**Java**

**Introduction 0**

- Java was developed by James Goslin in 1995 at Sun-Microsystems

- It’s an Object Oriented Programming Langauge

- It is a compiled and interpreted language

- It is a platform independent programming language that follows the logic of **“Write once , Run anywhere”**

**Features**

- **Easy** to learn and its syntax is quite simple

- Java can be used to develop virus-free systems .

- Java is **secure** because , its programs run inside virtual machine sandbox to prevent any activity from untrusted sources

- No use of explicit **pointers** , hence address access is restricted

- It is **Robust** :

- Java checks the code during the compile-time and run-time

- Java takes care of memory allocation and releasing(erasing) by its own or Java does Garbage- collection by itself only.

- **Garbage collection** means freeing up of the redundant memory loactions

- **Portable** : Application written on one platform of Java can be easily ported to another platform as it is platform independent

- **Dynamic** : Many objects are evaluated at run time and execution is carried out . ex: Run-time Polymorphism

- **Distributed**:

-It allows a Java program running in one machine can interact or call something from the program running on another machine in the network .

- RMI(Remote Method Invocation),EJB(Enterprise Java Beans) etc. are used for creating distributed applications using Java

- **Multi-threaded** :

- Thread is a task in a process/program

- Multi-threading means multiple tasks running/executing at the same time

- This facility is provided by Java so that multiple tasks can be executed at the same time

- **Object Oriented** :

- OOP Is an approach to problem-solving where all computations are carried out using objects

- Objects are the basic units of OOP

**Application domains**

- Android Apps

- Web-based Apps(Server-side programming)

- Desktop GUI Apps

- Distributed Apps

- IoT Apps

// TO COVER UP AT LAST MOMENT

\*\*\*JAVA vs CPP vs C\*\*\*

\*\*\*why use\*\*\*

\*\*\*Advantages and Disadvantages\*\*\*

**Introduction 1**

- A function is a block of code that perfomes a task

- Every java program must have atleast one function and that function is **main()**

- Class is a container for related functions

- Every java program must have atleast one class , which contains main() function , i.e

public class **M**ain(){

public static void main(String[] args){

}

}

- In java we use different coventions for naming classes and methods . Like ,**P**ascal**N**aming**C**onvention for classes and **c**amel**N**aming**C**onvention/**camelCaseNotation** for naming the methods

- A method is a function that belongs to a class

- In a java prpgram , each classes and methods should have access modifiers .

- Access modifier is a special keyword that determines if other classes and methods are accessible in a particular class or program . Ex:- public , private etc

- We use a package to group related classes , interfaces and enums into a single module :

- By convention the base package for a java project is the domain name of your company in reverse . ex:- **package com.company;**

- **System.out.print("Hello World")**; 🡨In this statement , **System** is a **class** belongs to **java.lang** package , **.** is an operator , **out** is a **field** and **println()** is **method** of the class .

Autopsy of **public static void main(String[] args)**

**public(Access Modifier)** - It specifies that main() method can be accessed globally or from outside of the current class.

**static(Keyword)** - It means we can access the main() method without instaciating it or without creating the object of Main() class in which exists . Hence JVM directly use this method without making any object , as the entry point of the program .

**String[]** args -It’s an Array of strings and it acts as a signature for main() function , as main() is the entry point of a Java program . As it is an array , it can store command line string-arguments passed while executing the program .

**JDK , JRE and JVM**

**JDK**

- JDK is a cross-platformed software development environment that offers a collection of tools and libraries

necessary for Java-based software applications and applets . JDK = JRE + Dev-tools

- It includes JRE , an interpreter/loader(**java)** , a compiler**(javac**) , an archiver(jar) , documentation generator(Javadoc) and other tools needed in Java development

- JDK is what we need to compile the Java source-code and It is the core package used in Java which contains JRE(Java Runtime Environment) and JVM(Java Virtual Machine)

- Applet is Java program that can be embedded into a web-page . It runs inside the web-browser and works at client-side

**JRE**

- JRE(Java Runtime Environment) is what we need to run a java program and contains a set of libraries and other files that JVM uses at the run time . JRE = JVM + Library Classes

- It is an installation package that provides an environment to only run(not develop) the java program(or application) onto your machine .

