Name of the Course : LEARNING JAVA 9 – MODULAR PROGRAMMING

Level : High

Tool Stack : Java 9

Problem Statement : The composition is another mechanism provided by OOP for reusing implementation.

In a nutshell, composition allows us to model objects that are made up of other objects, thus defining a “has-a” relationship between them.

Furthermore, the composition is the strongest form of association, which means that the object(s) that compose or are contained by one object are destroyed too when that object is destroyed.

To better understand how composition works, let's suppose that we need to work with objects that represent computers.

Write a program to solve the above task.

Step 1 : A computer is composed of different parts, including the microprocessor, the memory, a sound card and so forth, so we can model both the computer and each of its parts as individual classes.

Here's how a simple implementation of the Computer class might look:

**Code:**

**package** com.iiht.java9.javacomposition;

**import** java.util.Optional;

**public** **class** Computer {

**private** Processor processor;

**private** Memory memory;

**private** SoundCard soundCard;

**public** Computer(Processor processor, Memory memory) {

**this**.processor = processor;

**this**.memory = memory;

}

**public** **void** setSoundCard(SoundCard soundCard) {

**this**.soundCard = soundCard;

}

**public** Processor getProcessor() {

**return** processor;

}

**public** Memory getMemory() {

**return** memory;

}

**public** Optional<SoundCard> getSoundCard() {

**return** Optional.*ofNullable*(soundCard);

}

@Override

**public** String toString() {

**return** "Computer{" + "processor=" + processor + ", memory=" + memory + ", soundcard=" + soundCard +"}";

}

}

Step 2 : The following classes model a microprocessor, the memory, and a sound card (interfaces are omitted for brevity's sake):

**Code:**

**package** com.iiht.java9.javacomposition;

**public** **interface** Memory {

String getBrand();

String getSize();

}

**package** com.iiht.java9.javacomposition;

**public** **interface** Memory {

String getBrand();

String getSize();

}

**package** com.iiht.java9.javacomposition;

**public** **interface** SoundCard {

String getBrand();

}

**package** com.iiht.java9.javacomposition;

**public** **class** StandardMemory **implements** Memory {

**private** String brand;

**private** String size;

**public** StandardMemory(String brand, String size) {

**this**.brand = brand;

**this**.size = size;

}

**public** String getBrand() {

**return** brand;

}

**public** String getSize() {

**return** size;

}

@Override

**public** String toString() {

**return** "Memory{" + "brand=" + brand + ", size=" + size + "}";

}

}

**package** com.iiht.java9.javacomposition;

**public** **class** StandardProcessor **implements** Processor {

**private** String model;

**public** StandardProcessor(String model) {

**this**.model = model;

}

@Override

**public** String getModel() {

**return** model;

}

@Override

**public** String toString() {

**return** "Processor{" + "model=" + model + "}";

}

}

**package** com.iiht.java9.javacomposition;

**public** **class** StandardSoundCard **implements** SoundCard {

**private** String brand;

**public** StandardSoundCard(String brand) {

**this**.brand = brand;

}

@Override

**public** String getBrand() {

**return** brand;

}

@Override

**public** String toString() {

**return** "SoundCard{" + "brand=" + brand + "}";

}

}

In the above example, Computer meets the “has-a” condition with the classes that model its parts.

It's also worth noting that in this case, the containing Computer object has ownership of the contained objects if and only if the objects can't be reused within another Computer object. If they can, we'd be using aggregation, rather than composition, where ownership isn't implied.

Step 3 : Composition Without Abstraction

Alternatively, we could've defined the composition relationship by hard-coding the dependencies of the Computer class, instead of declaring them in the constructor:

public class Computer {

private StandardProcessor processor

= new StandardProcessor("Intel I3");

private StandardMemory memory

= new StandardMemory("Kingston", "1TB");

// additional fields / methods

}

Of course, this would be a rigid, tightly-coupled design, as we'd be making Computer strongly dependent on specific implementations of Processor and Memory.

We wouldn't be taking advantage of the level of abstraction provided by interfaces and dependency injection.

With the initial design based on interfaces, we get a loosely-coupled design, which is also easier to test.

Step 3 : Test Cases

Alternatively, we can develop testcases to test the functionality of application

**package** com.iiht.java9.javacomposition;

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

**import org.junit.jupiter.api.AfterAll;**

**import org.junit.jupiter.api.AfterEach;**

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

**import com.iiht.java9.javainheritance.Actress;**

**import com.iiht.java9.javainheritance.Person;**

**import com.iiht.java9.javainheritance.Waitress;**

**import java.util.Optional;**

**import static org.assertj.core.api.Assertions.assertThat;**

**class JavaCompositionTest {**

**@AfterAll**

**static void tearDownAfterClass() throws Exception {**

**}**

**@AfterEach**

**void tearDown() throws Exception {**

**}**

**@Test**

**public void givenComputerInstance\_whenExtractedEachField\_thenThreeAssertions() {**

**Computer computer = new Computer(new StandardProcessor("Intel I3"), new StandardMemory("Kingston", "1TB"));**

**computer.setSoundCard(new StandardSoundCard("Generic Sound Card"));**

**assertThat(computer.getProcessor()).isInstanceOf(Processor.class);**

**assertThat(computer.getMemory()).isInstanceOf(Memory.class);**

**assertThat(computer.getSoundCard()).isInstanceOf(Optional.class);**

**}**

**}**

Learning outcome: Participant could able to learn how to use class composition relationship using Java 9.