

# Sri Lanka Institute of Information Technology

## Faculty of Computing

SE1020 - Object-Oriented Programming

Year 01 and Semester 02

# Lecture 04

# Relationships and Class Diagram

# Recall...

- The SDLC in an Object-Oriented context
- More about Object-Oriented analysis
- Noun and verb analysis
- CRC cards
- Analysis classes

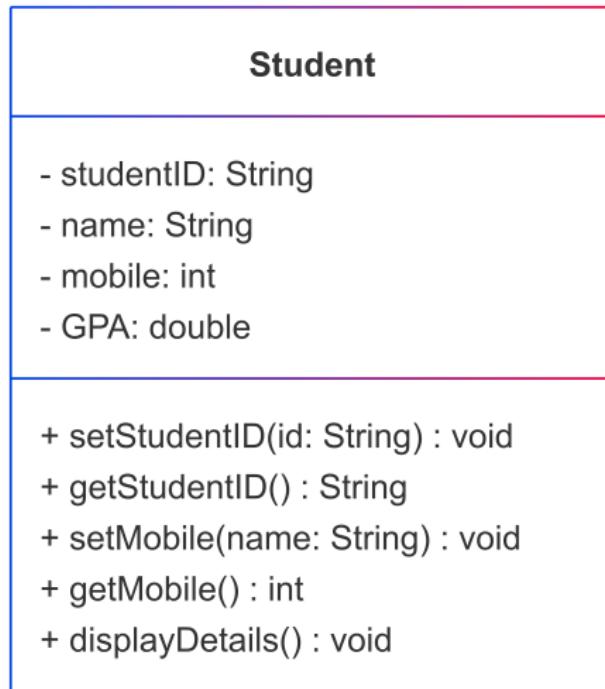
Lecture 04 - Part 01

# Inheritance and Composition

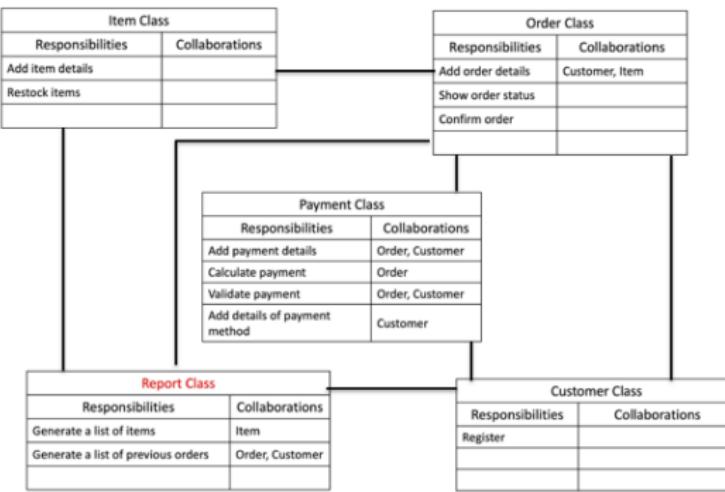
# Learning Objectives

- Understand the different types of relationships between classes.
- Implement inheritance relationship in Java.
- Understand and apply Polymorphism in real-world examples
- Identify and implement composition relationship.

