OOPS Introduction And Basics

- What is OOPS?
    - **Object**: An object is an instance of a class. It represents a real-world entity and encapsulates the data (attributes) and behaviors (methods) defined by its class.
    - **Classes**: A class is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of the class will have.

```
// Defining a class
public class Car {
    // Attributes (fields)
    String brand;
    String model;
    int year;

    // Constructor
    Public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    // Method
    public void displayInfo() {
        System.out.println("Car: " + brand + " " + model + " (" + year + ")");
    }
}

// Creating objects of the Car class
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Maruti", "Swift", 2016);
        myCar.displayInfo();
    }
}
```

    - **Encapsulation**: It helps in hiding the internal state of objects from the outside world and only exposes necessary functionalities.
    - **Inheritance**: one class inherits properties and methods of another class.
    - **Polymorphism** :Polymorphism means the ability of an object to take on different forms. In Java, polymorphism is achieved through method overriding (runtime polymorphism) and method overloading (compile-time polymorphism).

- **Introduction of Abstraction, Encapsulation and Inheritance**
  In Java, Abstraction, Encapsulation, and Inheritance are three key principles of object-oriented programming (OOP) that facilitate building modular, reusable, and maintainable software. Let's delve deeper into each of these concepts:

  1. Abstraction:
  Abstraction is the process of hiding the implementation details and showing only the essential features of an object. It helps in managing complexity by focusing on what an object does rather than how it does it. In Java, abstraction is achieved through abstract classes and interfaces.

  Ways to achieve Abstraction
  There are two ways to achieve abstraction in java

  Abstract class (0 to 100%)
  Interface (100%)

  Abstract Class:

  A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. Abstract classes can provide a blueprint for other classes to extend.

  An abstract class must be declared with an abstract keyword.
  It can have abstract and non-abstract methods.
  It cannot be **instantiated or can't create objects**.
  It can have constructors and static methods also.

  It can have final methods which will force the subclass not to change the body of the method.

Example:

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Example:

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction {
```

```
public static void main(String args[]){
Shape s=new Circle();//In a real scenario, object is provided through method,
e.g., getShape() method
s.draw();
}
}
```

Example:

```
public abstract class Animal {
    public abstract void makeSound();

    public void sleep() {
        System.out.println("Zzz");
    }
}

public class Dog extends Animal {
    public void makeSound() {
        System.out.println("Woof");
    }
}


Animal myDog = new Dog();
myDog.makeSound();  // Outputs: Woof
myDog.sleep();      // Outputs: Zzz
```

Interface:

An interface in Java is a reference type that can contain only method signatures (implicitly abstract), constants, default methods, static methods, and nested types.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method bodies. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Since Java 8, we can have default and static methods in an interface.
Since Java 9, we can have private methods in an interface.

Example:

```
interface A{
void a();
void b();
void c();
void d();
}

abstract class B implements A{
public void c(){System.out.println("I am c");}
}

class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Example:

```
interface Drawable{
void draw();
}

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 TestInterface{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g.
getDrawable()
d.draw();
}}
```

Example:

```
interface PrintData{
void print();
}
interface ShowData{
void show();
}
class A7 implements PrintData,ShowData{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Example:

```
interface PrintData{
void print();
}
interface ShowData extends PrintData{
```

```
void show();
}
class TestInterface implements ShowData{
public void print(){System.out.println("Print Data");}
public void show(){System.out.println("Show Data");}

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

```
///////////////////////////////////
public interface MyInterface {
   // Method signatures
   void method1();
   int method2(String str);
}
```

////////////////////////
Constants: Interfaces can have constant fields, which are implicitly public, static, and final. These fields must be initialized with a value.

```
public interface MyInterface {
   int MAX_VALUE = 999;
   String MODE = "DEFAULT";
}
```

Default Method:
Starting from Java 8, interfaces can have default methods, which provide a default implementation that can be overridden by implementing classes:

```
public interface MyInterface {
   void method1();  // Abstract method

   default void defaultMethod() {
      System.out.println("This is a default method");
   }
}
```

Static method:

```
public interface MyInterface {
    static void staticMethod() {
        System.out.println("This is a static method");
    }
}
```

Implement interface as:

```
public class MyClass implements MyInterface {
    @Override
    public void method1() {
        System.out.println("Implementation of method1");
    }

    @Override
    public int method2(String str) {
        System.out.println("Implementation of method2 with arg: " + str);
        return str.length();
    }
}

////////////////////
MyInterface obj = new MyClass();
obj.method1();  // Calls overridden method in MyClass
obj.method2("Hello");  // Calls overridden method in MyClass
```

Java8 Default & Static Method:

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
static int square(int x){return x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
System.out.println(Drawable.square(2));
}}
```

2. Encapsulation:
Encapsulation is the bundling of data (attributes) and methods (behaviors) that operate on the data into a single unit (class). It restricts direct access to some of the object's components and protects the integrity of the data. In Java, encapsulation is achieved using access modifiers (private, protected, public).

Access Modifiers:
Java provides four access modifiers:

private: Accessible only within the same class.
default (no modifier): Accessible within the same package.
protected: Accessible within the same package and subclasses.
public: Accessible from anywhere.
Example:

```
public class Person {
   private String name; // Private field

   public String getName() { // Public getter method
      return name;
   }

   public void setName(String newName) { // Public setter method
      if (newName != null && !newName.isEmpty()) {
         this.name = newName;
      }
   }
}
```

Example:

```
public class Car {
   private String brand;
   private String model;
   private int year;
```

```
    // Constructor
    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    // Getter methods (Accessors)
    public String getBrand() {
        return brand;
    }

    public String getModel() {
        return model;
    }

    public int getYear() {
        return year;
    }

    // Setter methods (Mutators)
    public void setBrand(String brand) {
        this.brand = brand;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public void setYear(int year) {
        this.year = year;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Maruti", "Swift", 2016);

        // Accessing object's state using getters
        System.out.println("Brand: " + myCar.getBrand());
        System.out.println("Model: " + myCar.getModel());
        System.out.println("Year: " + myCar.getYear());

        // Modifying object's state using setters
        myCar.setYear(2022);
        System.out.println("Updated Year: " + myCar.getYear());
    }
}
```

3. Inheritance:
Inheritance is a mechanism where one class (subclass or derived class) inherits the properties (attributes and methods) of another class (superclass or base class). It promotes code reusability and establishes a hierarchical relationship between classes.

Syntax:
```
class Parent {
    void display() {
        System.out.println("Parent class method");
    }
}

class Child extends Parent {
    // Child class inherits display() method from Parent
    void show() {
        System.out.println("Child class method");
    }
}
```

Example:
```
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);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Types of Inheritance:

Single Inheritance: A class inherits from only one superclass.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Multilevel Inheritance: A class inherits from a superclass, and another class inherits from this subclass.

```
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: Multiple classes 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...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Multiple Inheritance (supported through interfaces): A class can implement multiple interfaces.
To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

```
class C extends A, B { //suppose if it were
```

Hybrid: inheritance + interface / extends with implements

```
class Shape {
   // superclass Shape
}

interface Drawable {
   void draw();
}

class Circle extends Shape implements Drawable {
    public void draw() {
        // implementation of draw method
    }
}

class ColoredCircle extends Circle {
    // subclass ColoredCircle inherits from Circle
}
```

- **Polymorphism**
  Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class. It enables a single method name to be used for different types of objects. Polymorphism in Java is achieved through method overriding and method overloading.

Types of Polymorphism in Java:
Compile-time Polymorphism (static binding) (Method Overloading):
Method overloading allows a class to have more than one method having the same name, if their parameter lists are different. It is determined at compile-time based on the number, type, and order of arguments passed to the methods.

Example of method overloading:

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }
}
```

Run-time Polymorphism (Dynamic Binding) (Method Overriding):
Method overriding occurs when a subclass provides a specific implementation of a method that is already provided by its superclass.

```
class Car{
 void run(){System.out.println("running car");}
}
class Swift extends Car{
 void run(){System.out.println("running swift car");}

 public static void main(String args[]){
  Car car = new Swift();//upcasting
  car.run();
 }
}
```

Example of method overriding:

```
public class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();
        myDog.makeSound(); // Output: Dog barks
        myCat.makeSound(); // Output: Cat meows
    }
}
```

- **Instance initializer block:** Instance Initializer block is used to initialize the instance data member. It runs each time an object of the class is created.

```
class College{
    Static String collegeName;

    College(){
          System.out.println("college name is "+collegeName);
          collegeName +=" INDIA";
          }

    //initializer block
    {collegeName="IIT DELHI";}

    public static void main(String args[]){
    College c1=new College();
    College c2=new College();
    }
}
```

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when an instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear

Full Example with super():

```
class A{
A(){
System.out.println("parent class constructor invoked");
}
}

class B3 extends A{
```

```
B3(){
super();
System.out.println("child class constructor invoked");
}

B3(int a){
super();
System.out.println("child class constructor invoked "+a);
}

{System.out.println("instance initializer block is invoked");}

public static void main(String args[]){
B3 b1=new B3();
B3 b2=new B3(10);
}
}
```

