# Cognizant Digital Nurture 4.0 Deep Skilling

## 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:**

Step 1: Create a New Java Project

Open your IDE (e.g., Eclipse or IntelliJ IDEA).

Navigate to File > New > Java Project.

Enter a name for the project, for example, JUnitDemo.

Click Finish to create the project structure.

Step 2: Add JUnit Dependency to the Project

If you are using Maven, add the following dependency to your pom.xml file inside the tag:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13.2</version>

<scope>test</scope>

</dependency>

Save the pom.xml file.

Right-click the project and select Maven > Update Project to download and apply the dependency

For non-Maven projects:

* Right-click the project > Build Path > Add Libraries > Select JUnit > Choose JUnit 4 > Finish.

**Step 3: Create a New Test Class**

* Right-click on src/test/java (or your test package) and select New > Class.
* Name the class, for example, CalculatorTest.
* Add the following basic test code:

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

@Test

public void testAddition() {

int result = 2 + 3;

assertEquals(5, result);

}

}

**Conclusion:** After completing the above steps, your Java project is ready for unit testing using JUnit. You can run the test class by right-clicking it and selecting Run As > JUnit Test. This setup ensures that your project supports test-driven development and automated testing using the JUnit framework.

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.

**Solution :**

Create a Java Class with Methods to Test Create a new class named Calculator.java:

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;

}

}

**Step 2: Write JUnit Test Cases for the Class** Create a test class named CalculatorTest.java:

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

Calculator calc = new Calculator();

@Test

public void testAdd() {

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

}

@Test

public void testSubtract() {

assertEquals(1, calc.subtract(4, 3));

}

@Test

public void testMultiply() {

assertEquals(12, calc.multiply(4, 3));

}

@Test

public void testDivide() {

assertEquals(2, calc.divide(6, 3));

}

@Test(expected = IllegalArgumentException.class)

public void testDivideByZero() {

calc.divide(10, 0);

}

}

**Conclusion:** This test class verifies all basic arithmetic operations and also checks for edge cases like division by zero. Writing such unit tests helps ensure that your business logic behaves as expected under various conditions.

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());

}

}

Step 1: Write Tests Using Various JUnit Assertions Create a test class named AssertionsTest.java:

import org.junit.Test;

import static org.junit.Assert.\*;

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());

}

}

**Explanation:**

* assertEquals(expected, actual): Verifies that two values are equal.
* assertTrue(condition): Verifies that the condition is true.
* assertFalse(condition): Verifies that the condition is false.
* assertNull(object): Verifies that the object is null.
* assertNotNull(object): Verifies that the object is not null.

**Conclusion:** These assertions help validate expected results in unit tests and catch incorrect behaviors in the code. Assertions are essential to ensure correctness, especially when your code base grows.

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 :**

Use the Arrange-Act-Assert Pattern The Arrange-Act-Assert pattern helps structure your test cases for readability:

Arrange: Set up the objects and prepare data.

Act: Call the method being tested.

Assert: Check that the result is as expected.

Step 2: Use @Before and @After Annotations These annotations are used to run setup and cleanup logic before and after each test.

import org.junit.Before;

import org.junit.After;

import org.junit.Test;

import static org.junit.Assert.\*;

public class CalculatorTest {

private Calculator calculator;

@Before

public void setUp() {

// Arrange: Create and initialize objects

calculator = new Calculator();

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

}

@After

public void tearDown() {

// Cleanup resources

calculator = null;

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

}

@Test

public void testAdd() {

// Act

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

// Assert

assertEquals(5, result);

}

@Test

public void testSubtract() {

int result = calculator.subtract(5, 2);

assertEquals(3, result);

}

}

**Conclusion:** Using the AAA pattern along with @Before and @After annotations makes your test code more maintainable and ensures consistent test environments. These features help reduce code duplication and ensure proper setup/cleanup for each test method.