Object Oriented Programming System

Introduction to Classes & Objects

- Classes is a collection of variable, functions.
- It is not a real world entity and is just a template / blueprint on which new object should be created.
- Does not occupy any memory. (Think of this as how Animal is not a real world entity and does not occupy any space but a object of type animal like dog would occupy space in the real world.)
- Syntax for declaring classes →

```
access-modifier class ClassName{
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
      // body of method
    }
    type methodname2(parameter-list) {
      // body of method
    }
    // ...
    type methodnameN(parameter-list) {
        // body of method
    }
}
```

- The data, or variables, defined within a class are called instance variables.
- Each instance of a class contains their own copy of the variables.
- Example class →

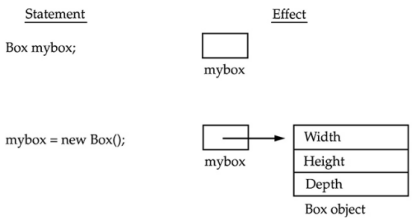
```
class Student{
  // instance variables
  int rollNumber;
  String name;

  // function/ methods
  void setRollNumber(int x){
    rollNumber = x;
  }
}
```

Creating Objects

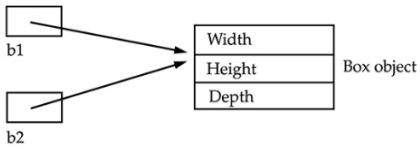
- A new object is created →

```
Student student1; // declare reference to object
student1 = new Student(); // allocate memory to a Student object
```

- The new operator dynamically allocates memory for an object during runtime.
- Also mybox will store the memory address only and data is actually in heap memory.




```
Box b1 = new Box();
Box b2 = b1; // b2 will refer to the same memory address
```




```
// if we do this then b2 will still point to the original
// reference of b1 only
b1 = null;
```

Constructors

- A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the new operator completes.
- Used to do setup at the time an object is created.

```
Student student1 = new Student();
// () -> the constructor is called
```

- If no constructor is available inside a class java already has a default constructor available that will put default values to all the variables.
- The implicit return type of a class’ constructor is the class type itself.

```
class Student{
  int rollNumber;
  String name;

  // assigned passed values to the instance of the class
  Student(int no,String c){
```

```
    rollNumber = no;
    name = c;
  }
}
```

This Keyword

- The `this` key word is used to refer to the current instance of the object inside a class.
- Example →

```
class Student{
  int rollNumber;
  String name;

  // this will refer to the vairable part of the class
  Student(int rollNumber,String name){
    // rhs has the local variables
    this.rollNumber = rollNumber;
    this.name = name;
  }
}
```

- Can also use to invoke methods of the current class.

Static Keyword

- Anything declared with the `static` keyword belong to the class and not to the instance of the class.
- Ex some variable that should remain the same across all objects should be decalred static.

```
class Student{
 static float CGPA_FACTOR = 9.5; // this should ideally remain the same for all students and no use in
// creating different variables for differnt student objects
  // functions
}
```

- Another example could be of a counter of number of students which in the constructor will increase it by 1 every time a new object is created. This should be declared `static` since this memory should be shared across all objects.

Static methods

- A static method belongs to the class and can be called without creating a new instance of the class.
- Example →

```
class Student{
  static String collegeName = "NAME";
  // static variable can only be changed in a static method
  static void change(){
    this.collegeName = "CHANGED";
  }
}

class Test{
  public static void main(String args[]){
    Student.change();// this can be directly called
  }
}
```

```
class Calculate{
  static int cube(int x){
    return x*x*x;
  }

  public static void main(String args[]){
    int result = Calculate.cube(5);
    System.out.println(result);
  }
}
```

- There are two main restrictions for the static method. They are:
  - The static method can not use non static data member or call non-static method directly.
  - this and super cannot be used in static context.

```
// will throw compile time error
class A{
  int a = 40; //non static variable
  public static void main(String args[]){
   System.out.println(a);
  }
}
```

