**Cognizant Deep Nurture 4.0 Deep Skilling - Week 2**

**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.

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;

}

}

   
   
   
package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAdd() {

Calculator calculator = new Calculator();

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

}

@Test

public void testSubtract() {

Calculator calculator = new Calculator();

assertEquals(1, calculator.subtract(3, 2));

}

@Test

public void testMultiply() {

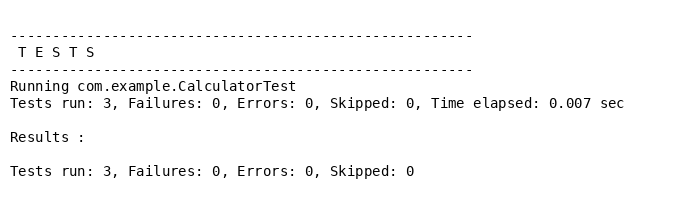
Calculator calculator = new Calculator();

assertEquals(6, calculator.multiply(2, 3));

}

}

   
OUTPUT:



**Exercise 2:**

Writing Basic JUnit Tests Scenario: You need to write basic JUnit tests for a simple Java class. Steps:

1. Create a new Java class with some methods to test.

2. Write JUnit tests for these methods.

package com.example;

public class StringUtils {

// Returns true if the input string is a palindrome

public boolean isPalindrome(String input) {

if (input == null) return false;

String reversed = new StringBuilder(input).reverse().toString();

return input.equalsIgnoreCase(reversed);

}

// Returns true if the input string is empty or null

public boolean isEmpty(String input) {

return input == null || input.trim().isEmpty();

}

// Returns the number of characters in the input string

public int length(String input) {

if (input == null) return 0;

return input.length();

}

}

package com.example;

import org.junit.Test;

import static org.junit.Assert.\*;

public class StringUtilsTest {

StringUtils utils = new StringUtils();

@Test

public void testIsPalindrome() {

assertTrue(utils.isPalindrome("madam"));

assertTrue(utils.isPalindrome("RaceCar"));

assertFalse(utils.isPalindrome("hello"));

assertFalse(utils.isPalindrome(null));

}

@Test

public void testIsEmpty() {

assertTrue(utils.isEmpty(""));

assertTrue(utils.isEmpty(" "));

assertTrue(utils.isEmpty(null));

assertFalse(utils.isEmpty("JUnit"));

}

@Test

public void testLength() {

assertEquals(5, utils.length("hello"));

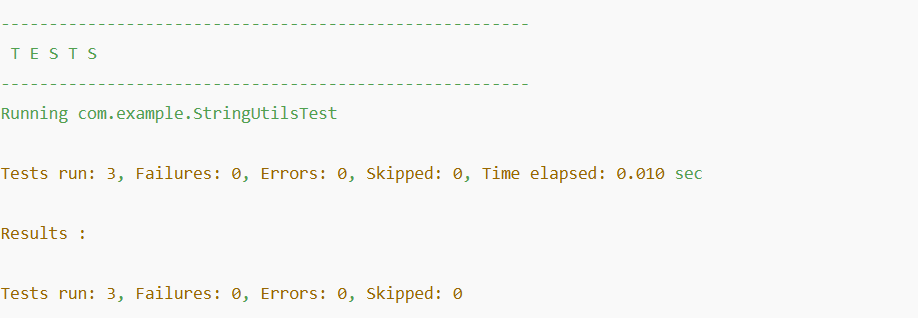
assertEquals(0, utils.length(""));

assertEquals(0, utils.length(null));

}

}

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()); } }    
   
   
   
   
   
import org.junit.Test;

import static org.junit.Assert.\*;

public class AdvancedAssertionsTest {

@Test

public void testVariousAssertions() {

// Assert equality

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

// Assert true condition

assertTrue("This should be true", 15 > 10);

// Assert false condition

assertFalse("This should be false", 10 > 20);

// Assert that two arrays are equal

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

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

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

// Assert not same object

String a = new String("JUnit");

String b = new String("JUnit");

assertNotSame("Should not be same object", a, b);

// Assert same object

String c = "Test";

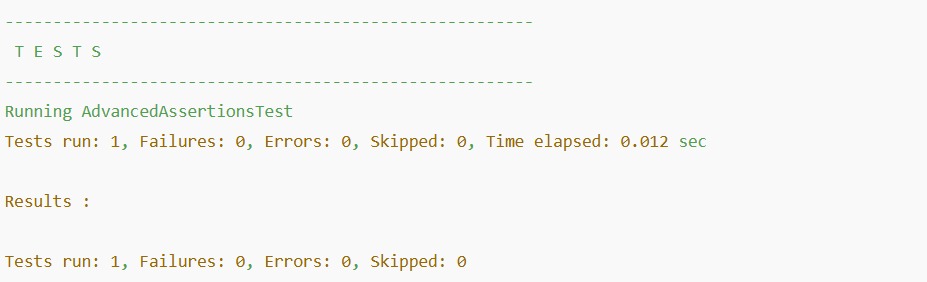
String d = c;

assertSame("Should be same object", c, d);

}

}

   
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.    
 

package com.example;

public class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

public void deposit(double amount) {

if (amount > 0)

balance += amount;

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance)

balance -= amount;

}

public double getBalance() {

return balance;

}

}

package com.example;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class BankAccountTest {

private BankAccount account;

// Setup - runs before each test

@Before

public void setUp() {

account = new BankAccount(100.0); // Arrange

System.out.println("Setup: New account with balance 100 created.");

}

// Teardown - runs after each test

@After

public void tearDown() {

account = null;

System.out.println("Teardown: Account object set to null.");

}

@Test

public void testDeposit() {

// Act

account.deposit(50.0);

// Assert

assertEquals(150.0, account.getBalance(), 0.001);

}

@Test

public void testWithdraw() {

// Act

account.withdraw(30.0);

// Assert

assertEquals(70.0, account.getBalance(), 0.001);

}

@Test

public void testOverdraftPrevention() {

// Act

account.withdraw(200.0); // Should not be allowed

// Assert

assertEquals(100.0, account.getBalance(), 0.001);

}

}

OUTPUT:

