1) Class

Definition:

A **class** in Java is a **blueprint or a template** for creating objects. It can be defined as a **collection of similar types of objects**. The class is a **logical entity** and a **user-defined data type**. It doesn't occupy any memory itself. Instead, it serves as a **template that describes the kinds of state (data or attributes) and behaviour (functions or methods) that the objects of its type will support**. A class is composed of a **name, attributes (also known as fields or instance variables), and methods (also known as member functions)**. Access modifiers like public, private, default, and protected control the accessibility of the class and its members.

Java Code Snippet Example:

```
class Car { // Defining a class named 'Car'
  String model; // Attribute to store the car's model
  String color; // Attribute to store the car's color

  // Method to start the car
  void startEngine() {
    System.out.println("Engine started for " + color + " " + model);
  }

  // Method to apply brakes
  void applyBrakes() {
    System.out.println("Brakes applied for " + color + " " + model);
  }
}
```

In this example, Car is a class that defines the blueprint for cars. It has attributes model and color, and methods startEngine() and applyBrakes().

Real-life Example:

Think of a **blueprint for a house**. The blueprint outlines the design, number of rooms, dimensions, and general features of a house. The blueprint itself isn't a house you can live in; it's the plan. Similarly, a **class is like the blueprint** for creating objects. It defines what an object will look like and what it can do.

2) Object

Definition:

An **object** is a **real-world entity** that has a **state** and **behavior**. It is considered a **building block of OOP** and is commonly known as an **'instance of a class'**. When the Java Virtual Machine (JVM) encounters the new keyword at runtime, it uses the appropriate class to create an object, which becomes a concrete instance of that class. Each object will have its own **state** (the current values of its attributes) and access to all the **behavior** (methods) defined by its class. Objects can interact with each other by invoking each other's methods; this is termed **Message Passing**. Objects are allocated **memory space and an address**.

Java Code Snippet Example:

```
public class Main {
```

```
  public static void main(String[] args) {
    Car myCar = new Car(); // Creating an object of the 'Car' class
    myCar.model = "Sedan";   // Setting the state of the 'myCar' object
    myCar.color = "Red";    // Setting the state of the 'myCar' object
    myCar.startEngine();    // Invoking the behavior of the 'myCar' object

    Car anotherCar = new Car(); // Creating another object of the 'Car' class
    anotherCar.model = "SUV";
    anotherCar.color = "Blue";
    anotherCar.applyBrakes(); // Invoking the behavior of the 'anotherCar' object
  }
}
```

Here, myCar and anotherCar are **objects** of the Car class. Each object has its own model and color (state) and can perform the actions defined by the startEngine() and applyBrakes() methods (behavior).

Real-life Example:

Relating back to the house blueprint, an **object is like an actual house** built using that blueprint. Many identical houses can be built from the same blueprint, and each house is an independent entity with its own address, residents, and current condition (state). Similarly, you can create multiple objects from a single class, and each object is an independent instance with its own set of attribute values.

3) Method

Definition:

A **method** in Java is a **collection of statements that are grouped together to perform a specific operation**. When you call a method, the system executes these statements in order to achieve a particular task. Methods can be created with or without **return values** and can be invoked with or without **parameters** (also known as arguments). A method definition consists of a **method header** (including modifiers, return type, method name, and parameter list) and a **method body** (containing the executable statements). Methods define the **behavior** of the objects of a class.

Java Code Snippet Example:

```
class Calculator {
  // Method to add two integers and return the result
  public int add(int num1, int num2) { // Method header: public, int return type, add name, two int parameters
    int sum = num1 + num2;       // Method body
    return sum;                  // Returning a value
  }

  // Method to print a greeting message (no return value)
  public void greet(String name) {     // Method header: public, void return type, greet name, one String parameter
```

```
    System.out.println("Hello, " + name + "!"); // Method body
  }
}

public class MethodExample {
  public static void main(String[] args) {
    Calculator calc = new Calculator();
    int result = calc.add(5, 3); // Calling the 'add' method and storing the return value
    System.out.println("The sum is: " + result);
    calc.greet("Alice");      // Calling the 'greet' method
  }
}
```

In this example, add() and greet() are **methods** of the Calculator class. The add() method takes two integer parameters, performs addition, and returns an integer value. The greet() method takes a String parameter and prints a greeting message; it does not return any value (indicated by void).

