**WEEK 2 - JUnit Testing Exercises**

**Exercise 1: Setting Up JUnit**

**Scenario:**  
You need to set up JUnit in your Java project to start writing unit tests.

**pom.xml:**

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

**Exercise 2: Writing Basic JUnit Tests**

**Scenario:**  
You need to write basic JUnit tests for a simple Java class.

**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;

}

}

**Test Code:**

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

Calculator calc = new Calculator();

@Test

public void testAddition() {

int result = calc.add(5, 2);

System.out.println("Addition: 5 + 2 = " + result);

assertEquals(7, result);

}

@Test

public void testSubtraction() {

int result = calc.subtract(9, 4);

System.out.println("Subtraction: 9 - 4 = " + result);

assertEquals(5, result);

}

@Test

public void testMultiplication() {

int result = calc.multiply(6, 3);

System.out.println("Multiplication: 6 \* 3 = " + result);

assertEquals(18, result);

}

@Test

public void testDivision() {

int result = calc.divide(8, 2);

System.out.println("Division: 8 / 2 = " + result);

assertEquals(4, result);

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

calc.divide(5, 0);

}

}

**Output:**

Addition: 5 + 2 = 7

Subtraction: 9 - 4 = 5

Multiplication: 6 \* 3 = 18

Division: 8 / 2 = 4

Time: 0.005

**Exercise 3: Assertions in JUnit**

**Scenario:**  
You want to understand how different assertion methods work in JUnit.

**Test Code:**

import org.junit.Test;

import static org.junit.Assert.\*;

public class AssertionsTest {

@Test

public void testAssertions() {

System.out.println("Testing assertEquals: 2 + 3 = " + (2 + 3));

assertEquals(5, 2 + 3);

System.out.println("Testing assertTrue: 5 > 3 is " + (5 > 3));

assertTrue(5 > 3);

System.out.println("Testing assertFalse: 5 < 3 is " + (5 < 3));

assertFalse(5 < 3);

System.out.println("Testing assertNull: null is null");

assertNull(null);

System.out.println("Testing assertNotNull: new Object() is not null");

assertNotNull(new Object());

System.out.println("All assertions passed successfully");

}

}

**Output:**

Testing assertEquals: 2 + 3 = 5

Testing assertTrue: 5 > 3 is true

Testing assertFalse: 5 < 3 is false

Testing assertNull: null is null

Testing assertNotNull: new Object() is not null

All assertions passed successfully

Time: 0.001

**✅ Exercise 4: Arrange-Act-Assert(AAA) Pattern, Test Fixtures, Setup and Teardown Methods in JUnit**

**Scenario:**  
Demonstrate Arrange-Act-Assert (AAA) pattern with setup and teardown methods in JUnit.

**Test Code:**

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class AAAPatternTest {

private Calculator calculator;

@Before

public void setUp() {

calculator = new Calculator();

System.out.println("Setup: Calculator initialized");

}

@After

public void tearDown() {

calculator = null;

System.out.println("Teardown: Calculator cleaned up");

}

@Test

public void testAddition() {

int a = 10, b = 5;

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

System.out.println("Testing addition: " + a + " + " + b + " = " + result);

assertEquals(15, result);

}

@Test

public void testDivision() {

int dividend = 20, divisor = 4;

int result = calculator.divide(dividend, divisor);

System.out.println("Testing division: " + dividend + " / " + divisor + " = " + result);

assertEquals(5, result);

}

}

**Output:**

Setup: Calculator initialized

Testing addition: 10 + 5 = 15

Teardown: Calculator cleaned up

Setup: Calculator initialized

Testing division: 20 / 4 = 5

Teardown: Calculator cleaned up

Time: 0.004

# **Mockito Exercises**

## **Exercise 1: Mocking and Stubbing**

**Scenario:**  
You are testing a MyService class that fetches data from an external API. To isolate this service, you use Mockito to mock the external API and stub its method to return predefined values.

### Project Structure :

MyMockitoProject/  
├── src/  
│ └── main/  
│ ├── ExternalApi.java  
│ └── MyService.java  
│ └── test/  
│ └── MyServiceTest.java

### **ExternalApi.java**

public interface ExternalApi {  
 String getData();  
}

### **MyService.java**

public class MyService {  
 private ExternalApi api;  
  
 public MyService(ExternalApi api) {  
 this.api = api;  
 }  
  
 public String fetchData() {  
 return api.getData();  
 }  
}

### **MyServiceTest.java**

### import static org.mockito.Mockito.\*;

### import org.junit.jupiter.api.Test;

### import org.mockito.Mockito;

### public class MyServiceTest {

### @Test

### public void testExternalApi() {

### System.out.println("Creating mock ExternalApi...");

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

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

### MyService service = new MyService(mockApi);

### System.out.println("Fetched data: " + mockApi.getData());

### String result = service.fetchData();

### System.out.println("Test passed: " + result);

### }

### @Test

### public void testVerifyInteraction() {

### System.out.println("Calling fetchData() in MyService...");

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

### MyService service = new MyService(mockApi);

### service.fetchData();

### System.out.println("Verifying interaction with ExternalApi...");

### verify(mockApi).getData();

### System.out.println("getData() called successfully.");

### System.out.println("Test passed: Interaction verified");

### }

### }

### **Output :**

## Creating mock ExternalApi...

## Fetched data: Mock Data

## Test passed: Mock Data

## Calling fetchData() in MyService...

## Verifying interaction with ExternalApi...

## getData() called successfully.

## Test passed: Interaction verified

## **Exercise 2: Verifying Interactions**

**Scenario:**  
You need to verify that a specific method (getData()) was called during execution.

**MyServiceTest.java**

import static org.mockito.Mockito.\*;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

public class MyServiceTest {

### @Test

### public void testVerifyInteraction() {

### System.out.println("Calling fetchData() in MyService...");

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

### MyService service = new MyService(mockApi);

### service.fetchData();

### System.out.println("Verifying interaction with ExternalApi...");

### verify(mockApi).getData();

### System.out.println("getData() called successfully.");

### System.out.println("Test passed: Interaction verified");

### }

### }

**OUTPUT:**

Calling fetchData() in MyService...

Verifying interaction with ExternalApi...

getData() called successfully.

Test passed: Interaction verified

**SLF4J Logging with Logback**

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

**Step 1: pom.xml**

<dependencies>

<!-- SLF4J API -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.30</version>

</dependency>

<!-- Logback Classic (SLF4J backend) -->

<dependency>

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

<artifactId>logback-classic</artifactId>

<version>1.2.3</version>

</dependency>

</dependencies>

**Step 2: Create the Java Logging Class**

**LoggingExample.java**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingExample {

// Create a logger specific to this class

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

public static void main(String[] args) {

System.out.println("=== Application Started ===");

try {

simulateWarning();

simulateError();

} catch (Exception e) {

logger.error("Unexpected exception caught in main method", e);

}

System.out.println("=== Application Finished ===");

}

private static void simulateWarning() {

logger.warn("Warning: Disk space is below 10%");

}

private static void simulateError() {

try {

String str = null;

str.length(); // Will throw NullPointerException

} catch (NullPointerException ex) {

logger.error("Critical Error: Null value encountered", ex);

}

}

}

**OUTPUT:**

Calling fetchData() in MyService...

Verifying interaction with ExternalApi...

getData() called successfully.

Test passed: Interaction verified