Inheritance

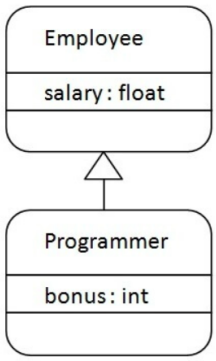
- **Inheritance** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents a **IS-A relationship** which is also known as a **parent child relationship**.
- Terms used in inheritance →
  - **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
  - **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- Syntax for inheritance →

```
class Subclass extends Superclass {
    // extends -> extend the functionality of the class
    // other methods and fields
}
```

- Programmer is the subclass and Employee is the superclass in this example. The relationship between the two classes is **Programmer IS-A Employee.**

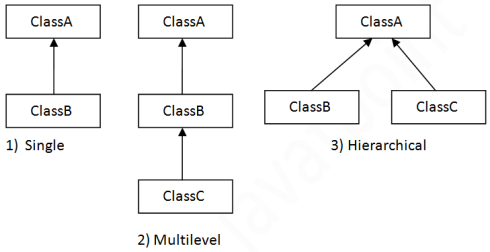
```
class Employee {
    float salary = 40000;
}
class Programmer extends Employee {
    int bonus = 10000;
    public static void main(String args[]) {
        Programmer p = new Programmer();
```

```
        System.out.println("Programmer salary is:" + p.salary); // 40000
        System.out.println("Bonus of Programmer is:" + p.bonus); // 10000
    }
}
```



Types of Inheritance in Java

- **Single Inheritance** → When a class inherits another class, it is known as a *single inheritance.* In the example given below, Dog class inherits the Animal class, so there is the single inheritance.




```
class Animal {
    // every animal eats
    void eat() {
        System.out.println("eating...");
    }
}
class Dog extends Animal {
    // only dogs will bark however
    void bark() {
        System.out.println("barking...");
    }
}
class TestInheritance {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.bark();
        d.eat();
    }
}
```

- **Multilevel Inheritance** → When there is a chain of inheritance, it is known as *multilevel inheritance.* In the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
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...");
    }
}
class TestInheritance2 {
    public static void main(String args[]) {
        BabyDog d = new BabyDog();
        d.weep();
        d.bark();
        d.eat();
```

```
    }
}
```

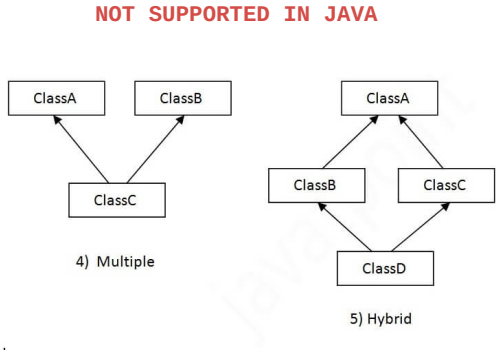
- **Hierarchical inheritance** → When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

```
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...");
    }
}
class TestInheritance3 {
    public static void main(String args[]) {
        Cat c = new Cat();
        c.meow();
        c.eat();
        //c.bark(); will thouw a compile time error
    }
}
```

Why multiple inheritance is not supported in java?

- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

```
class A {
    void msg() {
        System.out.println("Hello");
    }
}
class B {
    void msg() {
        System.out.println("Welcome");
    }
}
class C extends A, B { // agar aisa kar sakt e thos kya hota

    public static void main(String args[]) {
        C obj = new C();
        obj.msg(); //Now which msg() method should be invoked?
    }
}
```



- Java does not support mulitple inheritance through the use of classes **but can be done through interfaces.**

Polymorphism

- **Poly → Many , morphism → ways to represent.**

Method Overloading

- If a class has multiple methods having same name but different parameters, it

```
class Adder {
    static int add(int a, int b) {
```

is known as **Method Overloading.**

- Done at compile time by the compiler.
- Can’t acheive method overloading by changing just the return type of method.