**JVM**

- JVM(Java Virtual Machine) - It is an abstract machine offers the runtime environment for codes to be executed .Basically , whatever java program we run using JDK or JRE goes into JVM and JVM is responsible for executing java prpgram line by line . hence it is also known as **interpreter**

\* full description on GFG : https://www.geeksforgeeks.org/differences-jdk-jre-jvm/

|  |  |  |
| --- | --- | --- |
| JDK | JRE | JVM |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**How Java codes executed under the hood**

- **Source-code is first compiled into the Bytecode and then it is interpreted to Machine code**

- Basically there are two steps involved here : Compilation and Execution

- In Compilation step , IDE uses java compiler to compile the souce-code(Main.java) into a different format called Java Byte-Code(Main.class)

- Main.class is the Bytecode-representation of the java file and Bytecode is a machine independent-encoding.

-In our java-project , **src** directory holds our source-code and , **out** directory holds the results of the compilation . Hence , Java Bytecode(**Main.class)** is also stored in **out** folder

- The Java Byte-code(Main.class) is platform-independent and That means , it can run over any OS that has JRE

- In Execution step , JRE has JVM that takes the Java Bytecode and translate it into the native code for the underlying OS . Due to this architecture is the reason why Java applications are portable or platform independent .

**Fore more info : https://www.geeksforgeeks.org/compilation-execution-java-program/**

**Data-Types**

**Primitive-types :** for storing simple values . In it , we don’t need to allocate and release memory for the variable as it is done by JRE .

int a = 5 ; 🡨 Here , 5 is stored in the location of variable **a** .

int b = a ; 🡨Here , actual value of a is assigned to **b** and stored at its location in memory.

**a = 7 ;**

System.out.println(b);

**output: 5** (b not updated alongwith a)

- Above a and b are having different memory locations . Hence , a and b are completely independent

of each other . Or , if we change the value of a to 7 , the value at b is not gonna change at all .

|  |  |  |
| --- | --- | --- |
| Type | Bytes | Range |
| **byte** | 1 | [-128,127] |
| **short** | 2 | [-32k,32k] |
| **int** | 4 | [-2B,2B] |
| **long** | 8 |  |
| **float** | 4 |  |
| **double** | 8 |  |
| **char** | 2 | A,B,c,d |
| **boolean** | 1 | true/false |

int viewCount = 123456789 or 123\_456\_789(**<2B**)

**long** viewCount = 3\_123\_456\_789 (**>2B**) , Even though we have specified long data-type , it will still act like an integer by default and hence will give error. Hence , we need to add ‘**L/l**’ suffix after the value , to make it work

**long** viewCount = 3\_123\_456\_789**L**

**float** price = 10.99 , Here , Java sees this number-value as a **double** by default and hence will throw error . Hence ,here we need to add suffix **‘F/f’**

**float** price =10.99**F**

**Reference-types:** for storing complex objects . We need to allocate memory for this variable but release is handled by JRE .

Ex:- Date **now** = **new** Date(); Here , **now** is an instance/object of Date class and it has the access to all the members of Date() class . like , **now.getTime();**

Point point1 = new Point(1,1);

Point point2 = point1;

**point1.x = 2;**

System.out.println(point2);

**Output : java.awt.Point[x=2,y=1]** (point2.x updated alongwith point1.x)

- JRE will allocate some memory for **Point(1,1)** object and point1 label is attached to a separate memory location and in that memory location the address of the Point(1,1) object will be stored instead storing the Point(1,1) object .

- When we declare a primitive variable , the value that we assigned in that variable will be stored in that memory location of the variable but , in Reference-type the variable is going to store/hold the address of that Point(1 , 1) object in memory not the actual Point() object .

- Hence , point1 and point2 are storing the address of Point() object or these are the references to the Point() object in the memory .

- Hence , point1 and point2 are not independent of each-other as they are referencing the same object in the memory .