- **Final Keyword in Java:**

1. **Variable:** If you make any variable as final, you cannot change the value of final variable(It will be constant).
   final int maxSpeed=90;
2. **Method:** If you make any method final, you cannot override, but the final method can be inherited.

```
class Bike{
  final void run(){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();
   }
}
```

3. **Class:** If you make any class as final, you cannot extend it.
   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();
  }
}
```

Final blank variable initialize:

```
class Bike{
  final int speedlimit;//blank final variable

  Bike(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike();
 }
}
```

Static final blank variable:

```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
 }
}
```

Final as parameter:

```
class Bike{
  int cube(final int n){
   n=n+2;//can't be changed as n is final
   n*n*n;
  }
  public static void main(String args[]){
    Bike b=new Bike();
    b.cube(5);
 }
}
```

Dynamic Binding:

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

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Classes & Objects

- **this Keyword**

  this keyword refers to the current object within an instance method or constructor.
  It is used to differentiate between instance variables and method parameters with the same name.
  Example:

```
class MyClass {
    int value;

    MyClass(int value) {
        this.value = value; // 'this' is used to refer to the instance variable
    }
}
```

- **Call by Value**

  Method overloading allows defining multiple methods with the same name but different parameters within the same class.

Example:

```
class MyClass {
    void method() {
        // Method implementation
    }

    void method(int num) {
        // Method implementation with an integer parameter
    }
}
```

- **Random class**

  The Random class in Java is used to generate pseudo-random numbers.
  It provides methods to generate random integers, doubles, floats, and longs.
  Example:

  import java.util.Random;
  Random random = new Random();
  int randomNumber = random.nextInt(100); // Generates a random integer between 0 and 99

- **Constructors**

  Constructors are special methods used to initialize objects.
  They have the same name as the class and no return type.
  Example:

```
// DEFAULT CONSTRUCTOR
    Main(){
        a = 12;
        name = "kushal";
        System.out.println(name+ " " +a);
    }
    // PARAMETERIZED CONSTRUCTOR
    Main(int a,String name){
        this.a = a;            //THIS KEYWORD IS USE TO REFER INSTANCE VARIABLE
        this.name = name;


        System.out.println(name+ " " +a);
    }
```

- **Parameterized Constructors**

```
Main(int a){

    this.a = a;          //THIS KEYWORD IS USE TO REFER INSTANCE
VARIABLE



    System.out.println(a);

 }
```

- **Overloading Constructors**

```
// OVERLOADING CONSTRUCTOR

 Main(int a,String name){

    this.a = a;          //THIS KEYWORD IS USE TO REFER INSTANCE
VARIABLE

    this.name = name;

    System.out.println(name+ " " +a);

 }
```

- **Instance Variable Hiding**

Instance variable hiding occurs when a local variable or method parameter has the same name as an instance variable.
Example:

```
class MyClass {
   int value;

   void setValue(int value) {
      this.value = value; // 'this' is used to refer to the instance variable
   }
}
```

- **Initialization block**

Initialization blocks are used to initialize instance variables of a class.

They are executed when an object of the class is created, before the constructor.
Example:

```
class Keshav{
   Keshav(){
      int y=23;
      System.out.println(y);
   }
    int x;
    {                              /INITIALIZATION/INSTANCE BLOCK They are executed when
an object of the class is created, before the constructor./*
       x =10;
          System.out.println(x);
    }
   }
public class Intializationblock {
   public static void main(String[] args) {
      Keshav m = new Keshav();
   }
}
```

Aggregation: aggregation is a relationship between two classes where one class contains a reference to another class but each class has its own lifecycle. This means that the contained class can exist independently of the containing class. Aggregation is often described as a "has-a" relationship.

Address.java

```
public class Address {

String city,state,country;



public Address(String city, String state, String country) {

   this.city = city;

   this.state = state;

   this.country = country;
}


}
```

Emp.java

```
public class Emp {

int id;

String name;

Address address;


public Emp(int id, String name,Address address) {

   this.id = id;

   this.name = name;

   this.address=address;

}


void display(){

System.out.println(id+" "+name);

System.out.println(address.city+" "+address.state+" "+address.country);

}
public static void main(String[] args) {

Address address1=new Address("gzb","UP","india");

Address address2=new Address("gno","UP","india");



Emp e=new Emp(111,"varun",address1);

Emp e2=new Emp(112,"arun",address2);



e.display();

e2.display();



}

}
```

Inner Classes

- **Simple Inner Class / Non - Static Nested class:**

A non-static nested class, also known as an inner class, is associated with an instance of its enclosing class. It can access all members (including private) of the outer class and can be instantiated only with an instance of the outer class.