Real-life Example:

Think about the **functions or actions you can perform with a car**. These could include "start the engine," "accelerate," "brake," "turn the steering wheel," etc. Each of these actions is like a **method** associated with the Car object. When you perform one of these actions, the car executes a series of steps to achieve the desired outcome. For instance, the "accelerate" action might involve increasing the fuel supply to the engine and increasing the wheel speed. Similarly, methods define the actions that an object of a class can perform.

Detailed Explanation of OOP Concepts in Java

Here are detailed explanations of Abstraction, Encapsulation, Inheritance, and Polymorphism, along with code snippets, real-life examples, and types, followed by the advantages of OOP.

1) Abstraction

Definition: Abstraction is the process of **hiding internal details and showing only the essential functionality**. It means focusing on what an object does rather than how it achieves it. In Java, a class is also referred to as an **'abstract data type'**.

How to Achieve in Java: Abstraction in Java is primarily achieved through **abstract classes and interfaces**.

- **Abstract Class:** A class declared with the abstract keyword. It can have both abstract methods (methods without a body) and non-abstract methods (methods with a body). An abstract class **cannot be instantiated** and needs to be extended by a subclass that implements its abstract methods.
- **Interface:** A blueprint of a class containing **static constants and abstract methods**. Since Java 8, interfaces can also have default and static methods. An interface **cannot be instantiated** and classes implement interfaces to provide method bodies.

Real-Life Example: Consider a **phone call**. When you make a call, you interact with the phone's interface (dialing numbers, speaking, listening). You are not concerned with the complex internal processing that happens to connect your call. The phone system abstracts away these details, providing you with a simple way to communicate. Similarly, when **driving a bike**, you focus on using the accelerator and brakes without needing to know the intricate workings of the engine or gearbox.

Types of Abstraction: The primary mechanisms for abstraction in Java are:

- **Abstract Classes:** Provide partial implementation and can contain both abstract and non-abstract methods. Subclasses must implement the remaining abstract methods.
- **Interfaces:** Define a contract of methods that implementing classes must adhere to. They promote loose coupling and can support a form of multiple inheritance.

Java Code Snippet (Abstract Class):

```
abstract class Bike {
  abstract void run(); // abstract method
  void changeGear() { // non-abstract method
    System.out.println("gear changed");
  }
}

class Honda4 extends Bike {
  void run() {
    System.out.println("running safely..");
  }

  public static void main(String args[]) {
    Bike obj = new Honda4(); // Upcasting
    obj.run();
    obj.changeGear();
  }
}
```

Java Code Snippet (Interface):

```
interface Drawable {
  void draw(); // abstract method
}

class Rectangle implements Drawable {
  public void draw() {
    System.out.println("drawing rectangle");
  }
}
```

```
class Circle implements Drawable {
  public void draw() {
    System.out.println("drawing circle");
  }
}

class TestInterface1 {
  public static void main(String args[]) {
    Drawable d = new Circle(); // Upcasting through interface reference
    d.draw();
  }
}
```

2) Encapsulation

Definition: Encapsulation is the process of **binding (or wrapping) code and data together into a single unit**. This is done to **protect the data from outside interference** and to **camouflage the implementation from the rest of the program**. In encapsulation, data within a class is **hidden from other functions and classes** ('Data Hiding'). A java **class** is an example of encapsulation. A **Java bean** is considered a fully encapsulated class because all its data members are private.

