Introduction to Java

(1.) Java was developed by James Gosling in 1991 at Sun Microsystems.

(2.) Earlier it was called as "Oak".

(3.) First version was released in 1996 Java 1.1.

(4.) In 2010, oracle company acquired java.

(5.) Java is Everywhere

- resides in mobile, client m/cs, Server m/c's, embedded devices, smart phones etc.

(6) Principle of Java WORA

(7.) Java has library – is a collection of predefined classes. we can use these classes either by inheriting them or instantiating them.

(8.) Java Flavours

- Java SE (Standard edition) core j'ava
- Java EE (Enterprise edition) Advance java
- Java ME (Micro edition for mobiles) (diff from Android)

and Many more

UNIT - I, II, III

Introduction to Java

- Java is an object oriented programing language whereas C is a procedure oriented programing language.
- Java language was developed by James Gosling in 1991 at Sun Microsystems. In 1995, the first version of Java i.e Java 1.1 is publically released.
- Java SE 8.0 was released in March 2014.
- Earlier Java was called as "Oak".
- In 2010, Orcale corporation company acquired Java.
- Java is a popular language either ~~for~~ we use it for web based Application or for ~~we can use it for writing cod~~ games.
- Java is easy, Simple, Robust, Secure.
- Features of Java.

① Simple & secure.

as it is based OOP, if you Know C++ you can easily learn Java.

we use it for web based App^n and when these App^{n}s run on Internet its byte code is verified

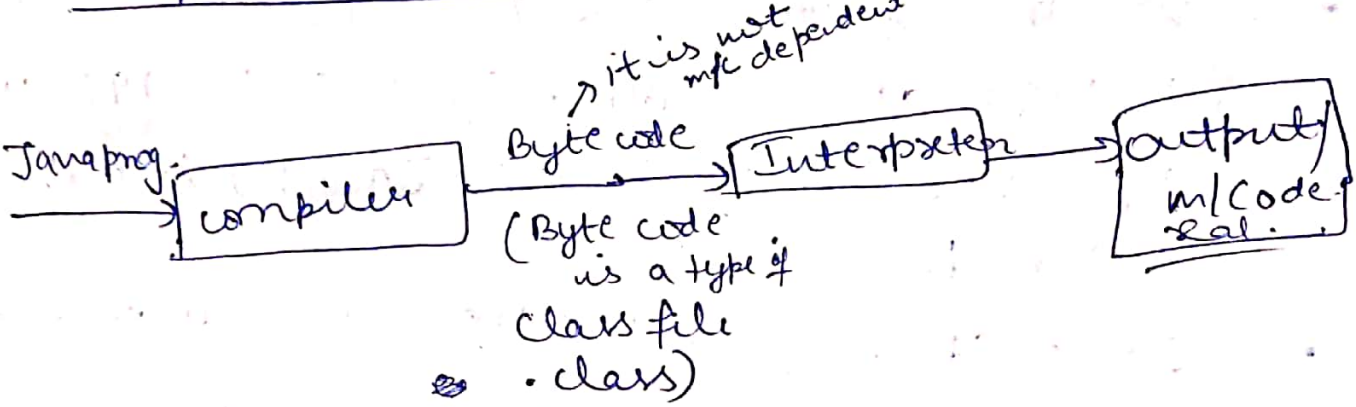
we don't use pointers in java

(2) Portable / M/c independent.

Write once run anywhere.

∴ Java byte code is generated after compilation., and byte code is easily run on any other m/cs.

(3) Compiled and Interpretted.



whenever we make a java program either in any editor or on a plain text and when compiler compiles the Java prog., a class file is generated as .class (extension is .class) and is known as Byte code or it is known as intermediate code. And this Byte code is interpretted on different m/cs.

The Interpreter of any m/c is interpreted according to its m/c. and generated its m/c code, which run on its m/c and we get the o/p.

And this can be done through JVM.

Java introduces JVM.

- JVM — Java Virtual M/c. (It is one of the Specific feature of Java.)

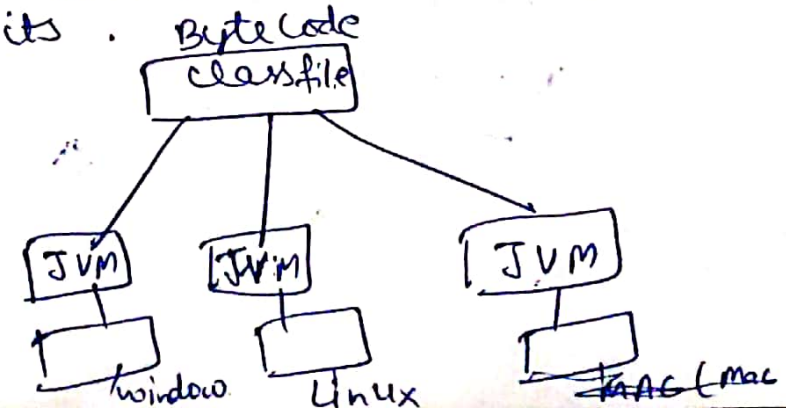
It is not a m/c actually, it is a S/w, but it works as a m/c so it is known as Virtual m/c. JVM interpreters the Java bytecode

The main fuction of JVM is to accept the Byte code and generate the m/c code. When the Byte code is generted on the m/c then JVM on the other m/c interprets that Byte code and generating the m/c code according to the m/c.

(4) Distributed — means the Java progs can run on network i.e remote progs can run on local m/cs. Java has Java.net package

(5) Multi-threaded — thread is a small prog or a part of prog. and multithreading means small prog run concurrently. Eg:- When we have prog for any game, then the small progs are runing concurrently.

(6) Object Oriented → It uses objects and classes means data & method are encapsulated in a class and this class is implemented with help of objects.



Multithreading → perform a para... task, we can run small progs. In Java, we have Java.lang package ha... (we can make threads).

(7) Dynamic → new methods can be loaded du... runtime and these methods are native methods. also link ... libraries ...

(8) Robust → Robust means reliable, early checking for possible errors, automatic garbage collection, exception handling, type checking makes the system secure.

UNIT-I

Object oriented Programming concepts (OOPs concept) + operators

① **Objects** → Any real world entity that has State and behavior is known as an object. For eg:- chair, pen, table etc. It can be physical and logical.
We can create objects in ~~ja~~ j'ava using new ~~java~~ Keyword.

Syntax -:

```
classname objectname = new classname();
```

② **Class** → A class can be defined as a template or blueprint that describes the behaviors / State that objects of its type support.

Syntax

```
class classname
{
    Body of class
}
```

Eg:-

```
class Hello
{
    public static void main (String ar[])
    {
        System.out.println ("Hello");
    }
}
```

③ Inheritance → The process by which one class acquires the properties and functionalities of another class. Inheritance provides the idea of reusability of code and each subclass defines only those features that are unique to it.

The existing class is called the base class or super class or parent class. The new class which inherits from the base class is called the derived class or subclass or child class.

④ Polymorphism → when one task is performed by different ways. In java, we use method overloading and method overriding to achieve polymorphism.

⑤ Abstraction → The process of hiding the (internal details) unnecessary details from the users and show only relevant information. In java, we use abstract class and interface to achieve abstraction.

Encapsulation → Binding (or wrapping) code and data together into a single unit is known as encapsulation. A java class is the example of encapsulation.

(7.) Message Passing → One object invoking methods on another object is known as Message Passing.

(8.) Dynamic Binding → when compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or Late binding.
Eg:- Method Overriding.

