**JUnit Testing Exercises**

**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:**

*// Calculator.java*

**package** com.code.java.JUnitPractice;

**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) {

**return** a / b;

}

**Else** {

**return** -1;

}

}

}

*// CalculatorTest.java*

package com.code.java.JUnitPractice;

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

import org.junit.jupiter.api.Test;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calculator = new Calculator();

int result = calculator.add(3, 4);

assertEquals(7, result);

System.out.println("Addition Successful...");

}

@Test

public void testSubtract() {

Calculator calculator = new Calculator();

int result = calculator.subtract(10, 5);

assertEquals(5, result);

System.out.println("Subtraction Successful...");

}

@Test

public void testMultiply() {

Calculator calculator = new Calculator();

int result = calculator.multiply(8, 6);

assertEquals(48, result);

System.out.println("Multiplication Successful...");

}

@Test

public void testDivide() {

Calculator calculator = new Calculator();

int result = calculator.divide(10, 5);

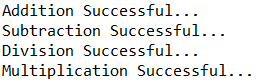
assertEquals(2, result);

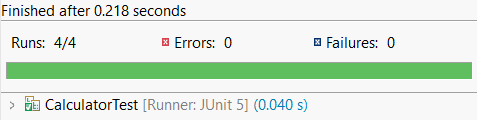
System.out.println("Division Successful...");

}

}

**Output:**





**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:**

**Code:**

package com.code.java.JUnitPractice;

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

import org.junit.jupiter.api.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);

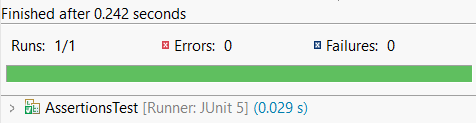
// Assert not null

assertNotNull(new Object());

}

}

**Output:**



**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:**

**Code:**

*// BankAccount.java*

**package** com.code.java.JUnitPractice;

**public** **class** BankAccount {

**private** **int** accbalance;

**public** BankAccount(**int** initialBalance) {

**this**.accbalance = initialBalance;

}

**public** **void** deposit(**int** depositeamount) {

accbalance += depositeamount;

}

**public** **void** withdraw(**int** withdrawlamount) {

accbalance -= withdrawlamount;

}

**public** **int** getBalance() {

**return** accbalance;

}

}

*//BankAccountTest.java*

package com.code.java.JUnitPractice;

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

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeEach;

import org.junit.jupiter.api.Test;

public class BankAccountTest {

private BankAccount account;

*// Setup method - runs before each test*

@BeforeEach

public void accountSetUp() {

account = new BankAccount(1000); *// Arrange: create new account with balance 1000*

System.out.println("Setting up the account before test...");

}

@Test

public void testDeposit() {

*// Arrange: Already done in setUp()*

*// Act*

account.deposit(500);

*// Assert*

assertEquals(1500, account.getBalance());

System.out.println("Amount Deposited Successfully...");

}

@Test

public void testWithdraw() {

*// Arrange: Already done in setUp()*

*// Act*

account.withdraw(800);

*// Assert*

assertEquals(200, account.getBalance());

System.out.println("Amount Withdrawn Successfully...");

}

*// Tear down method - runs after each test*

@AfterEach

public void accountTearDown() {

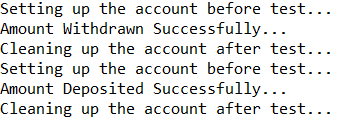
account = null;

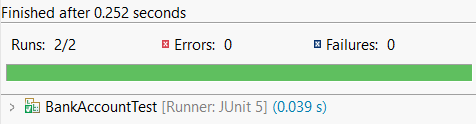
System.out.println("Cleaning up the account after test...");

}

}

**Output:**





**Mockito Hands-On 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:**

**Code:**

*// ExternalApi.java*

**package** com.code.java.MokitoPractice;

**public** **class** ExternalApi {

**public** String getData() {

// In real life, this could make an API call

**return** "Welcome to My Service";

}

}

*// MyService.java*

**package** com.code.java.MokitoPractice;

**public** **class** MyService {

**private** ExternalApi externalApi;

**public** MyService(ExternalApi externalApi) {

**this**.externalApi = externalApi;

}

**public** String fetchData() {

**return** externalApi.getData();

}

}

*// MyServiceTest.java*

package com.code.java.MokitoPractice;

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() {

*// Step 1: Create a mock of ExternalApi*

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

*// Step 2: Stub the getData() method to return a fake value*

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

*// Step 3: Inject mock into MyService*

MyService service = new MyService(mockApi);

*// Step 4: Call the method under test*

String result = service.fetchData();

*// Step 5: Assert the result*

assertEquals("Mock Data", result);

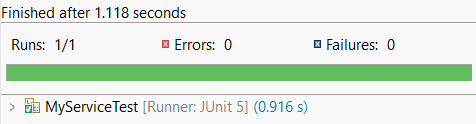
System.out.println("The Mocked Data : "+result);

}

}

**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:**

**Code:**

*// ExternalApi.java*

**package** com.code.java.MokitoPractice;

**public** **class** ExternalApi {

**public** String getData() {

// In real life, this could make an API call

**return** "Welcome to My Service";

}

}

*// MyService.java*

**package** com.code.java.MokitoPractice;

**public** **class** MyService {

**private** ExternalApi externalApi;

**public** MyService(ExternalApi externalApi) {

**this**.externalApi = externalApi;

}

**public** String fetchData() {

**return** externalApi.getData();

}

}

*// MyServiceTest.java*

package com.code.java.MokitoPractice;

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest1 {

@Test

public void testVerifyInteraction() {

*// Step 1: Create mock object*

ExternalApi mockApi = Mockito.mock(ExternalApi.class);

*// Step 2: Inject mock into MyService*

MyService service = new MyService(mockApi);

*// Step 3: Call method under test*

service.fetchData();

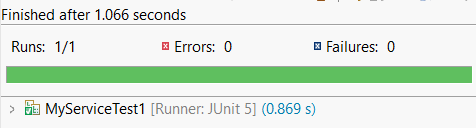
*// Step 4: Verify that getData() was called exactly once*

verify(mockApi).getData();

}

}

**Output:**



**Logging using SLF4J**

**Exercise 1: Logging Error Messages and Warning Levels**

**Task:** Write a Java application that demonstrates logging error messages and warning levels using SLF4J.

**Step-by-Step Solution:**

1. Add SLF4J and Logback dependencies to your `pom.xml` file:

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

2. Create a Java class that uses SLF4J for logging:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

public static void main(String[] args) {

logger.error("This is an error message");

logger.warn("This is a warning message");

}

}

**Solution:**

**Code:**

package come.code.java.SLF4JLoggingPractice;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

private static final Logger logger = LoggerFactory.getLogger(LoggingExample.class);

public static void main(String[] args) {

logger.error("This is an Error message");

logger.warn("This is a Warning message");

}

}

**Output:**

