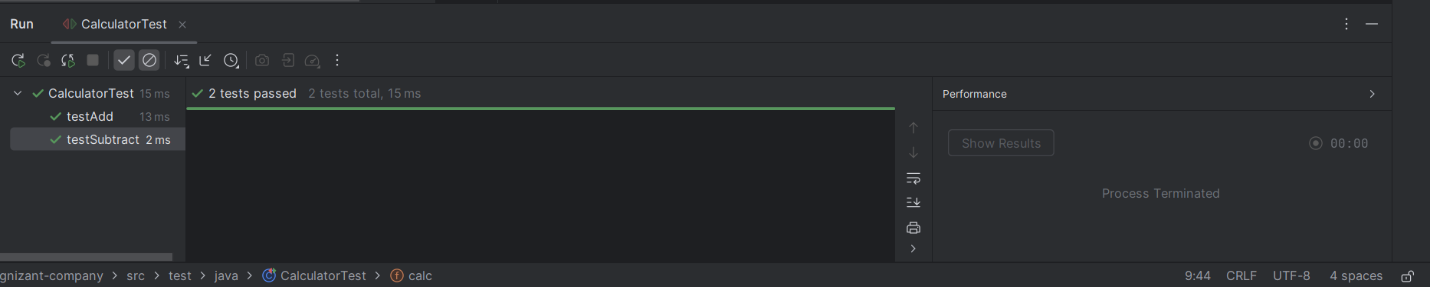
**CalculatorTest.java**

import org.example.Calculator;  
import org.junit.Test;  
import static org.junit.Assert.\*;  
  
public class CalculatorTest {  
  
 Calculator calc = new Calculator();  
  
 @Test  
 public void testAdd() {  
 int result = calc.add(5, 3);  
 *assertEquals*(8, result);  
 }  
  
 @Test  
 public void testSubtract() {  
 int result = calc.subtract(10, 4);  
 *assertEquals*(6, result);  
 }  
 }

**Calculator.java**

package org.example;  
  
public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
  
 public int subtract(int a, int b) {  
 return a - b;  
 }  
 }

**Output:**



**Exercise 2: 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());

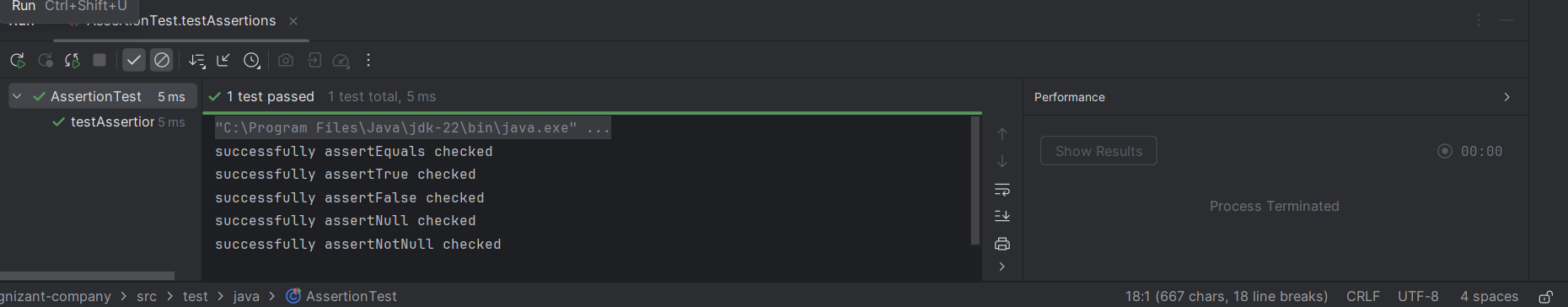
}

}

**Source code:**

import org.junit.Test;  
import static org.junit.Assert.\*;  
public class AssertionTest {  
 @Test  
 public void testAssertions() {  
 *assertEquals*(5, 2 + 3);  
 System.*out*.println("successfully assertEquals checked");  
 *assertTrue*(5 > 3);  
 System.*out*.println("successfully assertTrue checked");  
 *assertFalse*(5 < 3);  
 System.*out*.println("successfully assertFalse checked");  
 *assertNull*(null);  
 System.*out*.println("successfully assertNull checked");  
 *assertNotNull*(new Object());  
 System.*out*.println("successfully assertNotNull checked");  
 }  
 }

**Output:**



**Exercise 3: 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.

Steps: 1. Write tests using the AAA pattern.