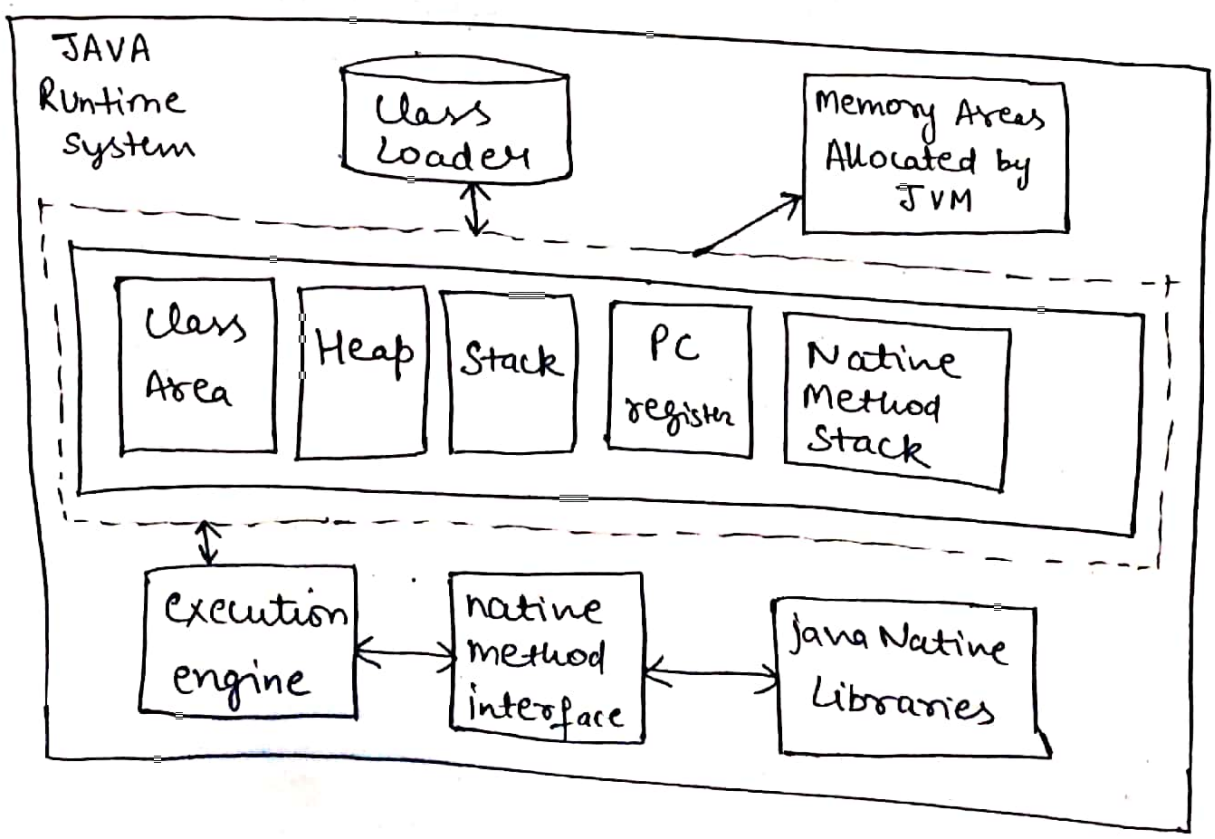
→ JVM

JVM is a java virtual machine or a prog. that provides run-time enviornment in which java byte code can be executed. JVMs are available for many hardware and software platforms.

The JVM performs following operation -:

- Loads code
- Verifies code
- Executes code
- provides runtime environment.

→ Internal Architecture of JVM



(a) Classloader → It is a subsystem of JVM that is used to load class files.

(b) Class (Method) Area → Stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

(c) Heap → It is the runtime data area in which objects are allocated.

(d) Stack → It holds local variables and partial results, and plays a part in method invocation and return.

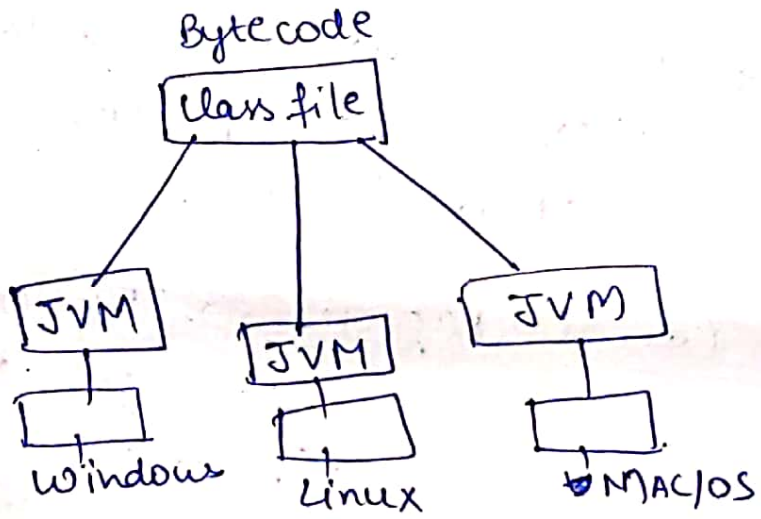
(e) Program Counter Register → It contains the address of the java virtual machine instruction currently being executed.

(f) Native Method Stack → It contains all the native methods used in the application.

(g) Execution Engine → It contains virtual processor, interpreter, Just-In-Time (JIT) compiler.

(→ Bytecode

Java bytecode is the instruction set of the java virtual machine. Bytecode is the compiled format for java programs. Once a java program has been converted to bytecode, it can be transferred across a network and executed by JVM. Bytecode files generally have a .class extension.

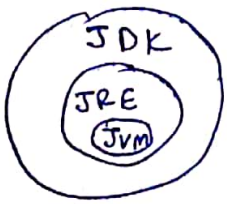


Difference b/w JVM, JDK, JRE

JVM — runs the bytecode and generate the m/c code.
it is different ∵ (WORA).

JRE → Java Runtime Environment.
- provides the libraries
  - JVM
  - other components to run applets & app^{n}s.
  - java plug-in, which enables applets to run in browsers.
  - doesn't contain tools and utilities such as compilers or debuggers.

JDK — java development kit
- is superset of JRE and contains everything that is in JRE. + tools such as compilers, debuggers.



getting i/p from user

For i/p we have a class Scanner, which is in a package Utility.

```
import java.util.Scanner;
```

java → main package; util → subpackage; Scanner → class

To access the methods of the Scanner class, we need to create the objects of this class.

```
Scanner obj = new Scanner(System.in);
```

Scanner → predefined class; System.in (arguments) → class object

So to take the i/p from this class we need to pass arguments

for Standard i/p → Keyboard
" " o/p → monitor

Scanner is final class.

Methods in Scanner class:-

```
nextInt()    → for integer
nextFloat()  → for float
nextline()   → for String or line
```

eg:-

```
int a;
a = obj.nextInt();
```

Value from user is stored in a.

