**Object-Oriented Programming (OOP)** concepts using a **Railway System**.

**1. Class:**

A **class** is like a **blueprint** or a **template** for creating objects. It defines the **properties** (attributes) and **behaviors** (methods) that the objects of that class will have.

In the Railway System, a **Train** can be thought of as a class. The Train class defines the properties (like trainNumber, origin, destination, etc.) and methods (like depart(), arrive(), stop()) that all trains will have.

**Example**:

class Train {

String trainNumber;

String origin;

String destination;

void depart() {

System.out.println(trainNumber + " is departing from " + origin);

}

void arrive() {

System.out.println(trainNumber + " has arrived at " + destination);

}

void stop() {

System.out.println(trainNumber + " is stopping.");

}

}

**2. Object:**

An **object** is an **instance** of a class. It is created from a class and represents an actual entity in the system, with its own specific data.

In the Railway System analogy, an object could be an **actual train** that operates on a particular route. For instance, a train object could represent **Train 101** traveling from **New York** to **Los Angeles**.

**Example**:

public class Main {

public static void main(String[] args) {

Train myTrain = new Train(); // Creating an object of the Train class

myTrain.trainNumber = "Train 101";

myTrain.origin = "New York";

myTrain.destination = "Los Angeles";

myTrain.depart(); // Output: Train 101 is departing from New York

}

}

**3. Abstraction:**

**Abstraction** is the concept of **hiding complexity** and exposing only the **essential details** to the user. It allows the user to interact with the system without needing to understand the complex details behind it.

In the Railway System, when a passenger boards a train, they don't need to know the internal workings of how the train operates or how signals are managed. They just need to know how to board the train, where it’s going, and when it will depart.

In terms of coding, we can create an abstract class or interface to define high-level functionality and hide complex implementation details.

**Example**:

abstract class Train {

abstract void depart(); // Abstract method - No implementation here

abstract void stop(); // Abstract method - No implementation here

void showSchedule() {

System.out.println("The train schedule is displayed.");

}

}

class ExpressTrain extends Train {

void depart() {

System.out.println("Express Train is departing.");

}

void stop() {

System.out.println("Express Train is stopping.");

}

}

class Main {

public static void main(String[] args) {

Train myTrain = new ExpressTrain(); // Creating an object of the derived class

myTrain.depart(); // Output: Express Train is departing.

myTrain.showSchedule(); // Output: The train schedule is displayed.

}

}

In this example, the Train class provides an abstraction, hiding the complex details of how trains depart or stop. The ExpressTrain class provides specific implementations of these actions.

**4. Encapsulation:**

**Encapsulation** is the practice of **bundling data and methods** that operate on the data into a single unit (the class) and restricting access to some of the object's components. This is usually done by making some fields **private** and providing **public getter and setter methods** to access or modify the fields.

In the Railway System analogy, encapsulation could involve **hiding the internal state of the train**, such as its current speed, and providing methods to control or retrieve that state.

**Example**:

class Train {

private int speed; // Private field to store speed, cannot be accessed directly

// Public method to set the speed

public void setSpeed(int speed) {

if (speed > 0) {

this.speed = speed;

System.out.println("Train speed set to " + speed + " km/h.");

} else {

System.out.println("Invalid speed.");

}

}

// Public method to get the speed

public int getSpeed() {

return speed;

}

}

class Main {

public static void main(String[] args) {

Train myTrain = new Train();

myTrain.setSpeed(80); // Setting speed through setter method

System.out.println("Current speed: " + myTrain.getSpeed() + " km/h"); // Accessing speed through getter method

}

}

**5. Polymorphism:**

**Polymorphism** means **"many forms"** and refers to the ability of different objects to respond to the same method call in different ways. This is typically achieved through **method overriding** (runtime polymorphism) or **method overloading** (compile-time polymorphism).

In the Railway System analogy, different types of trains (like **ExpressTrain**, **PassengerTrain**, **FreightTrain**) can all respond to the depart() method, but each train may behave differently when the depart() method is called.

**Example**:

class Train {

void depart() {

System.out.println("Train is departing.");

}

}

class ExpressTrain extends Train {

@Override

void depart() {

System.out.println("Express Train is departing at high speed!");

}

}

class PassengerTrain extends Train {

@Override

void depart() {

System.out.println("Passenger Train is departing with passengers.");

}

}

class FreightTrain extends Train {

@Override

void depart() {

System.out.println("Freight Train is departing with goods.");

}

}

class Main {

public static void main(String[] args) {

Train train1 = new ExpressTrain();

Train train2 = new PassengerTrain();

Train train3 = new FreightTrain();

train1.depart(); // Output: Express Train is departing at high speed!

train2.depart(); // Output: Passenger Train is departing with passengers.

train3.depart(); // Output: Freight Train is departing with goods.

}

}

In this example, **method overriding** allows different types of trains to implement their own version of the depart() method. This is a classic example of **runtime polymorphism**, where the method that is executed depends on the **actual object type** (whether it's ExpressTrain, PassengerTrain, or FreightTrain).

**6. Inheritance:**

**Inheritance** is a mechanism where a new class (called the **subclass** or **child class**) inherits properties and behaviors (methods) from an existing class (called the **superclass** or **parent class**). It allows you to create a new class based on an existing class, with the possibility of **extending** or **modifying** its functionality.

In the Railway System analogy, a **PassengerTrain** might inherit basic behavior from a general **Train** class but also have additional functionality specific to passenger trains (like serveSnacks()).

**Example**:

class Train {

String trainNumber;

String origin;

String destination;

void depart() {

System.out.println(trainNumber + " is departing.");

}

void stop() {

System.out.println(trainNumber + " is stopping.");

}

}

class PassengerTrain extends Train {

void serveSnacks() {

System.out.println("Serving snacks to passengers.");

}

}

class Main {

public static void main(String[] args) {

PassengerTrain myPassengerTrain = new PassengerTrain();

myPassengerTrain.trainNumber = "Passenger Train 1";

myPassengerTrain.depart(); // Inherited behavior from Train class

myPassengerTrain.serveSnacks(); // Specific to PassengerTrain class

}

}

In this example, the **PassengerTrain** class **inherits** properties and methods from the **Train** class and also **extends** its functionality by adding the serveSnacks() method.

**Summary of OOP Concepts in the Railway System:**

1. **Class**: A blueprint or template for creating train objects (e.g., Train class).
2. **Object**: An instance of a class representing a specific train (e.g., Train 101).
3. **Abstraction**: Hiding the complexity of train operations from passengers and exposing only essential methods like depart().
4. **Encapsulation**: Protecting internal train data (e.g., speed) and providing controlled access via getter and setter methods.
5. **Polymorphism**: The ability of different types of trains (e.g., ExpressTrain, PassengerTrain, FreightTrain) to respond to the same method (depart()) in different ways.
6. **Inheritance**: A specific type of train (e.g., PassengerTrain) inherits general properties and behaviors from the Train class and can add or modify its own behavior.