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

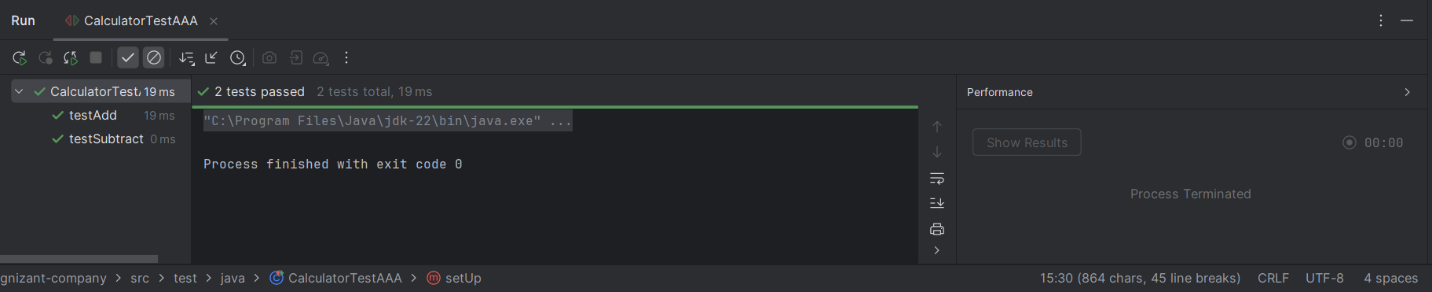
**CalculatorTestAAA.java**

import org.junit.Before;  
import org.junit.After;  
import org.junit.Test;  
import static org.junit.Assert.\*;  
import org.example.Calculator;  
public class CalculatorTestAAA {  
  
  
  
  
 private Calculator calculator;  
  
 @Before  
 public void setUp() {  
 calculator = new Calculator();  
 }  
  
 @After  
 public void tearDown() {  
 calculator = null;  
 }  
  
 @Test  
 public void testAdd() {  
 // Arrange  
 int a = 10;  
 int b = 5;  
  
 // Act  
 int result = calculator.add(a, b);  
  
 // Assert  
 *assertEquals*(15, result);  
 }  
  
 @Test  
 public void testSubtract() {  
 int a = 10;  
 int b = 3;  
 int result = calculator.subtract(a, b);  
 *assertEquals*(7, result);  
 }  
 }

**Calculator.java**

package org.example;  
  
public class Calculator {  
 public int add(int a, int b) {  
 return a + b;  
 }  
  
 public int subtract(int a, int b) {  
 return a - b;  
 }  
 }

**Output:**



**Exercise 4: 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);

} }

**Source Code:**

**MyServiceTest.java**

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

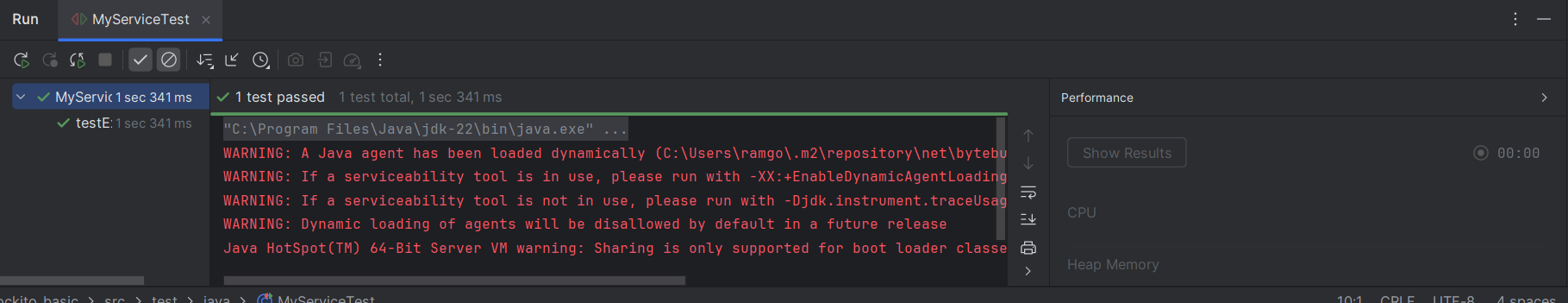
**MyService.java**

package org.example;  
  
public class MyService {  
 private ExternalApi externalApi;  
  
 public MyService(ExternalApi externalApi) {  
 this.externalApi = externalApi;  
 }  
  
 public String fetchData() {  
 return externalApi.getData();  
 }  
}

**ExternalApi.java**

package org.example;  
  
public interface ExternalApi {  
 String getData();  
}

**Output:**



**Note**: Above Warnings gives the information about External Api loading dynamically. So there is no errors in code and syntax

**Exercise 5: 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();

} }

**Source Code:**

**MyService.java**

package org.example;  
  
public class MyService {  
 private ExternalApi externalApi;  
  
 public MyService(ExternalApi externalApi) {  
 this.externalApi = externalApi;  
 }  
  
 public String fetchData() {  
 return externalApi.getData();  
 }  
}

**ExternalApi.java**

package org.example;  
  
public interface ExternalApi {  
 String getData();  
}

**MyServiceTest.java**

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

**Output:**

