OOPS (Object-Oriented Programming System)

Java Classes

A class in Java is a set of objects that shares common characteristics/ behavior and properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, a Student is a class while a particular student named Ravi is an object.

Java Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class. A typical Java program creates many objects, which as you know, interact by invoking methods

There are four major concepts in OOPs

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

Access Modifiers

1. Public
2. Private
3. Protected
4. Default

Public - Can be accessed by anyone.
Default - Can be accessed by anyone in the same package.
Protected - Can be accessed by anyone in the same package and if anyone in another package wants to use it then only subClass can access it.
Private - Can only be accessed in the same class. To modify private data we use geter and seter functions.

Constructor

A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes

Constructors are used to create objects. There are 3 types of constructors in Java

Properties

1. The name of the Class and its constructor should be the same
2. constructors will not return any data
3. A constructor can be called only once for an object

Types

1. Non-parameterized constructor - we don't pass any values into it
2. Parameterized constructor - we do pass the parameters we need at initialization
3. Copy constructor - we pass in another instance of class i.e. object into this object

```
class Student{

  String name;
  int age;
  Student(){                        // Non-parameterized constructor
     System.out.println("Non-parametarized constructor has been called");
  }
  Student(String name, int age){          // Parameterized constructor
     this.name = name;
     this.age = age;
     System.out.println("Parametarized constrctor has been called");
  }
  Student(int age){                    // Parameterized constructor
     this.age = age;
     System.out.println("Parametarized constrctor has been called");
  }
  Student(StudentNew s2){               // Copy constructor
     this.name = s2.name;
     this.age = s2.age;
     System.out.println("Copy constructor has been called");
  }
}
```

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

Types

1. Compile-time polymorphism (Function overloading)
2. Runtime polymorphism (Function overriding)

Method Overloading : Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

1. The type of the parameters passed to the function.
2. The number of parameters passed to the function.

Compile-time polymorphism (Function overloading)

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading.

```
class Vehicle {
  String name;
  int number;
  public void CompileTimePoly(String name){
     System.out.println(name);
  }
  public void CompileTimePoly(int number){
     System.out.println(number);
  }
  public void CompileTimePoly(String name, int number){
     System.out.println(name + " - " + number);
  }
}
```

Runtime Polymorphism : Runtime polymorphism is also known as dynamic polymorphism. Function overriding is an example of runtime polymorphism. Function overriding means when the child class contains the method which is already present in the parent class. Hence, the child class overrides the method of the parent class. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.

```
class Shape {
  public void area() {System.out.println("Displays Area of Shape");}
}
class Triangle extends Shape {
  public void area(int h, int b) { System.out.println((1/2)*b*h); }
}
class Circle extends Shape {
  public void area(int r) { System.out.println((3.14)*r*r); }
}
```

Encapsulation

Encapsulation means bundling data(variables) and methods(functions) together in one entity. The entity is the class. The class combines the variables and functions.
It is used for data hiding.
It is done using the help of access modifiers.

```
// Java Program to demonstrate
// Java Encapsulation

// Person Class
class Person {
 // Encapsulating the name and age
 // only approachable and used using
 // methods defined
 private String name;
 private int age;

 public String getName() { return name; }

 public void setName(String name) { this.name = name; }

 public int getAge() { return age; }

 public void setAge(int age) { this.age = age; }
}

// Driver Class
public class Main {
 // main function
 public static void main(String[] args)
 {
   // person object created
   Person person = new Person();
   person.setName("John");
   person.setAge(30);

   // Using methods to get the values from the
   // variables
   System.out.println("Name: " + person.getName());
   System.out.println("Age: " + person.getAge());
 }
}
```

Abstraction in Java

Abstraction in Java is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

Abstraction is achieved in two ways

- Abstract class
- Interface (pure abstraction)

