Composition in Java

Objective

To understand the concept of **Composition** in Java, its practical application, and its distinction from inheritance. The lecture includes a detailed explanation of a program demonstrating composition and guidelines on when to use composition versus inheritance.

Introduction to Composition

What is Composition?

- Composition is an object-oriented programming principle where one class is composed of other classes as attributes. It represents a **"has-a"** relationship.
    - Example: A `PC` **has a** `Monitor`, `Case`, and `Motherboard`.

Why Use Composition?

- Promotes modular and reusable code.
- Allows flexible design, as components can be replaced or extended without changing the parent class.
- Supports aggregation of functionality, making code more organized.

Composition vs. Inheritance

Inheritance

- Represents an **"is-a"** relationship.
    - Example: A `Dog` is an `Animal`.
- Facilitates code reuse by inheriting fields and methods from a parent class.
- Can lead to a tightly coupled hierarchy, which may become rigid over time.

Composition

- Represents a **"has-a"** relationship.
    - Example: A `Car` has an `Engine`.
- Promotes loose coupling by embedding instances of other classes as fields.
- More flexible compared to inheritance, as behaviors can be dynamically composed or replaced.

When to Use Inheritance vs. Composition

Use Inheritance	Use Composition
When there is a clear "is-a" relationship.	When there is a "has-a" relationship.
To leverage polymorphism for method overriding.	To encapsulate behavior or functionality.
When class hierarchies are stable.	When design requires flexibility or reuse.

The Program: A PC Using Composition

The following program demonstrates composition by building a `PC` class that contains objects of other classes (`Case`, `Monitor`, `Motherboard`, etc.).

PC Class

```
public class PC {
    private Case theCase;
    private Monitor monitor;
    private Motherboard motherboard;

    public PC(Case theCase, Monitor monitor, Motherboard motherboard) {
        this.theCase = theCase;
        this.monitor = monitor;
        this.motherboard = motherboard;
    }

    public Case getTheCase() {
        return theCase;
    }

    public Monitor getMonitor() {
        return monitor;
    }

    public Motherboard getMotherboard() {
        return motherboard;
    }
}
```

Line-by-Line Explanation:

1. **Fields**:
   - `Case theCase`, `Monitor monitor`, and `Motherboard motherboard` represent the components of the PC.
   - This is the "has-a" relationship, as the `PC` has a `Case`, `Monitor`, and `Motherboard`.
2. **Constructor**:

- Initializes the `PC` by taking instances of `Case`, `Monitor`, and `Motherboard` as parameters.

3. **Getter Methods**:
   - Provide access to the individual components of the `PC`.

Case Class

```
public class Case {
    private String model;
    private String manufacturer;
    private String powerSupply;
    private Dimensions dimensions;

    public Case(String model, String manufacturer, String powerSupply,
Dimensions dimensions) {
        this.model = model;
        this.manufacturer = manufacturer;
        this.powerSupply = powerSupply;
        this.dimensions = dimensions;
    }

    public void pressPowerButton() {
        System.out.println("Power button pressed");
    }

    public String getModel() {
        return model;
    }

    public String getManufacturer() {
        return manufacturer;
    }

    public String getPowerSupply() {
        return powerSupply;
    }

    public Dimensions getDimensions() {
        return dimensions;
    }
}
```

Key Points:

- **Composition**: Includes a `Dimensions` object to represent the physical dimensions of the case.
- **Encapsulation**: Private fields are accessed through getter methods.
- **Behavior**: `pressPowerButton()` simulates powering up the PC.

Monitor Class

```
public class Monitor {
    private String model;
    private String manufacturer;
    private int size;
    private Resolution nativeResolution;

    public Monitor(String model, String manufacturer, int size, Resolution
nativeResolution) {
        this.model = model;
        this.manufacturer = manufacturer;
        this.size = size;
        this.nativeResolution = nativeResolution;
    }

    public void drawPixelAt(int x, int y, String color) {
        System.out.println("Drawing pixel at " + x + "," + y + " in colour " +
color);
    }
}
```

Key Points:

- **Composition**: Contains a `Resolution` object for screen resolution.
- **Behavior**: `drawPixelAt(x, y, color)` simulates pixel drawing.

Motherboard Class

```
public class Motherboard {
    private String model;
    private String manufacturer;
    private int ramSlots;
    private int cardSlots;
    private String bios;

    public Motherboard(String model, String manufacturer, int ramSlots, int
cardSlots, String bios) {
        this.model = model;
        this.manufacturer = manufacturer;
        this.ramSlots = ramSlots;
        this.cardSlots = cardSlots;
        this.bios = bios;
    }

    public void loadProgram(String programName) {
        System.out.println("Program " + programName + " is now loading...");
    }
```

```
}
```

Key Points:

- **Encapsulation**: Attributes are private and accessed via getter methods.
- **Behavior**: `loadProgram(String programName)` simulates program loading on the motherboard.

Supporting Classes

1. **Dimensions**: Represents physical dimensions of the case.
2. **Resolution**: Encapsulates the resolution details for the monitor.

