**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: junit junit 4.13.2 test

3. Create a new test class in your project.

**Solution:**

* **Pom.xml file**

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

* **MainTest.java**

package org.example

public class MainTest {

}

**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());

}

}

**Solution:**

import org.junit.jupiter.api.Test;

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

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3, "Sum should be 5");

assertTrue(5 > 3, "5 is greater than 3");

assertFalse(5 < 3, "5 is not less than 3");

assertNull(null, "Value should be null");

assertNotNull(new Object(), "Object should not be null");

}

}

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**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.

Steps:

1. Write tests using the AAA pattern.

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

**Solution:**

* **Main.java**

package org.example;

public class Main {

public int add(int n1, int n2) {

return n1 + n2;

}

public int subtract(int n1, int n2) {

return n1 - n2;

}

public static void main(String[] args) {

System.out.println("Hey");

}

}

* **MainTest.java**

package org.example;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class MainTest {

private Main mainApp;

@Before

public void setUp() {

mainApp = new Main();

System.out.println("Setup complete");

}

@After

public void tearDown() {

mainApp = null;

System.out.println("Teardown complete");

}

@Test

public void testAddition() {

int a = 5;

int b = 3;

int result = mainApp.add(a, b);

assertEquals("Addition result should be 8", 8, result);

}

@Test

public void testSubtraction() {

int a = 10;

int b = 4;

int result = mainApp.subtract(a, b);

assertEquals("Subtraction result should be 6", 6, result);

}

}

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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.

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);

}

}

**Solution:**

* **ForMock.java**

package org.example;

public class ForMock {

private final ExternalAPI api;

public ForMock(ExternalAPI api) {

this.api = api;

}

public int add(int n1, int n2) {

return n1 + n2;

}

public int addWithRemote(int n1) {

int remoteValue = api.fetchRemoteValue();

return n1 + remoteValue;

}

public int subtract(int n1, int n2) {

return n1 - n2;

}

public static void main(String[] args) {

System.out.println("Hey");

}

}

* **ForMockTest.java**

package org.example;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

import static org.mockito.Mockito.\*;

public class ForMockTest {

private ExternalAPI apiMock;

private ForMock calculator;

@Before

public void setUp() {

apiMock = mock(ExternalAPI.class);

when(apiMock.fetchRemoteValue()).thenReturn(5);

calculator = new ForMock(apiMock);

}

@Test

public void testAddWithRemote() {

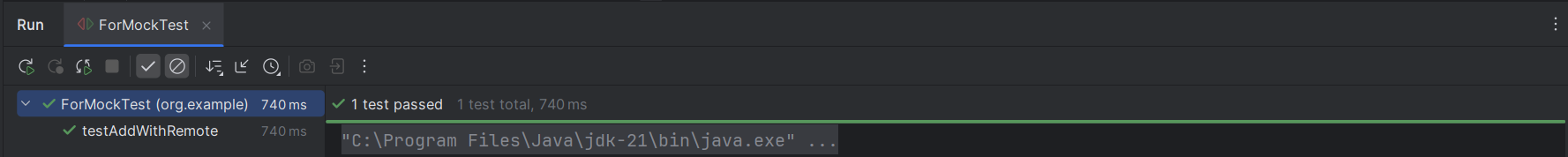
int result = calculator.addWithRemote(10);

assertEquals("Expected result of 10 + 5 = 15", 15, result);

}

}

**Output:**



**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();

}

}

**Solution:**

* **ExternalApi2.java**

package org.example;

public interface ExternalApi2 {

String getData();

}

* **MyService.java**

package org.example;

public class MyService {

private final ExternalApi2 api;

public MyService(ExternalApi2 api) {

this.api = api;

}

public void fetchData() {

api.getData();

}

}

**Output:**