```
// there will be ambiguity in this case and will
// give a compile time error
class Test{
  int add(int a,int b) return a + b;
  double add (int a,int b) return a + b;
}
```

- Can also be achieved by changing the data type of arguments/parameters.
- Changing sequence of arguments will also work.
- `main()` method can also be overloaded but the main main will work. To run other will have to create a new instance.

```
        return a + b;
    }
    static int add(int a, int b, int c) {
        return a + b + c;
    }
}

class Test{
    public static void main(String[] args) {
        // will call the one with 2 arguments
        System.out.println(Adder.add(11, 11));
          // three arguments
        System.out.println(Adder.add(11, 11, 11));
    }
}
```

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

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java.**
- In other words, if a subclass provides the specific implementation of the method that has been declared by one of its parent class.
- Number of arguments need to be same to acheive overriding.
- **Done at runtime by JVM.**

```
class Shapes {
    void area() {
        print("inside shapes area");
    }
}

class Circle {
    void area() {
        print("pi * r * e");
    }
}

class Square {
    void area() {
        print("side * side");
    }
}

class Rectangle {
    void area() {
        print("length * breadth");
    }
}

class Test{
    main(){
        Shapes shape = new Shapes();
        Circle circle = new Circle();
        shape.show(); // shape vaala
        circle.show(); // circle vaala show hoga
    }
}
```

Runtime Polymorphism

- It is a process in which a function call to a overridden method is resolved at runtime.
- Parent reference ke through hum child mein jo kuch bhi overriden h usko use kar sakte h but child ke self declared functionality use nahi kar sakte.

```
// Java Program for Method Overriding

// Class 1
// Helper class
```

```
class Parent {

  // Method of parent class
  void Print()
  {

    // Print statement
    System.out.println("parent class");
  }
}

// Class 2
// Helper class
class subclass1 extends Parent {

  // Method
  void Print() { System.out.println("subclass1"); }
}

// Class 3
// Helper class
class subclass2 extends Parent {

  // Method
  void Print()
  {

    // Print statement
    System.out.println("subclass2");
  }
}

// Class 4
// Main class
class GFG {

  // Main driver method
  public static void main(String[] args){

    // Creating object of class 1
    Parent a;

    // Now we will be calling print methods
    // inside main() method

    a = new subclass1();
    a.Print(); // subclass1

    a = new subclass2();
    a.Print(); // subclass2
  }
}
```

Super Keyword

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Uses of super →
  - Refer to data members of parent class.

```
class Animal {
    String color = "white";
}
class Dog extends Animal {
    String color = "black";
    void printColor() {
        System.out.println(color); // black
        System.out.println(super.color); // white
    }
}
class TestSuper1 {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.printColor();
    }
}
```

- To Invoke parent class method

```
class Animal {
    void eat() {
        System.out.println("eating...");
    }
}
class Dog extends Animal {
    void eat() {
        System.out.println("eating bread...");
    }
    void bark() {
        System.out.println("barking...");
    }
    void work() {
        super.eat(); // invoke the eat method of parent class Animal
        bark();
    }
}
class TestSuper2 {
    public static void main(String args[]) {
        Dog d = new Dog();
        d.work();
    }
}
```

- To Invoke parent class constructor

```
class Person {
    int id;
    String name;
    Person(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
class Emp extends Person {
    float salary;
    Emp(int id, String name, float salary) {
        super(id, name); //reusing parent constructor
        this.salary = salary;
    }
    void display() {
        System.out.println(id + " " + name + " " + salary);
    }
}
class TestSuper5 {
    public static void main(String[] args) {
        Emp e1 = new Emp(1, "ankit", 45000 f);
        e1.display();
    }
}
```

Final Keyword

Final variable

- Can’t change the value of a final variable.
  - **This code will give compile time error.**

```
class Bike {
    final int speedlimit = 90; //final variable
    void run() {
        speedlimit = 400; // can't do this
    }
    public static void main(String args[]) {
        Bike obj = new Bike();
        obj.run();
    }
}
```

Final Method

- A method declared `final` can’t be overriden.
  - This will throw compile time error.