Main Class

```
public class Main {
    public static void main(String[] args) {
        Dimensions dimensions = new Dimensions(20, 15, 5);
        Case theCase = new Case("220B", "Dell", "240", dimensions);

        Resolution nativeResolution = new Resolution(1920, 1080);
        Monitor monitor = new Monitor("27inch Beast", "Acer", 27,
nativeResolution);

        Motherboard motherboard = new Motherboard("BJ-200", "Asus", 4, 6,
"v2.44");

        PC thePC = new PC(theCase, monitor, motherboard);

        // Simulating PC usage
        thePC.getTheCase().pressPowerButton();
        thePC.getMonitor().drawPixelAt(1200, 50, "red");
        thePC.getMotherboard().loadProgram("Windows 11");
    }
}
```

Explanation:

1. **Objects Creation**:
    - `Dimensions`, `Resolution`, `Case`, `Monitor`, and `Motherboard` objects are created individually.
    - These objects are then used to construct the `PC` object, demonstrating composition.

2. **PC Simulation**:
    - `thePC.getTheCase().pressPowerButton()`: Accesses the `Case` object from the `PC` and simulates a power button press.
    - `thePC.getMonitor().drawPixelAt(1200, 50, "red")`: Accesses the `Monitor` to simulate drawing a pixel.
    - `thePC.getMotherboard().loadProgram("Windows 11")`: Accesses the `Motherboard` to simulate loading a program.

This main class brings all components together, showcasing how composition enables modular and reusable designs.

Better Practices for Using Composition

1. **Encapsulation**:
    - Always make fields private and access them through getter methods.
2. **Modularity**:
    - Design components as independent as possible to ensure reusability.
3. **Testing**:
    - Test each component individually before integrating them.
4. **Flexibility**:
    - Favor composition over inheritance when behaviors are subject to frequent change.

Guidelines on Using Inheritance vs. Composition

1. Use **inheritance** sparingly and only when the relationship is logically a clear "is-a".
2. Prefer **composition** when functionality can be aggregated and reused across multiple classes.
3. Avoid deep inheritance hierarchies to prevent tight coupling.

Discussion on Passing Objects in Java Constructors

When we call the constructors of classes like `Monitor`, `Case`, or `PC`, **object references** are being passed, not the objects themselves. Let's break this down:

Key Concept: Object References in Java

In Java, variables of non-primitive types (e.g., `Monitor`, `Resolution`) hold **references** to objects in memory, not the actual objects themselves. When an object is passed to a method or a constructor:

- The **reference** to the memory location of the object is passed.
- The object itself is not copied (unlike primitive types where values are copied).

This behavior ensures efficient memory usage and allows modifications made inside the constructor to affect the original object (if the object is mutable).

Example: Monitor Constructor

```
public Monitor(String model, String manufacturer, int size, Resolution
nativeResolution) {
    this.model = model;
    this.manufacturer = manufacturer;
    this.size = size;
    this.nativeResolution = nativeResolution;
}
```

What Happens?

1. **Primitive Types (`model`, `manufacturer`, `size`)**:
    - The values of these arguments are copied into the respective instance variables of the `Monitor` object.
2. **Object Type (`Resolution nativeResolution`)**:
    - The **reference** to the `Resolution` object is passed.
    - `this.nativeResolution` now points to the same memory location as the `nativeResolution` object passed to the constructor.

Example: PC Constructor

```
public PC(Case theCase, Monitor monitor, Motherboard motherboard) {
    this.theCase = theCase;
    this.monitor = monitor;
    this.motherboard = motherboard;
}
```

What Happens?

- The **references** to the `Case`, `Monitor`, and `Motherboard` objects are passed.

- The fields `this.theCase`, `this.monitor`, and `this.motherboard` now point to the same objects as those passed during the `PC` object creation.

Key Points to Understand

1. **Efficient Passing**:
   - Java avoids copying entire objects, ensuring efficient memory usage by passing references.
2. **Shared Objects**:
   - Since references are shared, any changes to a mutable object (like `Resolution`) inside the `PC` or `Monitor` class will reflect in the original object outside.
3. **Encapsulation and Safety**:
   - While references are passed, encapsulation ensures that fields marked as `private` can only be modified through controlled access (e.g., setters).

Visual Representation

1. When the `Monitor` object is created:
   - The `nativeResolution` reference is passed. Both the `nativeResolution` parameter and the `this.nativeResolution` field point to the same object in memory.
2. When the `PC` object is created:
   - The references to `Case`, `Monitor`, and `Motherboard` are passed. Both the fields in the `PC` class and the original objects point to the same memory locations.

Practical Implication

- If you modify the `Resolution` object inside the `Monitor` class using `getNativeResolution()`, the changes will affect the original `Resolution` object.
- Example:

```
Resolution nativeResolution = new Resolution(1920, 1080);
Monitor monitor = new Monitor("27inch Beast", "Acer", 27,
nativeResolution);

nativeResolution.setWidth(1280); // Modifying the original object
System.out.println(monitor.getNativeResolution().getWidth()); // Outputs
1280
```