**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.

**CODE**

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

</dependencies>

package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

package com.example;

import org.junit.Test;

import static org.junit.Assert.assertEquals;

public class CalculatorTest {

@Test

public void testAdd() {

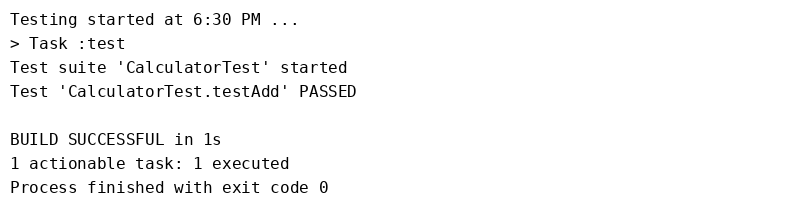
Calculator calc = new Calculator();

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

assertEquals(5, result);

}

}

**OUTPUT**

**Exercise 3: Assertions in JUnit**

Scenario:

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

Steps: Write tests using various JUnit assertions**.**

**CODE**

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

assertEquals(5, 2 + 3);

assertTrue(5 > 3);

assertFalse(5 < 3);

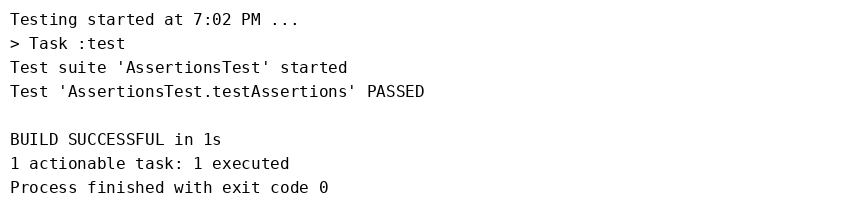
assertNull(null);

assertNotNull(new Object());

}

}

**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.

Steps:

1. Write tests using the AAA pattern.

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

**CODE**

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

public int multiply(int a, int b) {

return a \* b;

}

public int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Cannot divide by zero");

}

return a / b;

}

}

import org.junit.\*;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator();

}

@After

public void tearDown() {

calculator = null;

}

@Test

public void testAdd\_TwoPositiveNumbers\_ReturnsSum() {

int a = 5;

int b = 3;

int expected = 8;

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

assertEquals(expected, result);

}

@Test

public void testSubtract\_TwoNumbers\_ReturnsDifference() {

int a = 10;

int b = 4;

int expected = 6;

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

assertEquals(expected, result);

}

@Test

public void testMultiply\_TwoNumbers\_ReturnsProduct() {

int a = 6;

int b = 7;

int expected = 42;

int result = calculator.multiply(a, b);

assertEquals(expected, result);

}

@Test

public void testDivide\_ValidNumbers\_ReturnsQuotient() {

int a = 15;

int b = 3;

int expected = 5;

int result = calculator.divide(a, b);

assertEquals(expected, result);

}

@Test(expected = IllegalArgumentException.class)

public void testDivide\_ByZero\_ThrowsException() {

int a = 10;

int b = 0;

calculator.divide(a, b);

}

@Test

public void testAdd\_NegativeNumbers\_ReturnsCorrectSum() {

int a = -5;

int b = -3;

int expected = -8;

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

assertEquals(expected, result);

}

}

**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);

}

}

**CODE**

public interface ExternalApi {

String getData();

boolean isActive();

}

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

public String getStatus() {

return api.isActive() ? "Active" : "Inactive";

}

}

import static org.mockito.Mockito.\*;

import static org.junit.Assert.\*;

import org.junit.Test;

public class MyServiceTest {

@Test

public void testFetchData() {

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData()).thenReturn("Mock Data");

MyService service = new MyService(mockApi);

String result = service.fetchData();

assertEquals("Mock Data", result);

verify(mockApi).getData();

}

@Test

public void testGetStatus\_Active() {

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.isActive()).thenReturn(true);

MyService service = new MyService(mockApi);

String result = service.getStatus();

assertEquals("Active", result);

}

@Test

public void testGetStatus\_Inactive() {

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.isActive()).thenReturn(false);

MyService service = new MyService(mockApi);

String result = service.getStatus();

assertEquals("Inactive", result);

}

}

**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();

}

}

**CODE**

public interface ExternalApi {

String getData();

String getData(String id);

void saveData(String data);

boolean deleteData(String id);

}

public class MyService {

private ExternalApi api;

public MyService(ExternalApi api) {

this.api = api;

}

public String fetchData() {

return api.getData();

}

public String fetchDataById(String id) {

return api.getData(id);

}

public void storeData(String data) {

api.saveData(data);

