**Mandatory hands-on**

**JUnit Testing Exercises**

**Exercise 1: Setting Up JUnit**

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

**Code:**

**Calculator.java**package com.example;

public class Calculator {

public int add(int a, int b) {

return a + b;

}

public int subtract(int a, int b) {

return a - b;

}

}

**CalculatorTest.java**

package com.example;

import static org.junit.Assert.\*;

import org.junit.Test;

public class CalculatorTest {

@Test

public void testAddition() {

Calculator calc = new Calculator();

assertEquals(5, calc.add(2, 3));

}

@Test

public void testSubtraction() {

Calculator calc = new Calculator();

assertEquals(2, calc.subtract(5, 3));

}

}

**Output:**

Running com.example.CalculatorTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

**Exercise 3: Assertions in JUnit**

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

**Code:**

**AssertionsTest.java**

package com.example;

import static org.junit.Assert.\*;

import org.junit.Test;

public class AssertionsTest {

@Test

public void testPrimitiveAssertions() {

assertEquals("Sum should be 10", 10, 5 + 5);

assertTrue("Number should be positive", 7 > 0);

assertFalse("Number should not be negative", 7 < 0);

}

@Test

public void testStringAssertions() {

String expected = "JUnit";

String actual = "JU" + "nit";

assertEquals("Strings should match", expected, actual);

}

@Test

public void testArrayAssertions() {

int[] expected = {1, 2, 3};

int[] actual = {1, 2, 3};

assertArrayEquals("Arrays should be equal", expected, actual);

}

@Test

public void testNullAssertions() {

Object obj = null;

assertNull("Object should be null", obj);

obj = new Object();

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

}

@Test

public void testSameObjectAssertions() {

String a = "Hello";

String b = a;

String c = new String("Hello");

assertSame("Should refer to same object", a, b);

assertNotSame("Should not refer to same object", a, c);

}

@Test

public void testFloatingPointAssertions() {

double result = 10.0 / 3.0;

assertEquals("Should be close to 3.33", 3.33, result, 0.01);

}

}

**Output:**

Running com.example.MoreAssertionsTest

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0

Results :

Tests run: 6, Failures: 0, Errors: 0

**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.

**Code:**

**Calculator.java**

package com.example;

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;

}

}

**CalculatorTest.java**

package com.example;

import static org.junit.Assert.\*;

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

public class CalculatorTest {

private Calculator;

// Setup method (runs before each test)

@Before

public void setUp() {

System.out.println("Setting up...");

calculator = new Calculator();

}

// Teardown method (runs after each test)

@After

public void tearDown() {

System.out.println("Tearing down...");

calculator = null;

}

@Test

public void testAddition() {

int a = 10;

int b = 5;

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

assertEquals(15, result);

}

@Test

public void testSubtraction() {

int a = 10;

int b = 3;

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

assertEquals(7, result);

}

@Test

public void testMultiplication() {

int a = 4;

int b = 3;

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

assertEquals(12, result);

}

@Test

public void testDivision() {

int a = 20;

int b = 4;

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

assertEquals(5, result);

}

@Test(expected = IllegalArgumentException.class)

public void testDivisionByZero() {

calculator.divide(10, 0);

}

}

**Output:**

Setting up...

Tearing down...

Setting up...

Tearing down...

...

Running com.example.CalculatorTest

Tests run: 5, Failures: 0, Errors: 0, Skipped: 0

**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.

**Code:**

**WeatherApiClient.java**

package com.example;

public interface WeatherApiClient {

String getWeather(String city);

}

**WeatherService.java**package com.example;

public class WeatherService {

private final WeatherApiClient apiClient;

public WeatherService(WeatherApiClient apiClient) {

this.apiClient = apiClient;

}

public String fetchWeather(String city) {

return apiClient.getWeather(city);

}

}

**WeatherServiceTest.java**

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

public class WeatherServiceTest {

@Test

public void testFetchWeatherWithMock() {

// Step 1: Create a mock

WeatherApiClient mockApiClient = mock(WeatherApiClient.class);

// Step 2: Stub the method

when(mockApiClient.getWeather("Chennai")).thenReturn("Sunny");

// Step 3: Inject the mock into the service

WeatherService = new WeatherService(mockApiClient);

// Step 4: Call the method and assert result

String result = weatherService.fetchWeather("Chennai");

assertEquals("Sunny", result);

// Verify that the method was called once

verify(mockApiClient).getWeather("Chennai");

}

}

**Output:**

Running com.example.WeatherServiceTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

**Exercise 2: Verifying Interactions**

Scenario: You need to ensure that a method is called with specific arguments.

**Code:**

**MessageSender.java**

package com.example;

public interface MessageSender {

void sendMessage(String recipient, String message);

}

**NotificationService.java**

package com.example;

public class NotificationService {

private final MessageSender;

public NotificationService(MessageSender messageSender) {

this.messageSender = messageSender;

}

public void notifyUser(String userEmail, String message) {

messageSender.sendMessage(userEmail, message);

}

}

**NotificationServiceTest.java**

package com.example;

import org.junit.Test;

import static org.mockito.Mockito.\*;

public class NotificationServiceTest {

@Test

public void testSendMessageCalledWithCorrectArguments() {

MessageSender mockSender = mock(MessageSender.class);

NotificationService service = new NotificationService(mockSender);

service.notifyUser("user@example.com", "Hello!");

verify(mockSender).sendMessage("user@example.com", "Hello!");

}

}

**Output:**

Running com.example.NotificationServiceTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

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

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

**Code:**

**LoggingDemo.java**

package com.example;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class LoggingDemo {

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

public static void main(String[] args) {

logger.info("Application started");

try {

int result = divide(10, 0); // will throw error

logger.info("Result is: {}", result);

} catch (ArithmeticException ex) {

logger.error("An error occurred: Division by zero", ex);

}

logger.warn("This is a warning message: Low disk space");

logger.info("Application finished");

}

public static int divide(int a, int b) {

return a / b;

}

}

**Output:**

[main] INFO com.example.LoggingDemo - Application started

[main] ERROR com.example.LoggingDemo - An error occurred: Division by zero

java.lang.ArithmeticException: / by zero

at com.example.LoggingDemo.divide(LoggingDemo.java:21)

...

[main] WARN com.example.LoggingDemo - This is a warning message: Low disk space

[main] INFO com.example.LoggingDemo - Application finished