# UML notation - Class

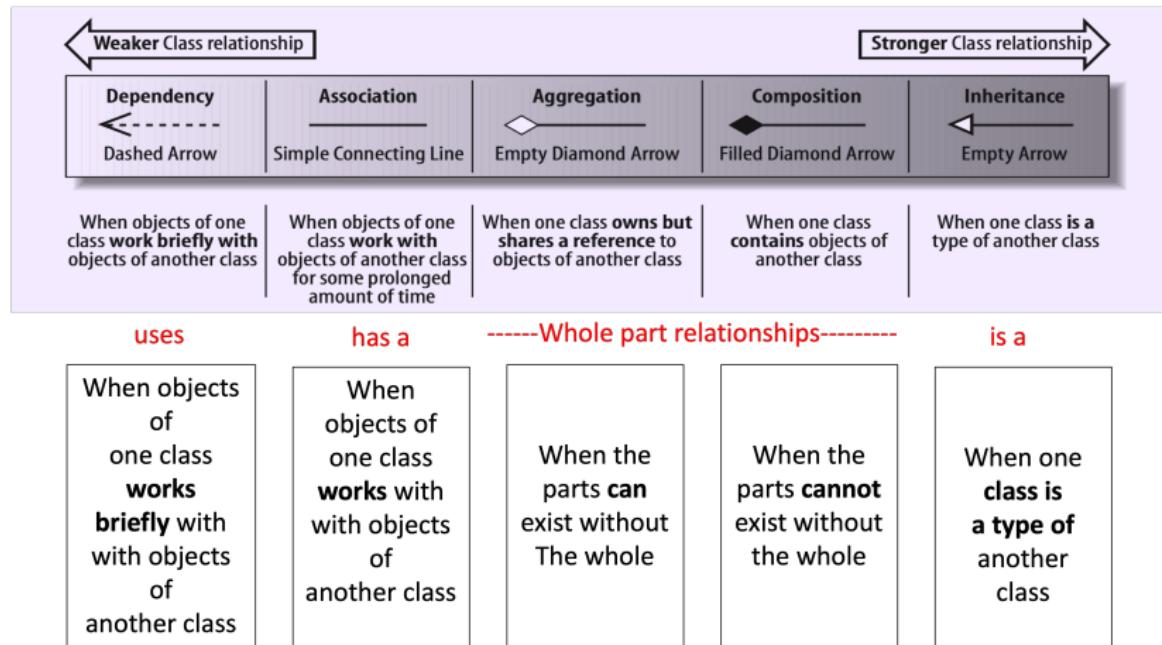


# Relationships between classes



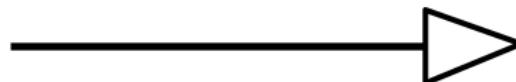
- We can see that there are relationships between classes when we draw CRC cards. We can divide all relationships into five different categories