- Changes made to point1 will be reflected in point2 , as can be seen in the output

**- Hence Reference-Types are copied by the references and Primitive-types are copied by values**

**Strings**

- In Java Strings are Reference type-always and we use **String class** to declare a string object .

like , **String new = var String(“Hello”);**

- **String var = “Hello” ;** is just a shorthand to write **String new = var String(“Hello”);**

**-** Since String is a class then the object which we declare of it can access its members(methods, classes) too . like ,

var.endsWith(“!!”) - gives a Boolean

var.startsWith(“!!”) - Boolean

var.length() - returns length od string .

var.indexOf(‘e’) - returns the index of first occurrence of character/string in the string otherwise it returns -1

var.replace(target , replacement) - replaces the target part with the replacement part and will return a new string as it doesn’t modify the actual string

var.toLowerCase() - It turns the string to Lower Case and return a new string

var.trim() - It removes the extra spaces from the beginning and the end of the string and returns a new string

- In java strings are immutable or unchangeable

**Escape Sequences**

- A character preceded by a backslash(\) is an escape sequence and has special meaning to the compiler

|  |  |
| --- | --- |
| Escape Sequences | Description |
| \n | insert a new line in the text at this point |
| \t | insert a tab in the text at this point |
| \’ | single-quote |
| \” | double-quote |
| \\ | backslash |

ex:- sout(“ **\”**Hello**\”** ”) output : “Hello” sout(“\\Hello\\”) output: \Hello\

**Arrays**

- Used to store a list of items like , list fo numbers , people etc .

- Arrays are also Reference-types in Java

Declaration : **int[] number = new int[size];** OR **int[] number = {2,3,8,1,6};**

- To print the array we go like , **sout(Arrays.toString(number)) ;** Output : **[2,3,8,1,6]**

**-** sout**(number.length)** prints the length of array i.e , 5 ; length is a field here

- sout(**Arrays.sort(number)**) wil print sorted array

**2-D Arrays**

**- int[][] number= new[2][3]** OR **int[][] = {{3,5,8},{4,7,3}};**

**-**To print the MD-array we go like , **sout(Arrays.deepToString(number)) ;** Output : **[[3,5,8],[4,7,3]]**

- To make a variable constant or unchangeable we use final keyword , like , **final float PI = 3.14F**

**Casting and Type-conversion**

**Implicit-casting/Automatic-casting/Auto-conversion :** In this conversion a shorter-type is converted/casted to higher-type automatically like , **short to int , int to long** etc .

**byte>short > int>long>float>double**

ex:- **short a = 5;** **int b = a + 6 ;** In this , variable ‘a’ would be converted to int first and then the operation will get executed

ex:- **double a = 3.1 ; double b = a + 6 ;** Since 6 is an integer and int is less precise than double , 6 will be typecasted first to double and then the sum will executed

**Explicit casting : We explicitly cast the variables form higher-type to shorter-type .**

**ex:- double x = 1.1 ; int y = (int)x + 2 ;** here , x is higher-type hence , to caste it to int we do so..

**- Intger.parseInt(“string”) - It takes a string and returns an integer . Here , Integer is wrapper-class for int-types**

- In Java wrapper classes provides the mechanism to convert primitive types into object and object into primitives .

like , Integer is a wrapper class for int-types , Short for short-types , Float for floats and Double for double-types.

**Math-class** and its **Methods**

**-** This class is defined in **java.lang** package

**int val = Math.round(1.1F) , val=1** : takes float/double and returns whole(int)/long

**int val = (int)Math.ceil(1.1) , val = 2** : it returns smallest integer greater than or equal to the number . It returns double , hence to store it returned value in int variable we first need to cast the returned value to integer otherwise it gives a compilation error

**int val = (int)Math.floor(1.1F) , val = 1** : It returns the largest integer that is smaller or equal to this number .

**Math.max/min(int , int) :** it returns max/min of the given integers

**double val = Math.random()**

- It generates a random double-value b/w 0 and 1

- to get a double value b/w 0 and 100 we simply do **, Math.random()\*100**

- to get a number without decimal values we go like , **double val = Math.round(Math.random()\*100);**

- to get random integer values b/w 0 and 100 we do , **int val = (int)Math.round(Math.random()\*100); ,** since Math.round returns a long value , hence we can’t store it in an int variable , hence we need casting the returned value to int.

- we can also go like , **int val = (int)(Math.random()\*100);** , to meet the above outcome

**Formatting Numbers(Format class)**

**- In formatting any numbers we make the simple numbers in(1235.54) different formats like , currency ($1,235.54), percentage(1235.54%)**

**NumberFormat class**

- It is the direct child class of format-class

- defined in **java.text** package and in this package we have lots of classes to handling numbers , dates ,texts etc.

- It is an abstract class , means it can’t be instanciated

- Since we can’t use **‘new’** operator to make an instance of this class , hence we use the bunch of methods defined in it and when we call these methods , it will create an instance of NumberFormat class and will return it . ex:- getCurrencyInstance() , getPercentageInstance() etc

- Hence , these methods are called Factory Methods , because it creates instances and return them

- These factory methods will return a NumberFormat object and can be stored in a variable of type NumberFormat , like

**NumberFormat currency = NumberFormat. getCurrencyInstance();**

- The returned object currency has a method **format()** , to format the values , and it will return a string value

**String result = currency.format(123456.54);**

sout(result); output : **$123,456.54**

also , **NumberFormat percent = NumberFormat. getPercentInstance();**

**String result = percent.format(0.5);**

**System.out.println(result);**

**ouput : 50%**

we can also use **method-chaining** to achieve the same result

**String result = NumberFormat. getCurrencyInstance().format(0.5);**

**Reading Input**

**Scanner scanner = new Scanner(System.in); ,** System.in specifies that we are gonna read the data from terminal window

- Scanner class is defined in **java.util** package

- here ‘**in’** is a field defined in System class

- scanner have multiple methods to read inputs like , integer , strings , floats , doubles , etc as , we use ,

int n = scanner.**nextInt()** for integer

String s = **scanner.next()** and.**nextLine()** for strings

- Two space-separated strings are called two distinct tokens .

|  |  |
| --- | --- |
| next() | nextLine() |
| It reads input from the input device till the space character | I reads the input till the line change |
| It ends reading the input after getting space | It ends reading the input after getting ‘\n’ or press enter |
| next() will read only one token at a time | nextLine() read multiple tokens at a time or it reads the entire line  we type no matter how many number of spaces are there |
| we can’t use trim() method with it to remove the  spaces in the string | We can use trim() method to remove the extra spaces present in the  beginning and at the end of the string typed . like,  String s = scanner.**nextLine().trim()**  **input: spaces**  **sout(s);**  **output:spaces** |

- Expression is a piece of code that produces values , like **a==b** is a Boolean expression and it produces true or false

Comparison Operators : **==** , **!=** , **<=** and **>=**

Logical Operators : &&(Logical AND) , ||(OR) , !(NOT)

Ternary Operator : condition **?** output **:** another output

String input = “” ; 🡨It is a string of a reference-type , Hence we can’t use comparison operators with it, OR , we can’t write like , **while(input!=”quit”)** , b/c input is holding an address , and we can’t use comparison operators b/w reference types b/c it will compare the addresses of input and “quit” and not their values , instead we can go like , **while(!input.equals(“quit”))**

**- String str = “SCHOOL” ,** we can use **.toLowerCase** method of string class-objects to convert it to lower case like , str.toLowerCase

**for-each loop**

- It is mainly used to traverse the array or collection elements

syntax : **for(array-datatype loop-variable : array/collection name){**

**sout(loop-var);**

**}**

ex:- String[] fruits = {"A","B","C"};

for(String **fruit**:fruits)

System.out.println(**fruit**);

- In the loop body , we use the loop variable we created rather than using an indexed array element

**Drawbacks:-**

- It always iterates forward only and so we can’t iterate over the array from the end to beginning

- Or , it can’t traverse the array in reverse order

- It doesn’t work on index-basis , hence , it doesn’t keep track of indexes

- Moreover , you can’t traverse the odd or even elements only

- You can’t modify the array

**Advantages:-**

- It makes the code more readable

- It eliminates the possibility of programming errors

**\*\*head to gfg for more ...........**

**OOPS**

- Classes and objects were introduced in programming because we can solve the real world problems thorugh programing

- A class is a group of objects which have common properties . It is a template or blueprint from which abjects are created

- A class is a user-defined blueprint or prototype from which objects are created .

- Class represents the set of properties/fileds/attributes/states or methods that are common to all objects of one type

\*\*\*application of OOPS/Purpose\*\*\*

- Fields are the variables declared inside a class .

**Exceptions and Errors**

- Exceptions are Java’s way to express errors

- methods(functions) and properties(fields) are the members of a class .

for exmpale : for a class car,

properties can be : name , prize , color , mileage etc

methods can be : drift , drive , stop etc

|  |  |
| --- | --- |
| classes | objects |
| A class is a template for creating objects | objects are the instances of classes |
| class is a logical entity | object is a physical entity |
| class doesn’t allocate memory space | objects allocate memory |
| class can be declared only once | multiple objects can be created |
| Ex: car | ex: Jaguar , bolero , benz |
| class generates objects | objects provide life to the classes |
| Classes can’t be manipulated as they are not  available in memory | can be manipulated |
| it doesn’t have any values which are associated  with the fields | Each and every object has its own values , which  are associated with the fields |
| classes can be created using ‘class’ keyword | can be created using “new ” keywords |

- creating an **object(instance)** of Bro class

**Bro obj = new Bro();**

- Here , the **obj** is the reference variable of its object

- **new** keyword allocates the memory for object in heap

- **Bro()** is a special function which is called **Constructor**

- **this** is also a reference variable that referx to current object in a method or constructor

**Constructor**

package com.company;

class Student{

String name ;

int age;

**Student(){**

**System.out.println("constuctor created");// it will be executed right after an object is created**

**};**

}

public class Main {

public static void main(String[] args) {

Student s1 = new **Student();**

}

}

- A constructor in java is a special method that is used to initialize objects or to construct objects

- It is called when an object of a class is created using **new** keyword

**Properties of Constructors**

- Its name is same as the class name

- It doesn’t return anything hence return-type is not used with it , like , void , int etc

- It is called only once , i.e at the time we create an object

**Types**

1.Non-Parameterized : parameters are not passed

**Student()**

**{**

**System.out.println("constuctor created");**

**};**

Student s1 = new **Student();**

2 . Parameterized : Paramteres are passed in the constructor

**Student(String name , int age)**

**{**

**this.name = name;**

**this.age = age;**

**};**

Student s1 = new **Student(“Rahul” , 22);**

**-** here, **this.name** refers the name field/property of the current object which is assigned with passed name a rgument and It works same for this.age

3 . Copy : It is a constructor that creates an object using the fields and methods of another object of the same Java class

- This concept comes from C++ , where it is defined by default as well as by user

- But , in Java copy constructor is defined only by user instead bydefault

class Student{

String name ;

int age;

public void printInfo()

{

System.out.println(this.name);

System.out.println(this.age);

}

**Student(Student s)**

**{**

**this.name = s.name;**

**this.age = s.age;**

**}**

**Student()**{ }//This empty constructor must be defined for s1 object , if we use copy constructor for s2

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student();

s1.name = "rahul";

s1.age = 21;

**Student s2 = new Student(s1);**

**s2.printInfo();**

}

}

- In this code we have assigned the properties of s1 object to s2 , by passing s1 to the constructor of s2

**If we don’t create any constructor in any class then Java automatically creates its own non-parameterized constructor by default and runs the code**

- As constructors creates objects , likewise **destructors** destroy the unused variables or objects .

- In java we don’t need to explicitly define destructors , as it is automatically present there to perform garbage collection unlike in c++. OR we can say , Java has special Garbage-collector for garbage-collection

- **Garbage collection** is the mechanism in which the redundant or unused objects and variables are destroyed to make the unused allocated memory free and this mechanism is automatic in Java

\*\*\*HEAD TO GFG FOR MORE CLARITY ON GARBAGE COLLECTION\*\*\*

**4 major concept under OOPS**

**1 . Polymorphism(many(poly) + forms(morphism))**

- The word polymorphism means having many forms

- Same entity(methods/operators/object) can perform different operations by acquiring different forms in different scenarios or in different subclasses

- It occurs when we have many classes that are related to each other by inheritance

class Polygon {

// method to render a shape

**public void render() {**

**System.out.println("Rendering Polygon...");**

}

}

class Square extends Polygon {

// renders Square

**public void render() {**

**System.out.println("Rendering Square...");**

}

}

class Circle extends Polygon {

// renders circle

**public void render() {**

**System.out.println("Rendering Circle...");**

}

}

- In above code **render()** method is showing different messages in different subclasses

**Types:-**

**a . Compile-time**

- also known as static polymorphism

- This type of polymorphism is achieved by function overloading or operator overloading , but java doesn’t support the operator overloading

- Method Overloading: when there are multiple functions with same name but different parameters .

ex:- void fun(int a) , void fun(int a, int b) , void fun(char a)

- Functions can be overloaded by changing the number of arguments or changing the type arguments

- In this concept we create different methods with same names or we can say that same methods having different functions and application

- it is said as compile time polymorphism because it detects the errors in compile-time only , by validating that whether the supposed overloading rules are followed by the functions or not

- There must be a differentiating factors between the overloaded functions for implementing overloading

**Overloading rules for making differentiating factors . Atleast Any one of the following should be there**

1 . return types of functions can be different , ex: **void** function() . **int** function() **String** function()

2 . If return types are same then the parameters types can be different . ex:- void fun(**int a**) void fun(**char a**)

3 . if return types are same and as well as parameters type are also same then number of arguments must be different. ex:- void fun(int a) void fun(int a, **int b**)

**b . Runtime**

Overriding : It means defining or re-writing/implementing a method in a child class that is already defined in parent class with some method-body modification without changing the name , parameters or return type .

- If subclass/childclass has the same method as declared in the parent class , it is known as method overriding

public Class{

public void method()

{

sout(**“Hello man”**);

}

}

public Child extends Class{

sout(“**Hello buddy**”);

}

|  |  |
| --- | --- |
| Overloading | Overriding |
| occurs within one class | occurs in two classes : super and subclass i.e  inheritance is involved |
| Name of method is same but parameters may or  may not be different | Name and Parameters both are same |
| Purpose : Increase readability of program | Use the method in the child class which is already  present in Parent class |
| Return type can be same or different | Return type is always same |
| It is an example of Compile time polymorphism | example of Runtime Polymorphism |

**2 . Inheritence**

- It is the mechanism in java by which one class is allowed to inherit the features(fields,methods,objects) of another class or interface

- Allows code reusability , as if we have some part of code written in a class and we want to include the same part of code in other classes also then we simply inherit that main class to every sub-classes and can reuse that part of code , rather writing the same code again and again in every sub-classes

**Terminologies:-**

- Super class/base class/parent class : The class whose features are inherited

- Sub class/derived class/extended cass/child class : class that inherits the other class

- the sub classes can add its own fields and methods in addition to the inherited properties and methods

class Child extiends Parent

{

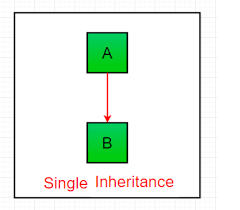
additional fileds;

additional methods();

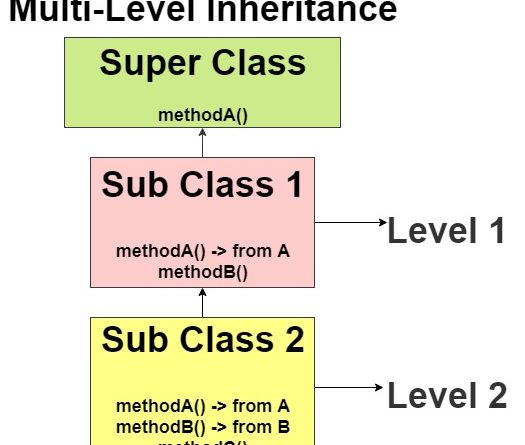
}

**Types:-**

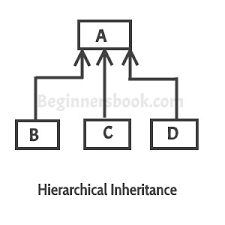
**1 . Single level Inheritence** : A sub-class inherits a single parent class

****

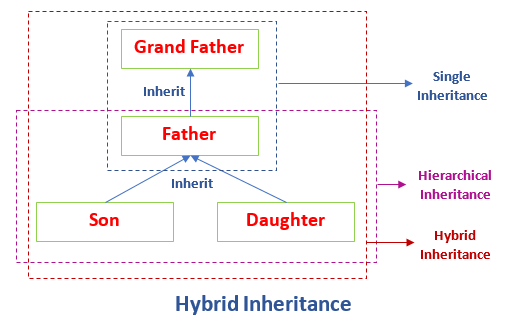
**2 . Multilevel Inheritence** : When a class extends a class which extends another class , it becomes a chain of inheritance and called multilevel inheritance



**3 . Hierarchical Inheritence** : In this type there is only single parent class having multiple derived classes



**4 . Hybrid Inheritence** : It is the combination of two or more types of inheritance , like , **single + hierarchical , hierarchical+multilevel etc .**



**3 . Encapsulation**

**Packages** : These are the groups of similar type of classes , interfaces and sub-packages

built-in packages : java , lang etc

user-defined packages

- OR packages are the result of a mechanism to encapsulate a group of classes , sub-packages and interfaces

- Packages are used for

- preventing naming conflicts

**Access Modifiers** : These are the keywords which are used to define the access scope of the method , class , or a variable(field)

**Default** : Default are accessible within the current package only . If there is nothing written then By default , all the classes , methods and variables are of default scope

**Public** : Can be accessed by any package outside of the current package

**Protected** : can be accessed within the package and outside the package only through child class

**Private :** can be accessed within the current class only and can’t even from the other class of current package

- Encapsulation is defined as wrapping up of data under a single unit or in a class . In it the variables or data of the class is hidden from any other class .

- Encapsulation focus on private data

- It allows the programmer to hide and restrict access to sensitive data

Can be achieved by

- Declaring the variables with the private access modifiers

- Create public getters and setters that allow indirect access to those private variables

- As in encapsulation the data in a class is hidden from other classes so it is also known as data-hiding

- To access private infos of a class from outside of the encapsulated class , we use getter and setter methods

- We set the getters and setters as public so that we can access them from other class.

- Hence , indirectly we can access the private fields and methods , and can set values to them with setter and then can get the value back using getter

// Java program to demonstrate encapsulation

public class Encapsulate {

// private variables declared

// these can only be accessed by

// public methods of class

**private** String geekName;

**private** int geekRoll;

**private** int geekAge;

// get method for age to access

// private variable geekAge

**public int getAge**()

{

return geekAge;

}

// get method for name to access

// private variable geekName

**public String getName**()

{

return geekName;

}

// get method for roll to access

// private variable geekRoll

public int getRoll()

{

return geekRoll;

}

// set method for age to access

// private variable geekage

**public void setAge(int newAge)**

{

geekAge = newAge; // It will set the value to the private variable

}

// set method for name to access

// private variable geekName

**public void setName(String newName)**

{

geekName = newName; // It will set the value to the private variable

}

// set method for roll to access

// private variable geekRoll

**public void setRoll**(**int newRoll)**

{

geekRoll = newRoll; // It will set the value to the private variable

}

}

// Class to access variables

// of the class Encapsulate

class TestEncapsulation {

public static void main(String[] args)

{

Encapsulate obj = new Encapsulate();

// setting values of the variables

**obj.setName("Harsh");**

**obj.setAge(19);**

**obj.setRoll(51);**

// Displaying values of the variables

System.out.println("Geek's name: " + **obj.getName());**

System.out.println("Geek's age: " + **obj.getAge());**

System.out.println("Geek's roll: " + **obj.getRoll());**

**// Direct access of geekRoll is not possible**

**// due to encapsulation**

// System.out.println("Geek's roll: " + obj.geekName); **it will give error**

}

}

**4 . Abstraction**

- Abstraction is a process of hiding the implementation details and showing only functionality to the user OR it shows only essential things to the user and hides the internal details .

ex:- sending SMS , where you type the text and send the message and don’t knowing the internal process about the message delivery

Ways to achieve Abstraction

1 . Abstract keyword(abstract classes , properties , methods)

2 . Interface

**abstract class Animal**{

abstract void walk(); // no need to define the function

Animal(){

System.out.println("creating a new animal");

}

public void eat(){ // this is not hidden/abstracted hence , this not a pure abstraction

sout(“eating”)

}

}

class Dog extends Animal{

public void walK(){

System.out.println("walks on fout legs");

}

Dog(){

System.out.println("creating horse");

}

}

class Cow extends Animal{

public void walk(){

System.out.println("walks on four legs");

}

Cow(){

System.out.println("creating dog");

