Day 22 july 30

Name Aravind Kasanagottu

Id mvsnarav

Task 1: Good Code vs Bad Code

Good Code:

- Like a well-organized room — clear, clean, and organized

Key Qualities:

• Readable: Easy to understand

• Modular: Broken into small, reusable parts

• Well-named: Meaningful variable/method names

• Follows principles: SOLID, DRY, etc.

• Error-handling: Handles edge cases properly

• Testable: Easy to unit test

• Maintainable: Easy to modify

Example of Good Code:

public class Calculator {

public int add(int a, int b) {

return a + b;

}

}

Bad Code:

Common Problems:

• Hard to read: Vague names

• Too long: Does too many things

• Repeated logic: Code duplication

• No structure: Poor organization

• Poor documentation

• Not testable

Example of Bad Code:

public class X {

public int x(int a, int b) {

int c = a + b;

return c;

}

}

Task 2: Bad Code vs Good Code (Oral Explanation)

Bad code is functional but:

- Hard to read and understand

- Difficult to maintain

- Poor naming and structure

- Duplicated logic

- Doesn't follow principles

Good code is:

- Clean and readable

- Well-organized

- Properly named

- Follows best practices

- Easy to understand and maintain

Remember: "Bad code tells the computer what to do. Good code tells the next developer what's going on."

Task 3: Data Hiding

Data hiding is a key OOP concept that involves:

- Keeping internal details private

- Only exposing necessary information

- Using access modifiers (private, protected, public)

Example:

class Employee {

private int salary; // data hiding

public int getSalary() {

return salary;

}

public void setSalary(int salary) {

if (salary > 0) {

this.salary = salary;

}

}

}

Benefits:

• Prevents unauthorized access

• Enables validation

• Improves security

• Easier maintenance

Task 4: Continuous Development

Definition: Software engineering practice for frequent, automated code changes and delivery.

Key Components:

• Continuous Integration (CI)

• Continuous Delivery (CD)

• Continuous Deployment

Benefits:

• Faster updates

• Higher quality

• Early bug detection

• Better team productivity

Example Process:

1. Code changes

2. Automated testing

3. Automated deployment

4. Customer feedback

Task 5: Conditions for Polymorphism

Two Types:

1. Compile-time (Method Overloading)

2. Run-time (Method Overriding)

Conditions for Overloading:

• Same method name in same class

• Different parameters

• Return type can vary

Example:

class MathUtils {

int add(int a, int b) { return a + b; }

double add(double a, double b) { return a + b; }

}

Conditions for Overriding:

• Inheritance required

• Same method signature

• Non-static, non-private parent method

• Uses upcasting

Example:

class Animal {

void sound() { System.out.println("Animal sound"); }

}

class Dog extends Animal {

void sound() { System.out.println("Dog barks"); }

}  
  
  
  
// Task 6

package July30;

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import static org.junit.jupiter.api.Assertions.fail;

public class TestCase003 {

String message = "hello all How are you";

@Test

void testMsg() {

System.out.println("Asserting the Test case");

assertEquals("hello all How are you", message);

}

}

// Output: Asserting the Test case

// Task 7

// Same as Task 6 but with different output:

// Output:

// Running runTestcase02

// Running runTestcase04

// Running testMethod01

// Running testMethod03

// Task 8

package July30;

public class Junit4Test {

public int compare(int n1, int n2) {

if (n1 > n2) return 1;

return -1;

}

}

// Test Class

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

import org.junit.jupiter.api.Test;

class TestCase005 {

@Test

public void testCompare\_WhenFirstIsGreater() {

Junit4Test obj = new Junit4Test();

int result = obj.compare(10, 5);

assertEquals(1, result);

}

@Test

public void testCompare\_WhenSecondIsGreater() {

Junit4Test obj = new Junit4Test();

int result = obj.compare(4, 9);

assertEquals(-1, result);

}

@Test

public void testCompare\_WhenEqual() {

Junit4Test obj = new Junit4Test();

int result = obj.compare(7, 7);

assertEquals(-1, result);

}

}

// Task 9

public class NumberComparator {

public int compare(int n1, int n2) {

if (n1 > n2) return 1;

else if (n1 < n2) return -1;

return 0;

}

}

// Test Class

package July30;

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

import org.junit.jupiter.api.Test;

class TestCase006 {

@Test

public void testCompare\_WhenFirstIsGreater() {

NumberComparator obj = new NumberComparator();

int result = obj.compare(10, 5);

assertEquals(1, result);

}

@Test

public void testCompare\_WhenSecondIsGreater() {

NumberComparator obj = new NumberComparator();

int result = obj.compare(4, 9);

assertEquals(-1, result);

}

@Test

public void testCompare\_WhenEqual() {

NumberComparator obj = new NumberComparator();

int result = obj.compare(7, 7);

assertEquals(0, result);

}

}  
  
  
home task:

/\*

JUNIT ASSERTIONS - SIMPLE GUIDE

1. assertEquals

- Checks if two values are equal

Example:

assertEquals(5, 2 + 3); // Pass

assertEquals("hello", "hello"); // Pass

2. assertNotEquals

- Checks if two values are NOT equal

Example:

assertNotEquals(5, 2 + 2); // Pass

assertNotEquals("hi", "hello"); // Pass

3. assertTrue

- Checks if condition is true

Example:

assertTrue(10 > 5); // Pass

assertTrue("hello".length() > 0); // Pass

4. assertFalse

- Checks if condition is false

Example:

assertFalse(10 < 5); // Pass

assertFalse("".length() > 0); // Pass

5. assertNull

- Checks if something is null

Example:

String str = null;

assertNull(str); // Pass

6. assertNotNull

- Checks if something is NOT null

Example:

String str = "hello";

assertNotNull(str); // Pass

7. assertSame

- Checks if two references point to same object

Example:

String str = "test";

String same = str;

assertSame(str, same); // Pass

8. assertThrows

- Checks if code throws an exception

Example:

assertThrows(ArithmeticException.class, () -> {

int x = 1/0; // Pass

});

PRACTICAL EXAMPLE:

\*/

@Test

void simpleTests() {

// Testing a calculator

Calculator calc = new Calculator();

// Basic assertions

assertEquals(4, calc.add(2, 2));

assertTrue(calc.isPositive(5));

assertFalse(calc.isPositive(-5));

// Testing null

String emptyString = null;

assertNull(emptyString);

// Testing exceptions

assertThrows(ArithmeticException.class, () -> {

calc.divide(10, 0);

});

}

/\*

Remember:

- Use assertions to verify expected results

- Each test should test one specific thing

- Keep tests simple and readable

- Use meaningful test names

\*/