}

public boolean removeData(String id) {

return api.deleteData(id);

}

}

import static org.mockito.Mockito.\*;

import static org.junit.Assert.\*;

import org.junit.Test;

public class MyServiceTest {

@Test

public void testVerifyBasicInteraction() {

System.out.println("=== Testing Basic Interaction Verification ===");

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData()).thenReturn("Test Data");

MyService service = new MyService(mockApi);

String result = service.fetchData();

System.out.println("Method called, result: " + result);

verify(mockApi).getData();

System.out.println("✓ Verified: getData() was called");

assertEquals("Test Data", result);

System.out.println("✓ Test passed: Basic interaction verified\n");

}

@Test

public void testVerifyWithSpecificArguments() {

System.out.println("=== Testing Verification with Specific Arguments ===");

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData("123")).thenReturn("User Data");

MyService service = new MyService(mockApi);

String result = service.fetchDataById("123");

System.out.println("Called fetchDataById with ID: 123");

System.out.println("Result: " + result);

verify(mockApi).getData("123");

System.out.println("✓ Verified: getData(\"123\") was called with correct argument");

assertEquals("User Data", result);

System.out.println("✓ Test passed: Specific argument verification\n");

}

@Test

public void testVerifyVoidMethod() {

System.out.println("=== Testing Void Method Verification ===");

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

String testData = "Important Data";

service.storeData(testData);

System.out.println("Called storeData with: " + testData);

verify(mockApi).saveData("Important Data");

System.out.println("✓ Verified: saveData(\"Important Data\") was called");

System.out.println("✓ Test passed: Void method verification\n");

}

@Test

public void testVerifyCallCount() {

System.out.println("=== Testing Call Count Verification ===");

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData()).thenReturn("Data");

MyService service = new MyService(mockApi);

service.fetchData();

service.fetchData();

service.fetchData();

System.out.println("Called fetchData() 3 times");

verify(mockApi, times(3)).getData();

System.out.println("✓ Verified: getData() was called exactly 3 times");

System.out.println("✓ Test passed: Call count verification\n");

}

@Test

public void testVerifyNeverCalled() {

System.out.println("=== Testing Never Called Verification ===");

ExternalApi mockApi = mock(ExternalApi.class);

MyService service = new MyService(mockApi);

System.out.println("Service created but no methods called");

verify(mockApi, never()).getData();

System.out.println("✓ Verified: getData() was never called");

verify(mockApi, never()).saveData(anyString());

System.out.println("✓ Verified: saveData() was never called with any string");

System.out.println("✓ Test passed: Never called verification\n");

}

@Test

public void testVerifyAtLeastOnce() {

System.out.println("=== Testing At Least Once Verification ===");

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData()).thenReturn("Data");

MyService service = new MyService(mockApi);

service.fetchData();

service.fetchData();

System.out.println("Called fetchData() multiple times");

verify(mockApi, atLeastOnce()).getData();

System.out.println("✓ Verified: getData() was called at least once");

verify(mockApi, atLeast(2)).getData();

System.out.println("✓ Verified: getData() was called at least 2 times");

System.out.println("✓ Test passed: At least verification\n");

}

@Test

public void testVerifyWithArgumentMatchers() {

System.out.println("=== Testing Argument Matchers Verification ===");

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.deleteData(anyString())).thenReturn(true);

MyService service = new MyService(mockApi);

boolean result1 = service.removeData("user1");

boolean result2 = service.removeData("user2");

System.out.println("Called removeData with user1, result: " + result1);

System.out.println("Called removeData with user2, result: " + result2);

verify(mockApi, times(2)).deleteData(anyString());

System.out.println("✓ Verified: deleteData() was called 2 times with any string");

verify(mockApi).deleteData("user1");

verify(mockApi).deleteData("user2");

System.out.println("✓ Verified: deleteData() was called with specific users");

System.out.println("✓ Test passed: Argument matchers verification\n");

}

@Test

public void testVerifyInOrder() {

System.out.println("=== Testing In Order Verification ===");

ExternalApi mockApi = mock(ExternalApi.class);

when(mockApi.getData(anyString())).thenReturn("Data");

MyService service = new MyService(mockApi);

service.fetchDataById("first");

service.fetchDataById("second");

service.fetchDataById("third");

System.out.println("Called methods in sequence: first, second, third");

InOrder inOrder = inOrder(mockApi);

inOrder.verify(mockApi).getData("first");

inOrder.verify(mockApi).getData("second");

inOrder.verify(mockApi).getData("third");

System.out.println("✓ Verified: Methods were called in correct order");

System.out.println("✓ Test passed: In order verification\n");

}

}