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Animal() // we can't do that since Animal class can't be instanciated

Cow cow = new Cow();// both the constructors will be called , of parent class i.e Animal as well as child class i.e Dog

}

}

- Here , Animal class is simply a concept we don’t need to create its object or instanciate it hence we declare it as abstract

- After creating Cow() object , constructors of both , of Animal(parent) as well as of Cow(child) are called and executed and this concept is called as **chain of constructors**

- Any class containing one or more abstract methods must be declared as abstract otherwise compile-time error will come

- If we don’t want to implement/define a method in a class and we want to leave the definition to a class that extends it we use abstract methods

- To achieve pure abstraction we use interfaces

**features:**-

- An abstract class must be declared as abstract using abstract keyword

- It can have abstract and non-abstract methods

- It cannot be instanciated or we can’t make objects of it

- It can have constructors and static methods also

- it can have final methods which will force the subclass/child-class of it not to change the body of the method

**Interfaces**

- In pure abstraction all the useless informations are hidden and thi can be achieved by interfaces.

- The interface in Java is a mechanism to achieve abstraction and multiple inheritence. There can be only abstract methods in the java interface , and no method body .

- It also can’t be instanciated like abstract class

**interface Animal**{

public void walk(); // it can't have method body as all the methods in an interface are abstract by default

}

**interface Herbivore**{

}

class Horse **implements** **Animal , Herbivore**{ // multiple inheritence

public void walk() // can be implemented/defined in child classes

{

System.out.println("walking");

}

}

public class Main {

public static void main(String[] args) {

Horse horse = new Horse();

horse.walk();

}

}

features:-

- All the fields in interfaces are public , static and final(fixed and can’t be changed) and abstract by default

- All methods are public and abstract by default

- A class that implements an interface must implement all the methods declared in the interface

- Interfaces support multiple inheritance(one class can implement multiple interfaces)

|  |  |
| --- | --- |
| Class | Interface |
| “class” keyword used to create a class | “interface ” keyword used to create interface |
| A class can be instantiated i.e , objects of a class can be created | An interface can’t be instantiated as it follows  abstraction i.e , objects cannot be created |
| don’t support multiple inheritance | supports multiple inheritance |
| it can inherit another class | it can’t inherit a class |
| it can be inherited by another class using the  keyword ‘extends’ | it can be inherited by a class using ‘implements’  and can be inherited by an interface using ‘extends’ |
| can contain constructors | can’t contain constructors |
| can’t contain abstract methods | contains abstract methods only |
| variables and methods in a class can be declared  using any access specifiers(public , private , default  , protected) | All variables and methods in an interface are  declared as public |
| Variables can be static , final , niether | all variables are static and final |

\*\*HEAD OVER GFG FOR DETAILED INFORMATION OF INTERFACE VS CLASS\*\*

**static keyword**

class Student{

String name;

static String school ; // Since all th students will have same school , hence we have declared it as static

public static void changeSchool(){

school = "new-school";

}

}

public class Main {

public static void main(String[] args) {

Student.school = "DPS"; // we can access any static variables of a class using class name directly

// Or , we don't need to create class object first to access a static field

// here we have set the school name for every object of the class unlike names of the students

Student stdone = new Student(); // since name is not common for all hence , it is not static

stdone.name = "tony";

Student stdtwo = new Student();

stdtwo.name - "peter";

}

- In the above code , name is the unique field as it would be different for all the students but school would be common

- We declare the properties or methods as static which are common for all the objects of current class ,and by which we can save some memory

- we can access the static fields directly using the class name because static fields are class thing and not object thing

- static variables belong to class because it is common to all the objects

- All the static variables get memory only once in the class area at the time of class loading and hence it makes your program memory efficient . On the other hand non-static fields get memory again and again

- And , hence , static keyword is used for memoty management mainly

- static can be used with Variables(known as class variables ) , Methods(class methods) , Block , Nested class

// TO COVER UP AT LAST MOMENT for OOPs concepts

\*\*Function/PURPOSE\*\*

\*\*Why use\*\*

\*\*properties and characteristics\*\*

\*\*Real-life examples of all above concepts\*\*

\*\*Advantages and Disadvantages\*\*

\*\*Practical/real-life application\*\*