# Relationships between classes



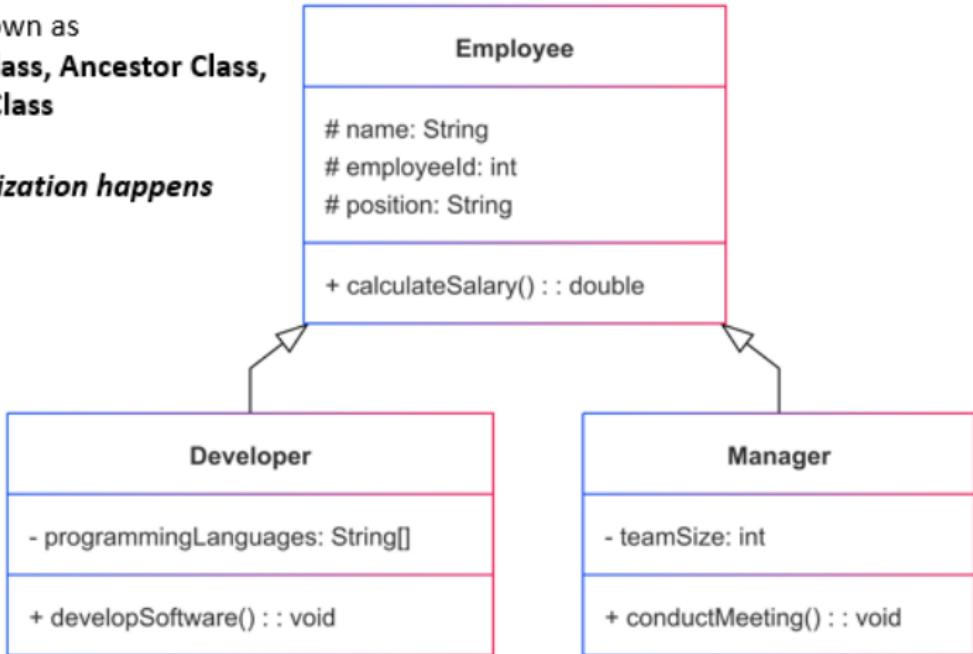
# Relationship Type 01: Inheritance

- Also Known as **Generalization**
- Inheritance represents an "**is-a-kind-of**" relationship in Java.
- It defines a relationship between a **general class (superclass/parent)** and a **more specific class (subclass/child)**.
- The subclass **inherits the properties and behaviors** (attributes and methods) of the superclass.
- In Java, inheritance is implemented using the "**extends**" keyword.
- Graphically, it is often represented with an empty arrow pointing **from the subclass to the superclass** in UML diagrams.



Also known as  
**Super Class, Ancestor Class,  
Parent Class**

*Generalization happens  
here*



Also known as, **Sub Class, Descendant Class, Derived Class**

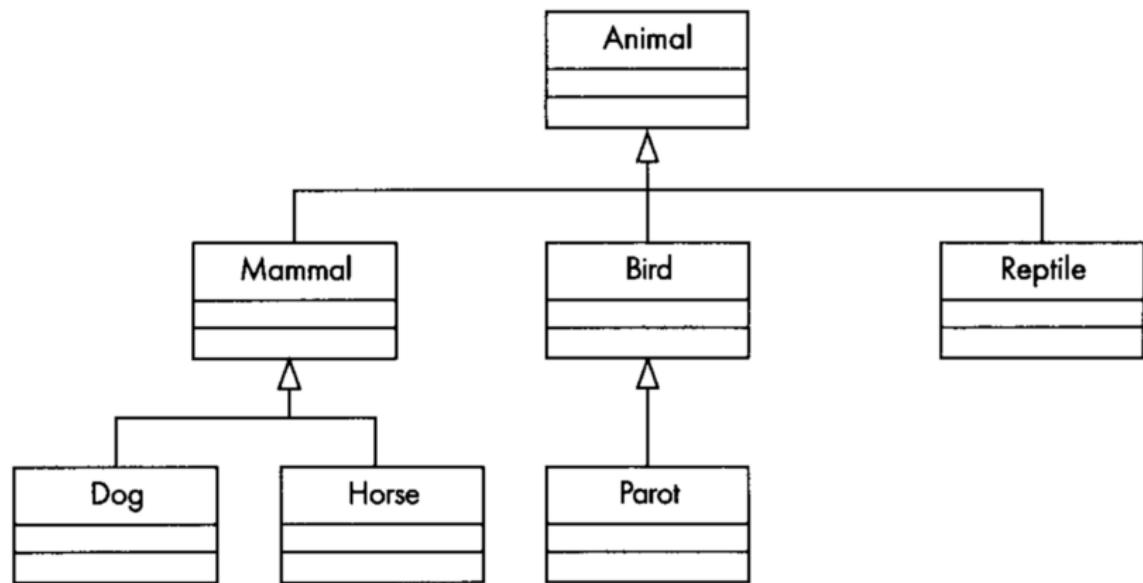
*Specialization happens here*



# Inheritance

- **Super Class / Parent : Employee**
  - **Sub Classes / Children : Manager, Developer**
- 
- Children (Manager and Developer) inherit the properties and methods of its parent class (Employee)
  - The super-class (Employee class) is a generalization of the sub-class (Manager class).
  - The sub-class (Manager class) is a specialization of the super-class (Employee class) .

# Real World Example: Inheritance



# Implementing Inheritance

- Step 01: Implement the Parent class
  - Make the attributes “**protected**” so the child class can access/inherit.



```
class Employee {  
    protected String name;  
    protected int employeeId;  
    protected String position;  
  
    //Default Constructor  
    public Employee() {  
        System.out.println("Employee Constructor called!!!");  
    }  
  
    // Parameterized Constructor  
    public Employee(String pName, int pEmpId, String pPosition) {  
        this.name = pName;  
        this.employeeId = pEmpId;  
        this.position = pPosition;  
    }  
  
    // Method to calculate salary  
    public double calculateSalary() {  
        System.out.println("Calculating salary for Employee...");  
        return 0.0;  
    }  
  
    // Display employee details  
    public void displayInfo() {  
        System.out.println("Name: " + name);  
        System.out.println("Employee ID: " + employeeId);  
        System.out.println("Position: " + position);  
    }  
}
```

# Implementing Inheritance

- Step 02: Implement the Child classes
  - Use the **extends** keyword to implement the inheritance relationship.
  - Declare attributes as **private** so they can only be accessed within the **Manager** or **Developer** class.
  - Define only the attributes specific to the subclass, as the parent class's attributes and methods are automatically inherited.
  - Use the **super()** keyword to pass values for the parent class's attributes to its constructor.
  - Override methods where necessary to customize their behavior in the subclass.

```
class Manager extends Employee {  
  
    private int teamSize;  
  
    //Default Constructor  
    public Manager() {  
        System.out.println("Manager Constructor called!!!");  
    }  
  
    // Parameterized Constructor  
    public Manager(String pName, int pEmpId, String pPosition, int pTeamSize) {  
  
        super(pName, pEmpId, pPosition);  
  
        this.teamSize = pTeamSize;  
    }  
  
    // Manager-specific method  
    public void conductMeeting() {  
        System.out.println(name + " is leading a team of " + teamSize + " members.");  
    }  
  
    //Calculate the salary for a Manager  
    public double calculateSalary() {  
        System.out.println("Calculating salary for Manager...");  
        return 90000.0;  
    }  
}
```

```
class Developer extends Employee {
    private String[] programmingLanguages;

    //Default Constructor
    public Developer() {
        System.out.println("Developer Constructor called!!!");
    }

    // Constructor
    public Developer(String pName, int pEmpId, String pPosition, String[] PLanguages) {
        super(pName, pEmpId, pPosition);
        this.programmingLanguages = PLanguages;
    }

    // Developer-specific method
    public void developSoftware() {
        System.out.println(name + " is developing software using: ");
        for(int i = 0; i < programmingLanguages.length; i++) {
            System.out.println("- " + programmingLanguages[i]);
        }
    }

    //Calculate the salary for a Developer
    public double calculateSalary() {
        System.out.println("Calculating salary for Developer... ");
        return 70000.0;
    }
}
```

```
public class CompanyHierarchy {
    public static void main(String[] args) {
        // Creating a Developer object
        String[] devLanguages = {"Java", "Python", "C++"};
        Developer dev = new Developer("Alice", 101, "Software Developer", devLanguages);

        // Creating a Manager object
        Manager mgr = new Manager("Bob", 102, "Project Manager", 5);

        // Display Developer details and actions
        System.out.println("Developer Details:");
        dev.displayInfo();
        dev.developSoftware();
        System.out.println("Salary: $" + dev.calculateSalary());

        System.out.println("\n-----\n");

        // Display Manager details and actions
        System.out.println("Manager Details:");
        mgr.displayInfo();
        mgr.conductMeeting();
        System.out.println("Salary: $" + mgr.calculateSalary());
    }
}
```

# Output:

```
Developer Details:  
Name: Alice  
Employee ID: 101  
Position: Software Developer  
Alice is developing software using:  
- Java  
- Python  
- C++  
Calculating salary for Developer...  
Salary: $70000.0
```

```
-----  
  
Manager Details:  
Name: Bob  
Employee ID: 102  
Position: Project Manager  
Bob is leading a team of 5 members.  
Calculating salary for Manager...  
Salary: $90000.0
```

# Constructor Chaining

Constructor chaining happens when a constructor of a **subclass** automatically calls the constructor of its **superclass** before executing its own body.

```
class Employee{  
    private int Id;  
  
    public Employee(){  
        System.out.println("Employee Constructor called!");  
    }  
  
    class Manager extends Employee{  
        private int teamSize;  
  
        public Manager(){  
            System.out.println("Manager Constructor called!");  
        }  
  
    }  
  
    public class Company {  
        public static void main(String[] args) {  
            Manager m = new Manager();  
        }  
    }  
}
```

Output:  
Employee Constructor called!  
Manager Constructor called!

## Step 1: Execution starts in main()

```
Manager m = new Manager();
```

This line creates an object of the Manager class. The constructor of **Manager** gets called.

## Step 2: Calling the Manager Constructor

```
public Manager() {  
    System.out.println("Manager Constructor called!");  
}
```

Since Manager extends Employee, before executing Manager constructor, Java first calls the constructor of the immediate parent class (Employee). Even though we don't explicitly call super();, Java automatically inserts super(); as the first line in the constructor.

## Step 3: Calling the Employee Constructor

```
public Employee() {  
    System.out.println("Employee Constructor called!");  
}
```

This constructor gets executed before the Manager constructor because of constructor chaining.

It prints:

**Employee Constructor called!**

## Step 4: Returning to the Manager Constructor

After finishing the Employee constructor, execution returns to the Manager constructor, where the next line is executed:

```
System.out.println("Manager Constructor called!");
```

It prints:

**Manager Constructor called!**

Final Output :

```
Employee Constructor called!  
Manager Constructor called!
```

# Activity 01: Guess the output of the code?

```
class Employee{
    private int Id;

    public Employee(){
        System.out.println("Employee Constructor called!");
    }
}

class Manager extends Employee{
    private int teamSize;

    public Manager(){
        System.out.println("Manager Constructor called!");
    }
}

class AreaManager extends Manager{
    private String allocated_area;

    public AreaManager(){
        System.out.println("AreaManager Constructor called!");
    }
}

public class Company {
    public static void main(String[] args) {
        AreaManager m = new AreaManager();
    }
}
```

# Method Overriding

- Method overriding occurs when a subclass provides a new version of a method that is already defined in its superclass.
- The overridden method in the subclass must have:
  - The same name
  - The same return type
  - The same parametersas the method in the superclass.

```
class Employee{  
    private int Id;  
  
    // Method to calculate salary  
    public double calculateSalary() {  
        System.out.println("Calculating salary for Employee...");  
        return 0.0;  
    }  
}  
  
class Manager extends Employee {  
    private int teamSize;  
  
    //Calculate the salary for a Manager  
    public double calculateSalary() {  
        System.out.println("Calculating salary for Manager...");  
        return 90000.0;  
    }  
}
```



Since **Manager** class inherits from **Employee** class, it also gets the **calculateSalary()** method, but we can change the implementation according to the child class (**Manager**) behaviour .

```
class Developer extends Employee {  
    private int teamSize;  
  
    //Calculate the salary for a Developer  
    public double calculateSalary() {  
        System.out.println("Calculating salary for Developer...");  
        return 70000.0;  
    }  
}
```

The **calculateSalary()** method is called on the **Manager** object (m). Because **Manager** overrides **calculateSalary()**, the overridden method in **Manager** class executes instead of the one in **Employee** class.

```
public class Company {  
    public static void main(String[] args) {  
        System.out.println();  
        Manager m = new Manager();  
        double salary1 = m.calculateSalary();  
        System.out.println("Salary of Manager is : " + salary1);  
        System.out.println();  
  
        Developer dev = new Developer();  
        double salary2 = dev.calculateSalary();  
        System.out.println("Salary of Developer is : " + salary2);  
    }  
}
```



**Output:**

```
Calculating salary for Manager...
Salary of Manager is : 90000.0

Calculating salary for Developer...
Salary of Developer is : 70000.0
```

The **calculateSalary()** method is called on the **Developer** object (dev). Because **Developer** overrides **calculateSalary()**, the overridden method in **Developer** class executes instead of the one in **Employee** class.

- In Java, the method that runs is decided while the program is running, based on the actual type of the object.
- Since m is a Manager object, Java calls the calculateSalary() method from the Manager class, even though the Employee class has the same method.
- This behavior is called **runtime polymorphism**

# Polymorphism

- Polymorphism means "**many forms**" – the ability of a method or object to take different forms.
- In Java, polymorphism allows a single method name to work with different types of objects.
- It helps in code reusability, flexibility, and maintainability.

Example: The **calculateSalary()** method in the Employee class was overridden in the Manager and Developer classes. When the **calculateSalary()** method is called, it executes the version that belongs to the specific object on which it was called.

# A real world example

Consider the request (analogues to a method) “*please cut this in half*”.  
*According to the context the steps of the procedure may be different.*



For a cake:

- Use a knife
- Apply gentle pressure

For a cloth:

- Use a pair of scissors
- Move fingers in a cutting motion

# Types of Polymorphism

## ① Compile-time Polymorphism (Method Overloading)

- Multiple methods with the same name but different parameters.
- The method that will be executed is decided during compile time.

```
class MathOperations {  
    int add(int a, int b) {  
        return a + b;  
    }  
  
    double add(double a, double b) {  
        return a + b;  
    }  
}
```

# Types of Polymorphism

## ② Runtime Polymorphism (Method Overriding)

- A subclass provides a new version of a method from the parent class.
- The program decides which method to execute while it is running.

```
class Employee {  
    void work() {  
        System.out.println("Employee is working.");  
    }  
}  
  
class Developer extends Employee {  
    void work() {  
        System.out.println("Developer is coding.");  
    }  
}
```

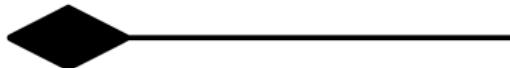
## Activity 02: Guess the output of the code?

```
class Animal {  
    void makeSound() {  
        System.out.println("Animal makes a sound.");  
    }  
}  
  
class Dog extends Animal {  
    void makeSound() {  
        System.out.println("Dog barks.");  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Animal myAnimal = new Dog();  
        myAnimal.makeSound();  
    }  
}
```

Let's try the Question 03 from the  
Tutorial 03

## Relationship Type 02: **Composition**

- Composition means one class is made up of one or more objects of other classes.
- It is a strong form of relationship, where one class contains an object of another class as a part of its attributes.
- **If the containing (whole) object is destroyed, the contained (part) object is also destroyed.**
- In Java, composition is implemented using instance variables that reference other objects.
- In diagrams, composition is shown using a filled diamond connected by a line. The diamond is placed next to the whole (containing) class, indicating a strong relationship between the whole and its parts.

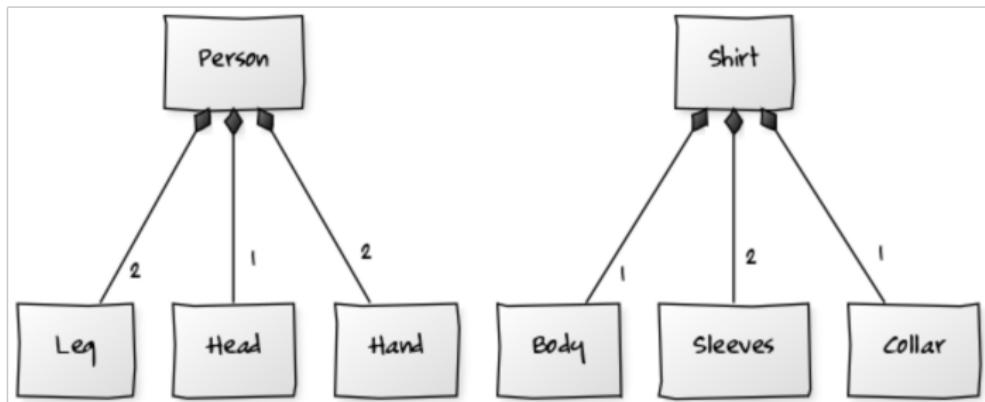


## Example 01



- Whole : Book
- Part : Page
- A Book consists of pages,
- A Page **cannot exist** without the Book.
- Implies that the “Part cannot exist without Whole”