Prog to display a Square of a no.

```
import java.Util.Scanner;
  Class Test
   { public Static Void main (Strin.args[])
     {   int no;
         Scanner Obj = new Scanner (System.in);
         System.Out.println ("Enter a no.");
        no = obj.nextInt ();
       System.Out.println ("Square is " + (no*no);
     }
   }
```

↓ to print the value of a variable

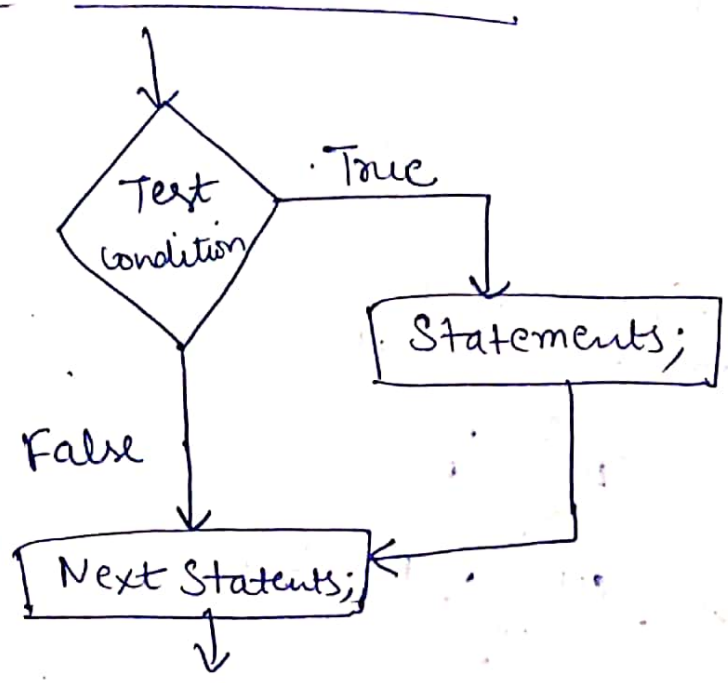
→ Decision Making and Branching

1. IF Statement
2. Switch statement
3. conditional operator Statement.

IF Statement

1. Simple IF
2. IF - - Else - -
3. Elseif - - IF ladder Statement
4. Nested IF- Else Statement.

IF Statement in Java



Syntax

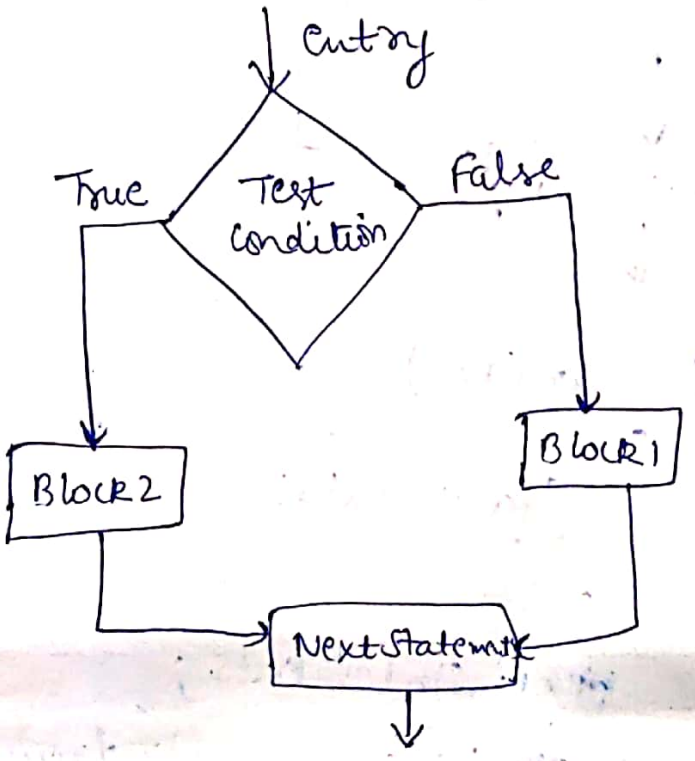
```
if (condition)
  {
    Statements;
    = =
  }
next statements;
```

eg:- WAP to take salary as i/p from user and if the salary is greater than 10,000 then give 10% bonus on salary.

```
import java.util.Scanner;
  class Test
  {
    public static void main (String args[])
    {
      Scanner obj = new Scanner (System.in);
      int sal; ,b;
      System.out.println ("Enter your salary");
        sal = obj.nextInt();
        if (sal >= 10000)
          { b = (sal * 10)/100;
            sal = sal + b;
          }
```

```
System.out.println("Salary is "+sal);
  }
}
```

If Else Statement



Syntax

```
if (condition)
  {
    Block1;
  }
else
  {
    Block2;
  }
nextStatements;
```

3:- If Salary is more than 10,000 then bonus is 10%. else 5%.

```
import java.Util.Scanner;
Class Test
{
    public Static Void main(String args[])
    {
        Scanner obj = new Scanner(System.in);
        int sal, b;
        System.out.println("Enter your Salary");
        int Sal = obj.nextInt();
        if(Sal >= 10000)
        {
            int b = (Sal * 10) /100;
            Sal = sal + b;
        }
        else
        {
            int b = (Sal * 5) /100;
            Sal = Sal + b;
        }
    System.Out.println("Total Salary is" + sal);
    }
}
```

Nested If Statement

Syntax

```
if (condition 1)
  { if (condition 2)
      { statement 1;
      }
    else
      { statement 2;
      }
  }
else
    { statement 3;
    }
```

Eg:-

```
import java.util.Scanner;
Class Test
  {
    psvm (String ar[])
    {
      Scanner obj = new Scanner (System.in);
      int a, b, c;
    System.out.println("enter first no.");
        a = obj.nextInt();
    System.out.println("enter second no.");
        b = obj.nextInt();
    System.out.println("enter third no.");
        c = obj.nextInt();
```

```
if (a>b)
{  if (a>c)
     { System.out.println("greater value is"+a);
     }
     else
       { System.out.println("greater value is"+c);
       }
}
  else
    { if (b>c)
       { System.out.println("greater value is"+b);
       }
      else
        { System.out.println("grea is"+c);
        }
    }
```

→ **Else if Ladder in Java.**

Syntax.

```
if (condition 1)
   { Statement 1;
   }
else if (condition 2)
   { Statement 2;
   }
   . . .
else if (condition-n)
   { Statement n;
   }
else { Default statement
     ;
```

while loop in Java

Syntax
```
while (condition)
{ Statement;
  ...
}
```

eg:- print odd no's b/w 1 to 100

```
Class Test
{ psvm (String args[])
  {
    int no; = 1;
    while (no < 100)
    { sop (no.);
      no = no+2;
    }
  }
}
```

do while loop

Syntax
```
do { statement;
  ...
}
while (condition);
```

eg:- print even no's b/w 1 to 100

```
Class Test
{ psvm ( - -)
  { int no.=2;
```

```
do
{ S.O.P(no);
  no. = no.+2;
}
while (no. <=100);
}}
```

For loop in java

Syntax for (initialization; condition; increment/decrement).

```
{
  ~~field declaration~~;
  ~~method declaration~~;
  Body of loop;
}
```

Eg:- prog. print even no's b/w 1 to 100.

```
Class Test
{ psvm(--)
  { int no.;
    for(no=2; no<=100; no.=no.+2)
    { S.O.P(no);
    }
  }}
```

→ Enhanced forloop — introduced in java 5. It provides a simpler way to iterate through the elements of a collection or array. It is flexible and its used in sequential manner without knowing the index of currently processed element.

```
for (Telement : collection/array)
{ Statements
}
```

For-each loop / enhanced for loop

J2SE 5.0

- traverse array or collection (Arraylist)
- makes code readable
- eliminates errors
- drawback - can't traverse in reverse order
- don't have option to skip any element.
- Syntax for (data type variable : array)

```
{ =
}
```

```
class Sample
{ psvm (--)
  { int a[] = {1, 2, 3};
    for (int k: a)
    { s.o.p(k);
  } }
}
```

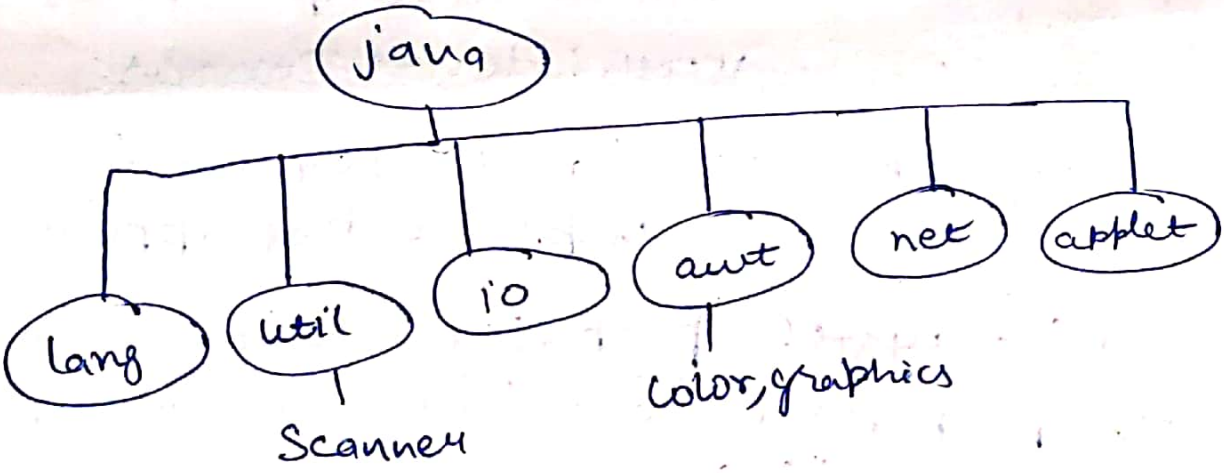
→ Example of Java Program to take input from user

```
import java.util.Scanner;
public class Student
{  psvm (String args[])
   {  String name;
      int roll;
      Scanner obj = new Scanner (System.in);
      System.out.println ("Enter name" +roll.+ "");
      name = obj.nextLine();
      roll = obj.nextInt();
      System.out.println ("roll no is" +roll);
      System.out.println ("name is" +name);
   }
}
System.out.println (" +roll+ ", " +name);
```

→ Packages are java's way of grouping a number of related classes and/or interfaces together into a single unit. That means, package act as a 'containers' for classes.

→ Java API packages

Java Application programming interface (API) is a list of all classes that are part of the java development kit (JDK). It includes all java packages, classes, interfaces along with their methods, fields and constructors.



java.lang — Language support classes. These are classes that java compiler itself uses and ∴ they are imported automatically. They include classes for primitive types, Strings, math functions, threads, exceptions.

java.util — Language utility classes such as vectors, hash tables, random numbers, date etc.

java.io — i/o support classes. They provide facilities for the i/o of data.

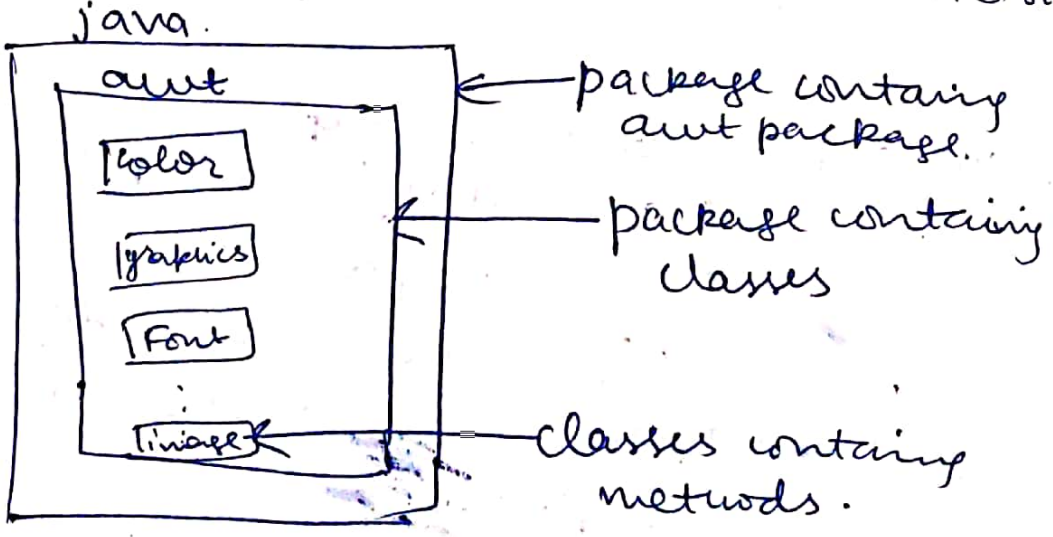
java.awt — Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

java.net — classes for networking. They include classes for communicating with local computers as well as with internet servers.

java.applet — classes for creating and implementing applets.

→ Using System packages.

The packages are organised in a hierarchical structure.



Packages

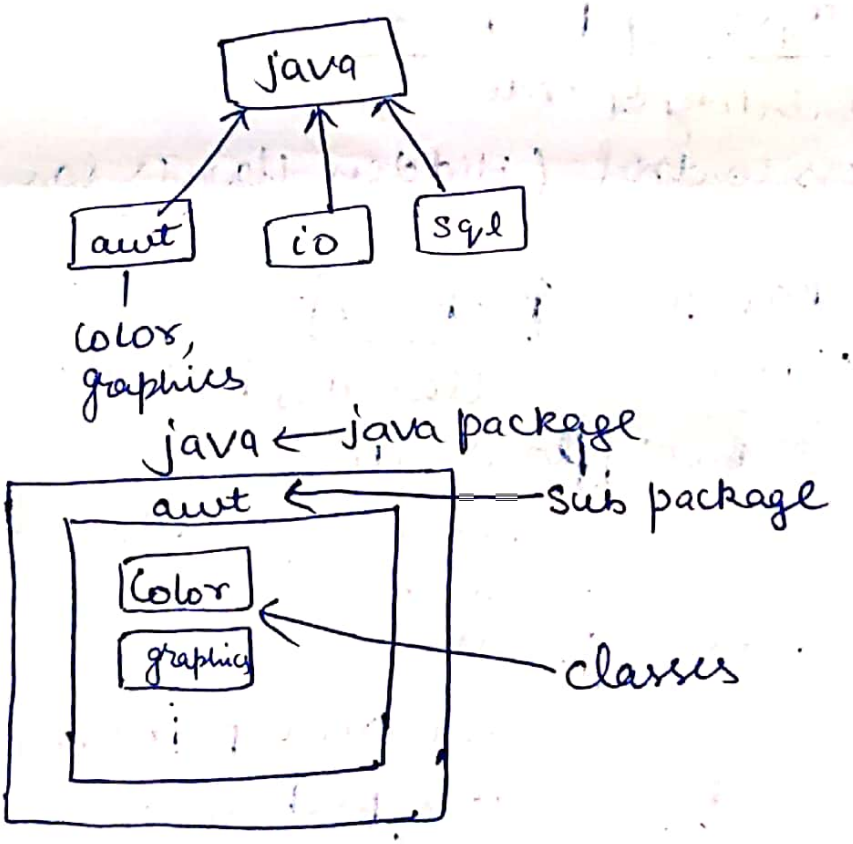
Packages are a collection of related classes, interfaces. It is like a container. (classes are categorised based on their functionality)

- Based on reusability. Like in inheritance we use the code by inheriting the related class.

→ Types of Packages → folder

1. Built-in Packages.

Eg:- java.io (contains i/o related classes)
java.lang
java.util.Scanner



Package is like a folder in which we have related classes. We use packages to organise the classes/interfaces in a container or folder.

(2) User defined packages

If the project is large, we have more number of classes, then we can make ~~~~ our own package.

~~be~~ when we create our own package, we have to follow some rules -:

Naming convention

(a) Name of the package should be unique.

(b) Name should be in lower case.

in case of class eg: Sample

" " method eg:- dispName

→ Advantages of Packages

(1) provide Reusability of code.

(2) Access control (Hidden classes cannot be accessed outside the package)

(3) Package1 — ClassA

Package2 — ClassA ✓ possible

But Program1 — ClassA, ClassA ✗ not allowed.

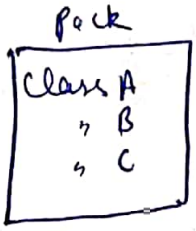
(4.) Easy to organise

(5.) can create our own package or extend already available package.

Accessing packages

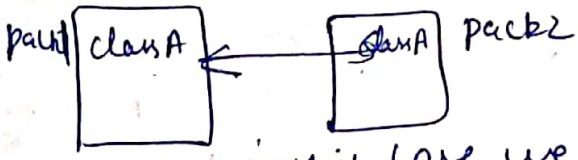
(1.) Fully qualified classname (rarely used).
(2.) import statement

(1') eg:-



pack.A obj = new pack.A(); } → fully qualified classname.

java.awt.color obj = new java.awt.color();

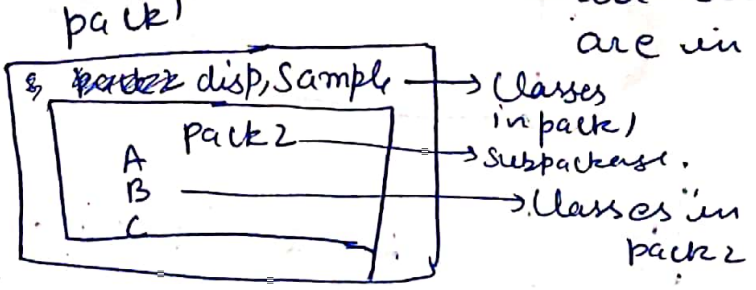


in this case we use fully qualified Class name.

(2.) Import Statement

can be used in two ways:-

(a) import package.(*)
↳ means we can access all classes, interfaces which are in package.



if we use import pack1.*;
↳ then we can access only disp, Sample classes.
we cannot access subpackages classes.

(5) import.package.classname;

eg:- import java.util.Scanner;

→ Creating and Using packages:

source file

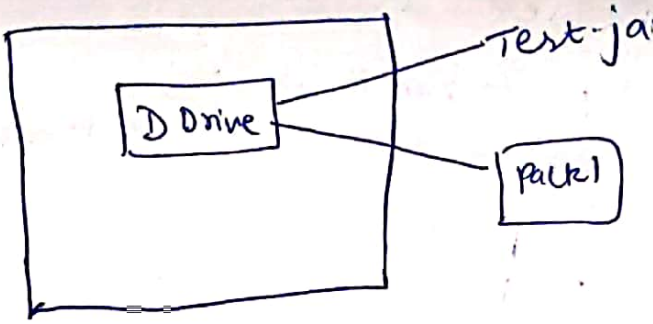
(1) package packagename;

(2) public class classname → when we save a program we use a classname which is public.

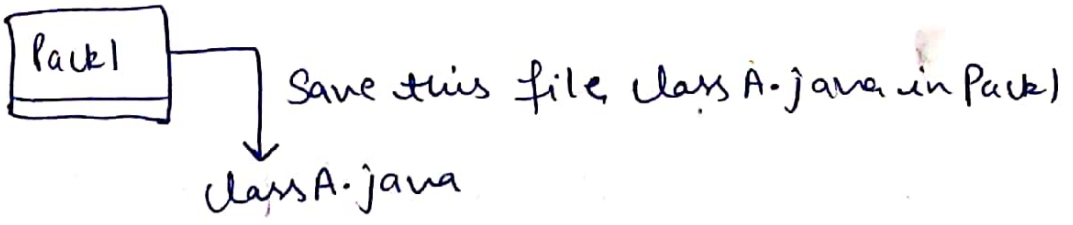
{
=
=
}

(3) To save. classname.java
↳ public class.

(4) create a folder, foldername is same as package name.

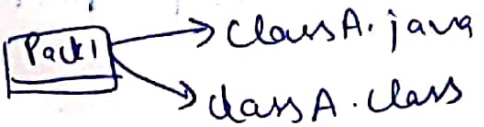


(5)



(6) Compile

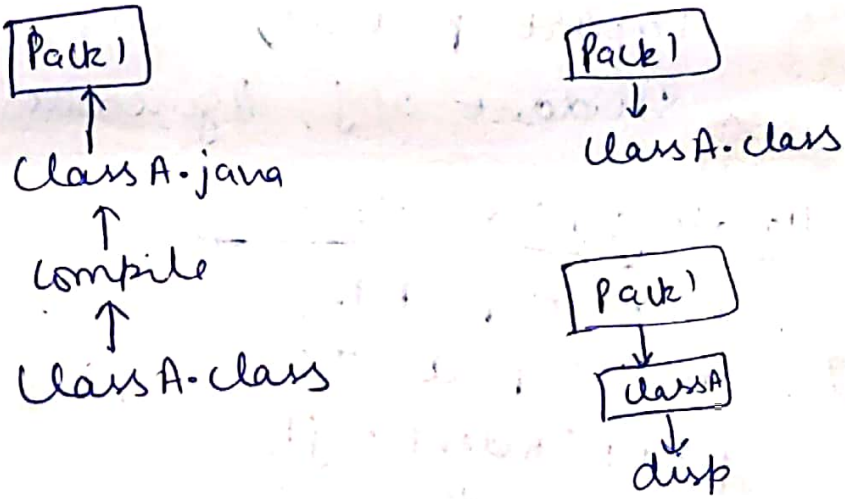
(7) classA.class



Example Source code1

```
package pack1;
public class classA
{   public void disp()
    {   SOP("Hello");
    }
}
```

Create a folder in any drive Eg: D drive.
The folder name should be pack1. and in this folder save the file classA.java.

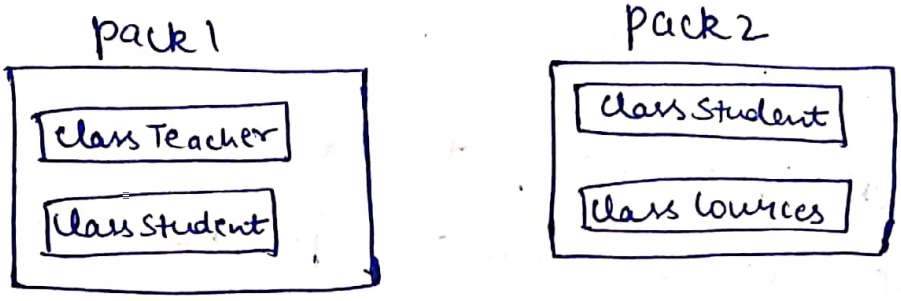


Source code2.

```
import pack1.*; or pack1.classA;
{ class Test {
    psvm(--)
    {   classA obj = new class();
        obj.disp();
    }
}
```

→ Name clashing

when packages are developed by different Organizations it is possible that multiple packages will have classes with same name, leading to name clashing.




```
import pack1.*;
import pack2.*;
Student obj; // generates compilation error.
```

Handling Name clashing

```
import pack1.*;
import pack2.*;
pack1.Student obj1;
pack2.Student obj2;
  Teather Obj3;
   Courses Obj4;
```

UNIT-IV

→ **Inheritance** → The process by which one class acquires the properties and functionalities of another class. Inheritance provides the idea of reusability of code and each subclass defines only those features that are unique to it.

→ **Types of inheritance**

(1) **Single Inheritance**

Example

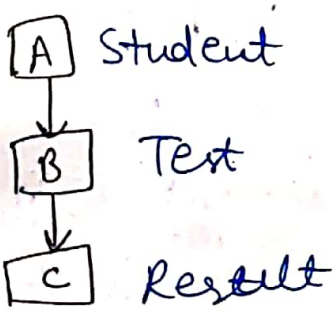
```
class A
{   int x=2;
    int y=3;
    void add()
    {   System.out.println("Addition is "+(x+y));
    }
}
class B extends A
{   void mul()
    {   System.out.println("Multiplication is "+(x*y));
    }
}
```

```
class Test
{ public static void main (String ar[])
    { B obj = new B();
      obj.add();
      obj.mul();
    }
}
```

Output

Addition is 5
Multiplication is 6

(2) Multilevel Inheritance



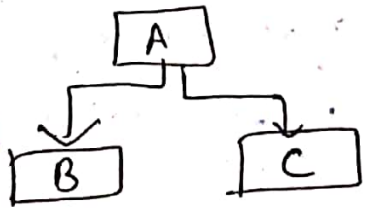

```
class Student
{ int roll;
  void getroll(int n)
    { roll=n;
    }
  void putroll()
    { System.out.println("roll no is" + roll);
    }
}
class Test extends Student
{ int m1, m2;
  void getmarks(int x, int y)
    { m1=x;
      m2=y;
```

```
void putmarks()
{ System.out.println("Test1" + m1);
  System.out.println("Test2" + m2);
}
}
class Result extends Test
{ int total;
  void display()
  { putroll();
    putmarks();
    total = m1 + m2;
    System.out.println("Total marks" + total);
  }
}
class Main
{ public static void main(String ar[])
  { Result obj = new Result();
    obj.getroll(101);
    obj.getmarks(70, 80);
    obj.display();
  }
}
```

Output

```
roll no is 101
Test 1 70
Test 2 80
Total marks 150
```

(3) Hierarchical Inheritance

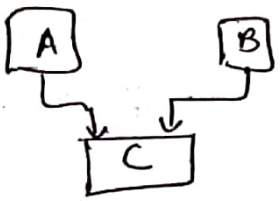



```
Class one
{ int n=10;
  int y=20;
  void display ()
    { System.out.println ("value of n"+n);
      System.out.println ("value of y"+y);
    }
}
Class Two extends one
  { void add ()
    { System.out.println ("addition is"+(n+y));
    }
  }
Class Three extends One
  { void mul ()
    { System.out.println ("Multiplication is"+(n*y));
    }
  }
Class Main
  { public static void main (String ar[])
    { Two obj1 = new Two ();
      Three obj2 = new Three ();
      obj1.display ();
      obj1.add ();
      obj2.mul ();
    }
```

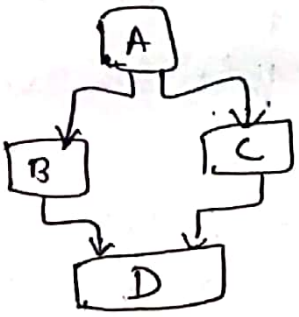
Output

Value of x 10
Value of y 20
addition is 30
Multiplication is 200

④ Multiple Inheritance



⑤ Hybrid Inheritance

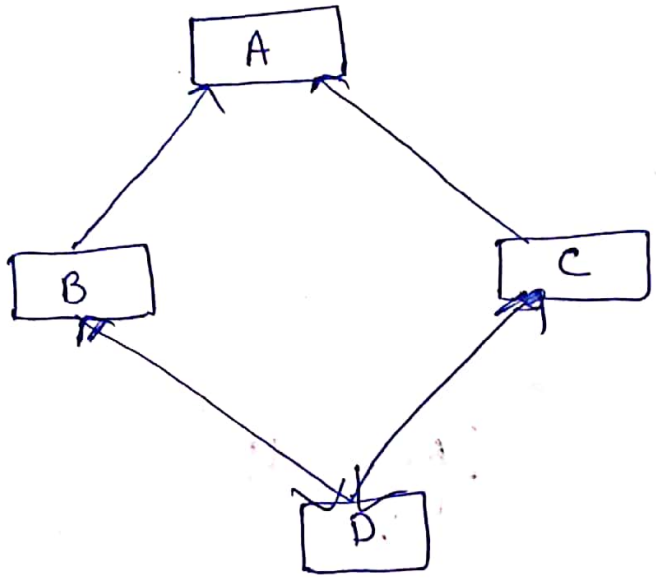


Multiple ~~Hybrid~~ Inheritance doesn't ~~possibility~~ supported in Java.

Sometimes, it results in ambiguity.

For eg!-

Example



In this diagram, we have two classes B and C inheriting from A. Assume that B and C are overriding an inherited method and they provide their own implementation. Now, D inherits from both B and C doing multiple inheritance. D should inherit that overridden method, which overridden method will be used? will it be from B or C? Here, we have an ambiguity. Therefore, multiple inheritance is not supported in Java. In this diagram, the classes form the shape of a diamond. Therefore, this problem is called the diamond problem.

Program

```
class A
{  void display ()
     { System.out.println("Hello");
     }
}
class B extends A
{   void display ()
     { System.out.println ("Hi");
     }
}
class C extends A
{  void display ()
     { System.out.println ("Bye");
     }
}
class D extends B, C
{    void MethodD ()
       { System.out.println (" Method D");
       }
}
class Test
{ public ~~static~~ void main (String ar[])
     {  D obj = new D();
           obj.display();    //error
     }
}
```

Output error.

→ Method overriding ✓

If Subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.

If Subclass provides the specific implementation of the method that has been provided by one of its parent class, Method overriding is used for runtime polymorphism.

→ Rules for Method overriding

1. Method must have same name as in the parent class.
2. Method must have same parameter as in the parent class.
3. Must be Is-A relationship (Inheritance).

Example

```
class A
 { void display()
      { System.out.println("Hello");
      }
}
class B extends A
   { void display()
        { System.out.println("Bye");
        }
   }
```

```
class Test
{ public static void main (String arg[])
    { B obj = new B();
      obj.display();
    }
}
```

output

Bye

In this example, subclass method overrides the superclass method.

Method Overloading	Method Overriding
(1) It happens at compile time.	(1) It happens at run time.
(2) Static methods can be overloaded.	(2) Static methods cannot be overridden.
(3) Overloading is being done in the same class.	(3) For overriding, base and child classes are required
(4) Static binding is being used for overloaded methods.	(4) Dynamic binding is being used for overriden / overriding methods.
(5) Private and final methods can be overloaded.	(5) private and final methods cannot be overridden

(6) Return type of method does not matter in case of method overloading.	(6) In case of overriding the overriding method can have more specific return type.
(7) Argument list should be different in method overloading.	(7) Argument list should be same in method overriding.
(8) Example of overloading int add (int a, int b) int add (int a, int b, int c)	(8) Example of overriding class A { int speed() { return 100; } } class B extends A { int speed() { return 50; } }

Final keyword in java

The final keyword in java is used to restrict the user. It can be used in many ways:

Final can be -:

(1) Variable (2) method (3) class.

- **Final Variable** → If we make any variable as final, we cannot change the value of final variable (it will be constant).

Example of final variable

```
Class x
  { final int n=0;
      n = n+2; //error
//The value of n is constant.
```

- **Final methods** → If we make any method as final, we cannot override it.

Example of Final Method

```
Class X
  { final void display()
      { System.out.println("Hello");
      }
}
class Y extends X
  { void display()
      { System.out.println("Bye"); //error
      }
}
```

// final method cannot be override.

- Final class → If we make any class as final, we cannot extend it (we cannot inherit it)

Example of Final class

```
final class X
  {
  }
class Y extends X
        → error.
```

// final class cannot be inherited.

→ Super keyword in java..

Super keyword in java is a reference variable that is used to refere immediate parent class object.
It is used in subclass, when the subclass has same name and same method as in superclass.

Super keyword is used in two ways:-

(1) Accessing super class variables and Methods.

(2) calling super class constructors.

Accessing super class Variables

Syntax: super. Variable name.

Example:

```
class A
  { int no;
    =
  }
class B extends A
  { int no;
    Void display()
      { no = 100;
        super.no = 200;
  }   }
```

- **Accessing super class Method**

Syntax: Super. method name();

Example:

```
class A
  { Void message()
    { =
    }
Class B extends A
  { void message()
    { =
    }
Void display()
  { message();
    super. message();
  }   }
```

→ Program which shows the usage of super variables and super methods

```
class A
 { int no;
   void message()
   { System.out.println("no in super class"+no);
   }

class B extends A
 { int no;
   B(int a, int b)
     { super.no = a;
       no = b;
     }
   void message()
   { System.out.println("no in sub class "+no);
   }
   void display()
     { super.message();
       message();
     }

class Test
 { public static void main(String a[])
     { B obj = new B(100, 200)
       obj.display();
     }
 }
```

Output

```
no in superclass 100
no in subclass 200
```

Calling Superclass Constructors

(Default Constructor)

```
class A
 {   A()
       {=
      }
  }
Class B extends A
       {  B()
            { super();
              =
           }
       }
```

Syntax super(); (Optional) Because, it can call automatically.

Example

```
class A
      {  A()
           {System.out.println("Hello");
           }
      }
Class B
   {   B()
         {  Super();
            System.out.println("Bye");
         }
   }
class Test()
   {
```

```
    public static void main (String arg[]
        B obj = new B();
    }
}
```

Output

Hello
Bye.

→ (Parameterized Constructor)

Example

```
Class A
{ int a;
    A (int x)
    { a = x;
      System.out.println ("Value of
                          a is" + a);
    }
}
Class B extends A
{ int b;
    B (int x, int y)
    { super (x);
      b = y;
      System.out.println ("Value of
                          B is " + b);
    }
}
```

```
Class Test
{ public static void main (String ar[])
    {  B obj = new B(2,3)
    }
}
```

Output

Value of a is 2
Value of b is 3

→ Abstract Method

A method that is declared as abstract and does not have implementation is known as abstract method.

~~abstract void print~~

Example

```
abstract void display ();
// no body.
```

→ Abstract class

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instanticated.

Example

```
abstract class A
{
```

- Abstract class can have abstract method as well as normal methods.
- we cannot make objects of abstract class.

→ Example of abstract class and abstract method

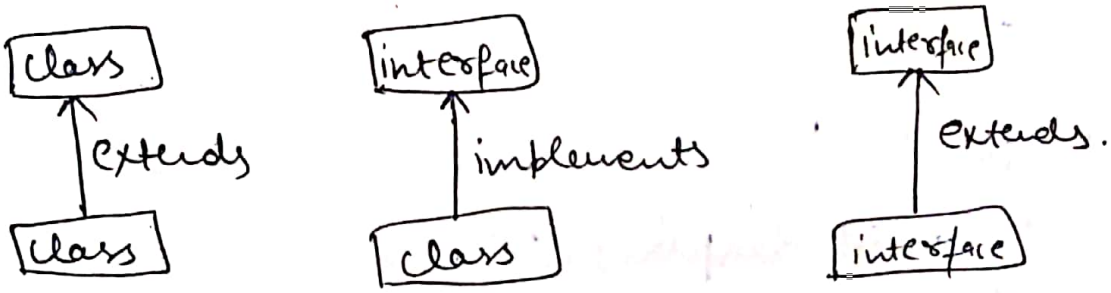
```
abstract class X
  { abstract void disp();
    void display()
      { System.out.println("Method of X");
      }
  }
class Y extends X
  { void disp()
      { System.out.println("Method defined
                            in Y");
      }
  }
class Test
  { public static void main(String args[])
      { Y obj = new Y();
        obj.disp();
        obj.display();
      }
  }
```

Output

Method defined in Y
Method of X

→ Interfaces

A interface is a blueprint of a class. It has final variables and abstract methods only.
Interface in java is used to achieve fully abstraction and multiple inheritance.

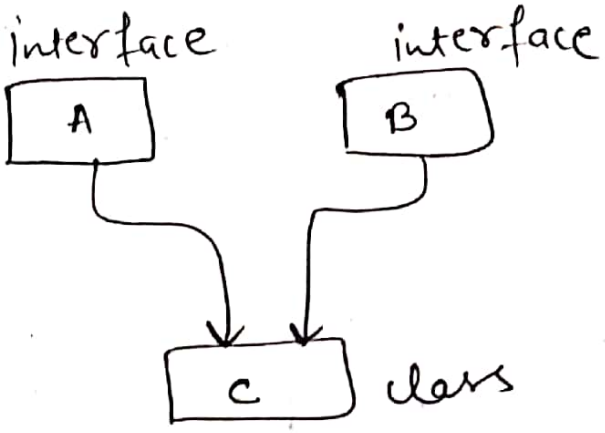


A class extends another class, an interface extends another interface but a class implements an interface.

Syntax

class classname implements interface name

Example



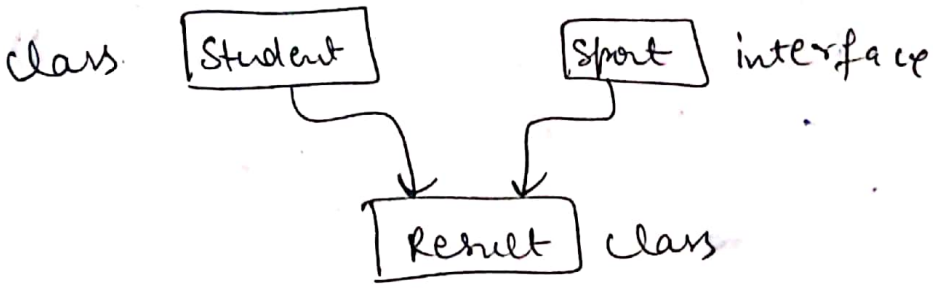

```
interface A
  { int roll = 101;
      void disp();
  }
interface B
  { void display();
  }
class C implements A, B
  {
      public void disp()
        { System.out.println("roll no. is" + roll);
        }
     public void display()
        { System.out.println("Method from B
                              interface");
        }
  }
```

```
class Test
{ public static void main (String args[])
    { C obj = new C();
      obj.disp();
      obj.display();
    }
}
```

Output

Roll no is 101
Method from B interface.

→ Another Example



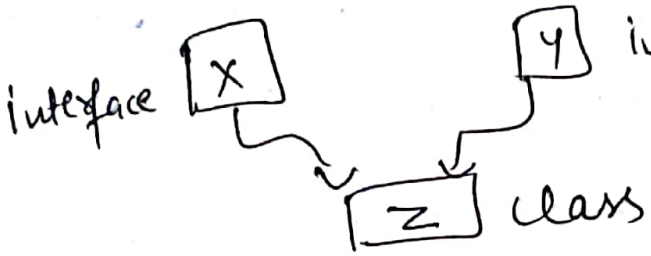

```
class Student
  { int m1, m2;
    void getmarks (int x, int y)
      { m1 = x;
        m2 = y;
      }
    void putmarks ()
      { SOP ("Marks1" + m1);
        SOP ("Marks2" + m2);
      }
  }
```

```
interface Sport
    { int sp = 6;
      void Spmarks();
    }
Class Result extends Student implements Sport
  { public void Spmarks()
      { SOP("Sport marks" + sp);
      }
    void display()
      { putmarks();
        Spmarks();
        int total = m1 + m2 + sp;
        SOP("total marks" + total);
      }
  }
Class Main
  { public static void main(String ar[])
      { Result obj = new Result();
        obj.getmarks(80, 60);
        obj.disp();
      }
  }
```

Output

```
marks 1 80
marks 2 60
sport marks 6
Total marks 146
```

Multiple inheritance using interface




```
interface X
 { public void display ();
   }
interface Y
  { public void display();
  -}
class z implements X, Y.
     { public void display ()
        {System.out.println("Hello");
     } }
Class Test
  { public static void main(String arc[])
       { z obj = new z();
         obj.display();
   }
```

output — Hello

As we can see that the class implemented two interfaces. In this case, there is no ambiguity even though both interfaces are having same method. Because methods in an interface are always abstract, which does not let them to give their implementation (method defination).

→ Difference between Abstract class and Interface

Abstract class	interface
(1.) Abstract class can have abstract and non-abstract methods.	(1.) Interface can have only abstract methods.
(2.) Abstract class can have final, non-final variables	(2.) Interface can only have final variables
(3.) Abstract class can provide the implementation of interface	(3.) Interface cannot provide the implementation of abstract class.
(4.) The abstract keyword is used to declare abstract class.	(4.) The interface keyword is used to declare interface.

abstract class	Interface
(5.) Abstract class does not support multiple inheritance.	(5.) Interface supports multiple inheritance.
(6.) Example abstract class shape { abstract void draw(); void display() { SOP("shape"); } }	(6.) Example interface shape { void draw(); }

→ Constructor in java

Constructor in java is a special type of method that is used to initialize the object

java constructor is invoked at the time of object creation. It constructs the values i.e it provides data for the object that is why it is known as constructor.

- Constructor name must be same as its Class name.
- Constructor doesn't have return type.
- Constructor can be public, private or protected.
- Constructor defination is not mandatory in class.

→ Type of Constructors

Default constructor

A constructor that have no parameters is known as default constructor.

Syntax of default constructor

```
Classname ()
  {
  }
```

Example of default constructor

```
class Test
{   Test()
    { System.out.println("This is a default constructor");
    }
  public static void main(String arg[])
    { Test obj = new Test();
    }
}
```

Output

This is a default constructor.

Parameterized Constructor

A constructor that have parameters is known as parameterized constructor. These constructor are used to provide different values to different objects.

Syntax of parameterized constructor.

```
classname (---parameters)
  {
  }
```

Example of parameterized constructor

```
class Test
{ int a;
  Test (int x)
   { a=x;
     System.out.println ("Value of a is"
                         +a);
   }
public static void main (String ar[])
   { Test obj = new Test (2);
   }
}
```

Output

Value of a is 2

→ Method Overloading

If a class have multiple methods by same name but different parameters, it is known as Method Overloading.

Advantage of Method Overloading

It increases the readability of the program.

Different ways to overload the method

① By changing number of arguments
② By changing the data type.

* Method overloading is not possible by changing the return type of method.

Example :- int add (int n, int y)
int add (float a, float b)

Example Program of Method Overloading

```
class Add
{
    void add (int a, int b)
    {System.out.println((a+b));
    }
    Void add (int a, int b, int c)
    { System.out.println((a+b+c));
    }
```

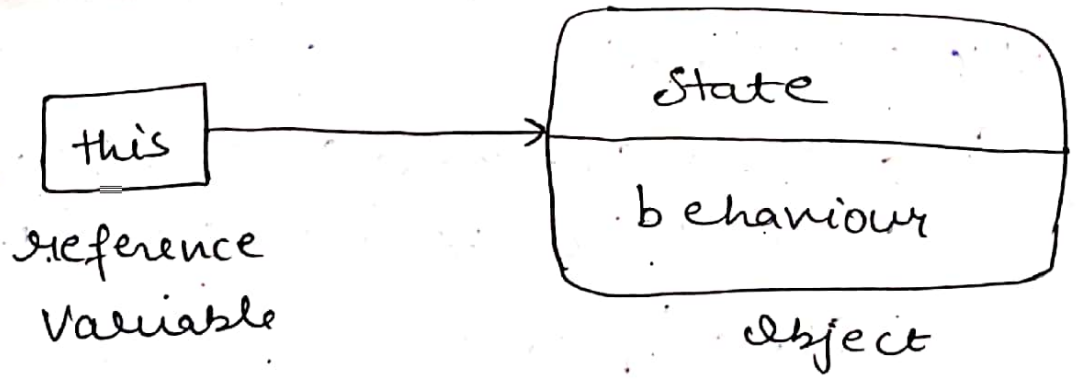
```
class Test {
public static void main (String ar[])
    {   Add obj = new Add ();
        obj.add (2,3);
        obj.add (4,5,6);
    }
}
```

Output

5
15

→ this Keyword

this is a reference variable that refers to the current object.



use of this Keyword

(1) this can be used to refer the current class instance variable.
(2) this() can be used to invoke current class constructor.
(3) this can be used to invoke current class method.

Example of this keyword

```
class Student
{ int id;
  String name;
  Student (int id, String name).
    { this.id = id;
      this.name = name;
    }
  void display ()
  { System.out.println ("id is" +id);
    System.out.println ("name is" +name);
  }
public static void main (String ar[])
  { Student S1 = new Student (1, "Nisha");
    Student S2 = new Student (2, "Neha");
    S1.display ();
    S2.display ();
  }
}
```

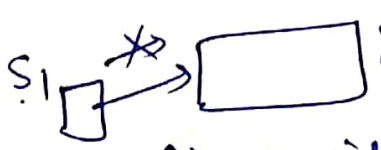
Output

```
id is 1
name is Nisha
id is 2
name is Neha
```

⑤ Automatic Garbage collection in java

- In java, destruction of object from memory is done automatically by the JVM.
- No delete keyword in java.
- when there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released.
- This technique is called G.C.
- This is accompalished by the JVM.

Student s1 = new Student();



(java will delete this object when s1 will not point to object.

eg:- s1=null;

or s1 = will have diff. address.

- whenever we run a java program, JVM creates 3 threads.
  - Main thread — executes main fn or prog.
  - Thread Scheduler
  - GC — garabage collector frees gc blocks.

→ Finalize method. — this is defined in object class

- GC calls finalize method before sweeping out an abandoned object.
- After finalize() method is executed, object is destroyed from memory.

→ Object can be unreferenced in 3 ways:

(1) by nulling the reference.

(2) By assigning a ref. to another

(3) By anonymous object.

```
Employee e = new Employee();
e = null;

e1 = e2;

new Employee();
```

method.
gc is found in System and Runtime classes.

```
Class A
{ psvm(--)
  { A obj1 = new A();
    A obj2 = new A();
    obj1 = null;
    obj2 = null;
    public void finalize();
    System.gc(); { SOP("GC");
```

output
GC
GC