```
class Outer {

   private int outerField = 20;
```

```
    class Inner {

        void display() {

            System.out.println("OuterField: " + outerField);

        }

    }

}
```

Outer outer = new Outer();

Outer.Inner inner = outer.new Inner();

inner.display();

- **Static Inner Class:** A static nested class is a nested class that is declared static. It cannot access non-static members of the outer class directly. This type of nested class is useful for grouping classes together that are logically related but do not require access to instance variables of the outer class

```
class Outer {

    private static int staticOuterField = 25;


    static class NestedStatic {

        void display() {

            System.out.println("Static OuterField: " + staticOuterField);

        }

    }

}


Outer.NestedStatic nestedStatic = new Outer.NestedStatic();
```

```
nestedStatic.display();
```

- **Local Method Inner Class:** A local class is a class defined within a method or a scope block. Local classes have access to the variables of the enclosing block, but these variables must be effectively final (i.e., their values should not change after initialization).

```
class Outer {
    void outerMethod() {
        final int localVar = 10;


        class Local {
            void display() {
                System.out.println("LocalVar: " + localVar);
            }
        }


        Local local = new Local();
        local.display();
    }
}


Outer outer = new Outer();
outer.outerMethod();
//////
class Top{
    int x = 10;
void display(){
    class In{
        void print(){
        System.out.println("HELLO METHOD LOCAL CLASS "+x);
        }
    }
    In inner = new In();
    inner.print();
}
}


public class Methodlocalinnerclass {
    public static void main(String[] args) {
        Top outer = new Top();
        outer.display();
    }
}
```

- **Anonymous Class:** An anonymous class is a class without a name that is defined and instantiated in a single step. Anonymous classes are useful when you need to override methods of a class or interface without explicitly subclassing.

```
interface Greeting {
    void greet();
}
```

```
public class Main {
    public static void main(String[] args) {
        Greeting greeting = new Greeting() {
            @Override
            public void greet() {
                System.out.println("Hello!");
            }
        };

        greeting.greet();
    }
}
```

Singleton Class:

Singleton class is a class that can have only one object (an instance of the class) at a time. After the first time, if we try to instantiate the Java Singleton classes, the new variable also points to the first created instance.

A singleton class is a class that restricts the instantiation of itself to only one object. This pattern is useful when you want to ensure that only one instance of a class is created and provide a global point of access to that instance.

The primary purpose of a java Singleton class is to restrict the limit of the number of object creations to only one. This often ensures that there is usage control to resources, for example, database and socket connection.

Remember the key points while defining a class as a singleton class:

1. Make a constructor private.

2. Write a static method that has the return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

   ```
   public class Singleton {

       private static final Singleton instance = new Singleton();

       private Singleton() {}

       public static Singleton getInstance() {

           return instance;

       }

   }
   ```

Example:

```
public class Singleton {

    // Private static variable to hold the single instance of the class

    private static Singleton instance;


    // Private constructor to prevent instantiation from outside

    private Singleton() {

        // Initialization code, if any

    }


    // Public static method to get the single instance of the class

    public static Singleton getInstance() {

        // Check if the instance has not been initialized yet
```

```
        if (instance == null) {

            // Create a new instance if it's not available

            instance = new Singleton();

        }

        // Return the single instance

        return instance;

    }

    // Other methods of the class can go here

}


public class Main {

    public static void main(String[] args) {

        // Get the singleton instance

        Singleton singleton = Singleton.getInstance();


        // Do something with the singleton instance

        // For example, print its hash code

        System.out.println("Singleton instance: " + singleton.hashCode());


        // Try to get another instance

        Singleton anotherSingleton = Singleton.getInstance();


        // The hash code of both instances should be the same
        System.out.println("Another singleton instance: " +
anotherSingleton.hashCode());

    }

}
```

/////////////

Math Class: Java Math class provides several methods to work on math calculations like min(), max(), avg(), sin(), cos(), tan(), round(), ceil(), floor(), abs() etc.

```
public class JavaMathExample1

{

    public static void main(String[] args)

    {

        double x = 28;

        double y = 4;


        // return the maximum of two numbers

        System.out.println("Maximum number of x and y is: " +Math.max(x, y));


        // return the square root of y

        System.out.println("Square root of y is: " + Math.sqrt(y));


        //returns 28 power of 4 i.e. 28*28*28*28

        System.out.println("Power of x and y is: " + Math.pow(x, y));
        // return the logarithm of given value

        System.out.println("Logarithm of x is: " + Math.log(x));

        System.out.println("Logarithm of y is: " + Math.log(y));


        // return the logarithm of given value when base is 10

        System.out.println("log10 of x is: " + Math.log10(x));

        System.out.println("log10 of y is: " + Math.log10(y));

    }

}
```