# Part cannot exist without Whole



Head, Hand and Leg are parts of the Person and these ***parts cannot exist*** without a Person

Body, Sleeve, Collar are parts of the Shirt and these ***parts cannot exist*** without a Shirt

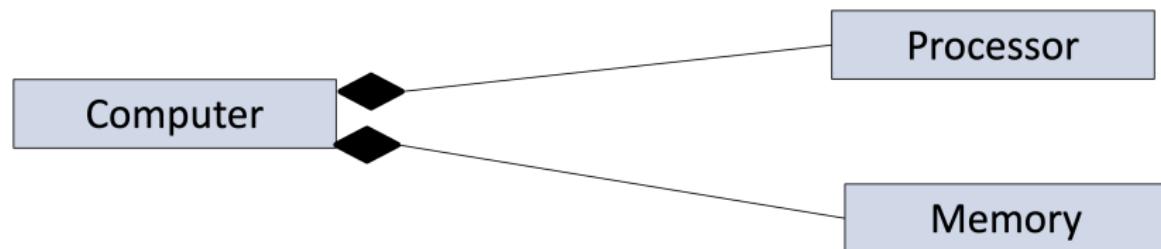
## Example 03



- Whole : Car
- Part : Engine
- A Car consists of an engine,
- An Engine cannot exist without the Car.
- Implies that the “Part cannot exist without Whole”

# Composition Implementation

Consider a Computer that is made up of a Processor and a Memory. A Computer cannot function without its Processor and Memory. If the Computer is destroyed, its Processor and Memory are also destroyed with it. This is a Composition relationship.



# Step 01: Implement the “Part” classes.

Create the Processor and Memory Classes.

```
public class Processor {  
    private String model;  
  
    public Processor(String model) {  
        this.model = model;  
    }  
  
    public void displayProcessor() {  
        System.out.println("Processor Model: " + model);  
    }  
}
```

```
public class Memory {  
    private int sizeGB;  
  
    public Memory(int sizeGB) {  
        this.sizeGB = sizeGB;  
    }  
  
    public void displayMemory() {  
        System.out.println("Memory Size: " + sizeGB + "GB");  
    }  
}
```

## Step 02: Implement the “Whole” class.

The Computer class is composed of a Processor and a Memory. So these objects are created inside the Computer class and belong only to it.

```
public class Computer {  
    private Processor processor;  
    private Memory memory;  
    private String brand;  
  
    public Computer(String pbrand, String pProcessorModel, int pmemorySize) {  
        this.brand = pbrand;  
        processor = new Processor(pProcessorModel);  
        memory = new Memory(pmemorySize);  
    }  
  
    public void displayComputerDetails() {  
        System.out.println("Computer Brand: " + brand);  
        processor.displayProcessor();  
        memory.displayMemory();  
    }  
}
```

## Step 03: Create the Main Class to Run the Program

Implement the Computer object and display its details..

```
public class CompositionExample {  
    public static void main(String[] args) {  
        // Create a Computer object  
        Computer myComputer = new Computer("Dell", "Intel i7", 16);  
  
        // Display computer details  
        myComputer.displayComputerDetails();  
    }  
}
```

Output:

Computer Brand: Dell  
Processor Model: Intel i7  
Memory Size: 16GB

Let's try the Question 04 from the  
Tutorial 03

# Summary

- Relationships define how classes work/communicate together.
- Inheritance allows one class to reuse features of another class.
- Polymorphism allows methods to work in different ways.
- In composition, the whole class object contains multiple part class objects.

# Thank You!