How to Achieve in Java: Encapsulation in Java is achieved by:

- Declaring the **data members (instance variables) of a class as private**.
- Providing **public methods (getters and setters)** to access and modify the private data members, controlling how the data is used.

Real-Life Example: Think of a **capsule** containing different medicines. The capsule itself encapsulates the various drugs, protecting them and ensuring they are taken together in the correct way. You don't see or directly interact with each individual medicine; you interact with the encapsulated unit. Similarly, in a class, the private variables and the methods that operate on them are encapsulated, controlling access to the data. Consider a **plane** class with a plane_number attribute. If this attribute is private, only the methods within the Plane class can directly access or modify it, providing control over how the plane number is handled.

Types of Encapsulation: While there aren't distinct types of encapsulation in the same way as inheritance or polymorphism, the **level of encapsulation** can vary based on how strictly data hiding is implemented (e.g., making all instance variables private and controlling all access through methods).

Java Code Snippet:

```
class Student6 {
  private int id; // private data member
  private String name; // private data member

  // Constructor to initialize the object
  public Student6(int i, String n) {
```

```
    this.id = i;
    this.name = n;
  }

  // Getter method for id
  public int getId() {
    return id;
  }

  // Setter method for id
  public void setId(int id) {
    this.id = id;
  }

  // Getter method for name
  public String getName() {
    return name;
  }

  // Setter method for name
  public void setName(String name) {
    this.name = name;
  }

  void display() {
    System.out.println(id + " " + name);
  }

  public static void main(String args[]) {
    Student6 s1 = new Student6(111, "Karan");
    s1.display();
  }
}
```

(Note: The example in the source shows copying constructor but illustrates the concept of private members being accessed within the class.)

3) Inheritance

Definition: Inheritance in Java is a mechanism in which **one object acquires all the properties and behaviors of a parent object**. It represents the **IS-A relationship**, also known as the parent-child relationship. The **superclass** (or base class or parent class) is the class being inherited from, and the **subclass** (or derived class or child class) is the class that inherits. Inheritance promotes **code reusability**.

How to Achieve in Java: Inheritance is achieved using the extends keyword. The syntax is class Subclass-name extends Superclass-name { // methods and fields }.

Real-Life Example: Consider the class **"Plane"** with common attributes like plane_number, total_seats, etc.. Specific types of planes, like **"Jet"**, can inherit these common attributes from the "Plane" class and then have their own distinct attributes like luxury_arrangements. This avoids redefining the common properties for each type of plane. An **Airport** "HAS-A" Plane, where Plane is a more general concept, and specific plane types inherit from it.

Types of Inheritance in Java:

- **Single Inheritance:** A subclass inherits from only one superclass.
- class Animal {
- void eat() {
- System.out.println("eating...");
- }
- }
-
- class Dog extends Animal {
- void bark() {
- System.out.println("barking...");
- }
- }
- **Multilevel Inheritance:** A subclass inherits from a superclass, and then another class inherits from that subclass, forming a chain.
- class Animal {
- void eat() {
- System.out.println("eating...");
- }
- }
-
- class Dog extends Animal {
- void bark() {
- System.out.println("barking...");

```
    }
}

class BabyDog extends Dog {
    void weep() {
        System.out.println("weeping...");
    }
}
```

- **Hierarchical Inheritance:** Multiple subclasses inherit from a single superclass.

```
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("barking...");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("meowing...");
    }
}
```

- **Hybrid Inheritance:** A combination of two or more of the above types. Java **does not directly support multiple inheritance** (inheriting from more than one class) to avoid the "diamond problem," but it can be achieved through interfaces.

4) Polymorphism

Definition: Polymorphism literally means **"many forms"**. In OOP, it is the **ability of an object to take on many forms**. It is expressed by **'one interface, many methods'**. Polymorphism allows you to perform a single action in different ways.