Abstract Class

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods that will force the subclass not to change the body of the method.

```
abstract class Animal {
  abstract void walk();
  void breathe() {
      System.out.println("This animal breathes air");
  }
  Animal() {
      System.out.println("You are about to create an Animal.");
  }
}
class Horse extends Animal {
  Horse() {
      System.out.println("Wow, you have created a Horse!");
  }
  void walk() {
      System.out.println("Horse walks on 4 legs");
  }
}
class Chicken extends Animal {
  Chicken() {
      System.out.println("Wow, you have created a Chicken!");
  }
  void walk() {
      System.out.println("Chicken walks on 2 legs");
  }
}
public class OOPS {
  public static void main(String args[]) {
     Horse horse = new Horse();
     horse.walk();
     horse.breathe();
  }
}
```

Interface

- All the fields in interfaces are public, static, and final by default.
- All methods are public & abstract by default.
- A class that implements an interface must implement all the methods declared in the interface.
- Interfaces support the functionality of multiple inheritance.

```
interface Animal {
  void walk();
}
class Horse implements Animal {
  public void walk() {
      System.out.println("Horse walks on 4 legs");
  }
}
class Chicken implements Animal {
  public void walk() {
      System.out.println("Chicken walks on 2 legs");
  }
}
public class OOPS {
  public static void main(String args[]) {
     Horse horse = new Horse();
     horse.walk();
  }
}
```

static Keyword in Java

The static keyword in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The users can apply static keywords with variables, methods, blocks, and nested classes. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.

The static keyword is a non-access modifier in Java that is applicable for the following:

- Blocks
- Variables
- Methods
- Classes

Inheritance

Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Types of Inheritance :

- Single inheritance : When one class inherits another class, it is known as single level inheritance
- Hierarchical inheritance : Hierarchical inheritance is defined as the process of deriving more than one class from a base class.
- Multilevel inheritance : Multilevel inheritance is a process of deriving a class from another derived class.
- Hybrid inheritance : Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.
- Multiple Inheritance (Through Interfaces)

```
class Shape{
   public void area(){
       System.out.println("display area");
   }
}

class Circle extends Shape{
   public void area(int r){                              // example of single level inheritance
       System.out.println( (3.14) * r * r );
   }
}

class Traingle extends Shape{
   public void area(int l, int h){                      // example of single level inheritance
       System.out.println( 0.5 * l * h );
   }
}

// Circle and Traingle combined make a example of Hierarchial Inheritance
// because there are multiple derived class from a base class

class EquilateralTraingle extends Traingle{
   public void area(int side){
 // example of multi level in heritance
// EquilateralTraingle extends Traingle and Traingle extends Shape
       System.out.println( (1.732/4) * side * side );        // ( 1.732 is root 3)
   }
}
```

```
// all this together is a example of Hybrid inheritance
// mainly its when we have hierarchial & multi level inheritance together
// in the above example Shape class is been inherited by hierarchial inheritance by Traingle &
Circle class
// and been inherited by multi level inheritance by EquilateralTraingle class as it extends
traingle and traingle extends Shape

// There is also a 5th type of inheritance called Multiple Inheritance it is not used in Java
// to apply its comcepts we use interface
```

Multiple Inheritance (Through Interfaces) : In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.

Package

There are In Build Package & User Defined Package.

Package in Java is a mechanism to encapsulate a group of classes, sub-packages, and interfaces. Packages are used for:

1. Preventing naming conflicts. For example, there can be two classes with the name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
2. Making searching/locating and usage of classes, interfaces, enumerations, and annotations easier
3. Providing controlled access: protected and default have package-level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
4. Packages can be considered as data encapsulation (or data-hiding).

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new java file to define a public class, otherwise we can add the new class to an existing .java file and recompile it. Subpackages: Packages that are inside another package are the subpackages. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers. Example :

```
import java.util.*;
```

util is a subpackage created inside java package.

Accessing classes inside a package

```
// import the Vector class from util package.
import java.util.vector;

// import all the classes from util package
import java.util.*;
```

First Statement is used to import Vector class from util package which is contained inside java.

Second statement imports all the classes from util package.

```
// All the classes and interfaces of this package
// will be accessible but not subpackages.
import package.*;

// Only mentioned class of this package will be accessible.
import package.classname;

// Class name is generally used when two packages have the same
// class name. For example in below code both packages have
// date class so using a fully qualified name to avoid conflict
import java.util.Date;
import my.package.Date;
```