```
class Bike {
    final void run() { // declared as final
        System.out.println("running");
    }
}

class Honda extends Bike {
    void run() {
        System.out.println("running safely with 100kmph");
    }

    public static void main(String args[]) {
        Honda honda = new Honda();
        honda.run();
    }
}
```

Final Class

- A class declared as `final` cannot be inherited from.
  - This will throw compile time error.

```
final class Bike {}

class Honda1 extends Bike {
    void run() {
        System.out.println("running safely with 100kmph");
    }

    public static void main(String args[]) {
        Honda1 honda = new Honda1();
        honda.run();
    }
}
```

Abstraction & Interfaces

Inheritance and polymorphism helps us in code reusability. Abstraction, Interfaces & Encapusaltion helps in code security.

Abstraction

- In java abstraction can be acheived by using abstract classes(0 - 100%) or by using interfaces(100%).
- A method without a body(no implementation) is known as abstract method.
- If a class has an `abstract` method then it should be declared `abstract` as well. Abstract classes don’t need to necessarily have all methods as abstract.

```
abstract class Vehicle {
    int tyres;
    abstract void start();
}

class Car extends Vehicle {
    void start() {
        print("Started using key");
    }
}

class Scooter extends Vehicle {
    void start() {
```

```
        print("Started using kick");
    }
}

class Test {
    main(){
        Car c = new Car();
        c.start(); // started using key
}
```

- The methods in abstract classes are meant to be overriden in derived classes and the derived classes need to implement all of them.
- **A object of abstract class cannot be created but a reference can be created.**

Interfaces

- Interfaces are similar to abstract classes but have all their methods declared as abstract type.
- Interfaces are a blueprint for a class which tells a class what to do and not how to do.
- **Using interfaces we can implement multiple inheritance in java.**
- Syntax →

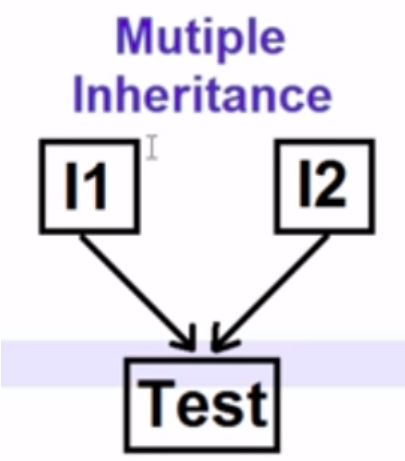
```
  interface interfaceName {
    // compiler automatically adds public abstract keyword in front of methods
    void method();

    // fields will be public static final by default
    int a = 10;
}
```

- Example →

```
interface Vehicle{
    void run();
}

class Car implements Vehicle{
    // jo function implement kar rahe ha uska access modifier ya toh fir same ya usse upar ka hona chaiye
    public void run(){
        print("car running");
    }
}
```

Implementing Multiple Inheritance using interfaces

```
interface Printable {
    void print();
}
interface Showable {
    void show();
}
class Test implements Printable, Showable {
    public void print() {
        System.out.println("Hello");
    }
    public void show() {
        System.out.println("Welcome");
    }

    public static void main(String args[]) {
        Test obj = new Test();
        obj.print();
        obj.show();
    }
}
```



- This works because there is no case of ambiguity here like there was in the case of multiple inheritance because the implementation will be provided by the class that is implementing the interface.

Encapsulation

Abstraction	Encapsulation
Detail Hiding (internal Implementation hiding)	Data Hiding (Information hiding)
Data abstraction deals with exposing the interface/functionality to the user and hiding the details of the implementation.	Encapsulation groups together data and methods that act upon the data.

- Class is encapsulation only. Declare variables as private and then provide with getter and setters accordingly.

Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y