How to Achieve in Java: Polymorphism in Java is achieved through:

- **Method Overloading (Compile-time Polymorphism or Static Binding or Early Binding):** Having multiple methods in the same class with the **same name but different parameter lists** (different

number of parameters, different types of parameters, or different order of parameters). The compiler determines which method to call based on the arguments passed at the time of the method call. **Constructor overloading** is also a form of method overloading.

- **Method Overriding (Run-time Polymorphism or Dynamic Binding or Late Binding):** When a subclass provides a **specific implementation for a method that is already defined in its superclass**. The method in the subclass must have the **same name and the same parameter list** as the method in the superclass. The decision of which method to execute is made at runtime based on the actual object type.

Real-Life Example: Consider the action of **speaking**. A **cat** speaks by "meowing," a **dog** speaks by "barking". The action of "speaking" has different forms depending on the type of animal. Similarly, in programming, a method like seatAvailability() in a general **"Plane"** class might have a basic implementation, but a specialized subclass like **"Jet"** might override this method to provide a specific implementation (e.g., always returning 0 if it's booked as a private jet). A **button click** can result in different actions depending on which button is clicked; this can be implemented using polymorphism with event listeners.

Types of Polymorphism in Java:

- **Compile-time Polymorphism (Method Overloading):** The compiler resolves which overloaded method to call based on the method signature at compile time.
- class Adder {
- static int add(int a, int b) {
- return a + b;
- }
-
- static int add(int a, int b, int c) {
- return a + b + c;
- }
- }
-
- class TestOverloading1 {
- public static void main(String[] args) {
- System.out.println(Adder.add(11, 11)); // Calls add(int, int)
- System.out.println(Adder.add(11, 11, 11)); // Calls add(int, int, int)
- }
- }
- **Run-time Polymorphism (Method Overriding):** The JVM resolves which overridden method to call based on the actual object's type at runtime. This requires inheritance and method overriding.
- class Vehicle {
- void run() {
- System.out.println("Vehicle is running");
- }

```
}

class Bike2 extends Vehicle {
  void run() {
    System.out.println("Bike is running safely");
  }

  public static void main(String args[]) {
    Vehicle obj = new Bike2(); // Upcasting
    obj.run(); // Calls Bike2's run() method
  }
}
```

Advantages of OOPs

The sources highlight several advantages of Object-Oriented Programming:

- **Code Reusability:** Inheritance allows reusing public methods and fields of a base class without rewriting the same code. Common code can be moved to a superclass, minimizing duplication.
- **Extensibility:** Inheritance allows extending the base class logic as per the business logic of the derived class.
- **Data Hiding:** Base classes can keep some data private so that it cannot be altered by the derived class, enhancing security. Encapsulation protects data from outside interference.
- **Polymorphism:** Allows one interface to represent many different underlying forms, reducing complexity. Method overriding enables specific implementations in subclasses.
- **Simpler Software Development and Maintenance:** OOP simplifies software development by organizing code into objects and classes that model real-world entities. Changes in one part of the program are less likely to affect other independent objects, making maintenance easier.
- **Better Organization of Code:** Inheritance and encapsulation lead to a better organization of code and smaller, simpler compilation units.
- **Flexibility to Change:** Classes inheriting from a common superclass can be used interchangeably, making application code more flexible.
- **Enhanced Security:** Data hiding through encapsulation and access modifiers provides enhanced security by controlling access to sensitive data.
- **Ability to Model Real-World Problems:** OOP allows programs to closely resemble real-world scenarios by representing entities as objects with states (data) and behaviors (methods).
- **Modularity:** OOP promotes a modular approach where data and functions are combined into objects.
- **Convenient for Complex Problems:** OOP provides tools to manage and solve complex problems effectively.
- **Bottom-Up Approach:** OOP often follows a bottom-up approach, focusing on identifying and defining basic parts (objects and classes) first and then building more complex systems by allowing these parts to interact.

- **Multiple Instances:** Multiple instances (objects) can be created from a single class.