13.03.23

JAVA

It is high-level object-oriented programming language

object of class

class classname
{
variables + methods
}

members → static members
members → non-static / Instance members

data members
function members

class marker
{
① static int ht = 10;
② double price = 200;
③ static void writer()
{ = }
④ static void draw()
{ = }
}

TO access static members:
[static reference syntax]

```
classname.staticmembers
```

```
S.o.pln(marker.ht);
marker.write();
```

TO access non-static members:
[object reference syntax:

```
new classname().nonstatic members
```

```
S.o.pln(new matric().price;)
new marker().draw();
```

```
class Marker
{
  static int ht = 10;
      double price = 20.0;
  static void write()
     {
      S.o.println("Marker is used to write");
     }
void draw()
     {
      S.o.println("Marker is used to draw");
     }
}

class Mainclass1
{
  public static void main(String[] args)
  {
     System.out.println(Marker.ht);
     Marker.write();

     System.out.println(new Marker().price);
     new Marker().draw();
  }
}
}
```

O/P

```
10
Marker is used to write
20.0
Marker is used to draw
```

```
class box
{
  static int ht = 20;
     clouble price = 200;

  static void store()
    {
     system.out.println("box is used to store");
    }
}
  class mainclass
  {
   public static void main(String[]args)
   {
    System.out.println(box.ht);
    System.out.println(new box().price);
    Void.store();
   }
  }
}
```

```
class shoe
{
    static double price = 150.0;
    static int len = 10;

    void wear()
    {
      system.out.println("shoe is used to wear");
    }
```

```
class mainclass
{
 public static void main (String[]args)
 {
   System.out.println( shoe. price );
   System. out. println (shoe. leng) ;
   new shoe(). wear();
 }
}
```

```
class phone
{
  static double price = 2500.0;

  static void use()
   {
     system. out.println(" phone is used to use");
   }
class mainclass
 {
   public static void main (String[] args)
   {
     System.out.println ( phone. price );
     phone. use();
   }
}
```

```
Class program1
{
static int x = 10 ;
    int y = 20 ;
    double z = 33.33 ;
}

Class mainclass
{
    public static void main (String[]args)
    {
        System.out.println( program1.x) ;
        Program1.x = 100;
        System.out.println ("reass x value =" + program1.x) ;
        System.out.println ("y value = " + new program1().
                                                    y );
        System.out.print("z value = " + new program1().z) ;
        new program1.y = 200;
        new program1().z = 333.3 ;
        System.out.println ("reass y value =" + new program1().
        System.out.println ("reass z value =" + new program1()
                                                    .z ));
    }
}
```

y=20
z=33.33

y value = 200
z value = 333.33

y=20
z=33.33

y value = 200
z value = 333.33

y value = 200
z value = 333.3

x value = 10

read x value = 100

y value = 20

z value = 23.33

read y value = 20

read z value = 30.33

For non-static

Every time there is a new object created and store all non-static datas.

For static

only one memory variable is assigned

---

reference var = It is a special type of variable used to store obj of class.

REFFERENCE VARIABLE

```
class program1
{
static int x = 10;
     int y = 20;
     double z = 33.33;
}

class mainclass
{
  public static void main (String [] args)
  {
```

```
program1 p1 = new program1();
System.out.println("y value=" + p1.y);
System.out.println("z value=" + p1.z);
p1.y = 200;
p1.z = 333.33;
System.out.println("reassaigned y value=" + p1.y);
System.out.println("reass z value=" + p1.z);
System.out.println("-------");
program1 p2 = new program1();
System.out.println("y value=" + p2.y);
System.out.println("z value=" + p2.z);
}
}
```

o/p

```
y value = 20
z value = 33.33
reassigned y value = 200
reass z value = 333.33
-------
y value = 20
z value = 333.33
```

11-03-22

* object is a real-time entity which contains some states, and behaviours.

* state will represents properties of an object
behaviour will represents the functionalities of an object.

* The class is a definition block which is used to define states & behaviour of an object.

* class is like blue-print of an object, if we have a class, we can create 'n' number of objects.

* Inside the class we can define data members (variables) & function members (methods) are called as members of a class.

* Based on static keyword, the members are categorized into two types.
static members
non static (or) Instance members

Static Members:

* The static members must be define with static keyword.

* We can access static members by static reference syntax. (i.e) classname. static members

* We will have a single copy of data in the memory.

* static members will be loaded in the static pool area

* static members will be loaded at the time of class loading. Hence we will have a single copy of static members.

* static members are called as class-level members.

Non-Static Members:

* Non-static members should not be defined with static keyword. We can access non-static members, by creating object. i.e) new classname(). nonstatic members

* We will have multiple copies of non-static members.

* The non-static members will be loaded in the heap area.

* The non-static members will be loaded at the time of object creation, Hence we will have a multiple copies of non static members.

17/03/23 * Non-static members are called as object-level members.

Reference Variable:

It is a special type of variable, which is used to store, address of an object

```
className var1, var2, ...
var1 = new className()
```

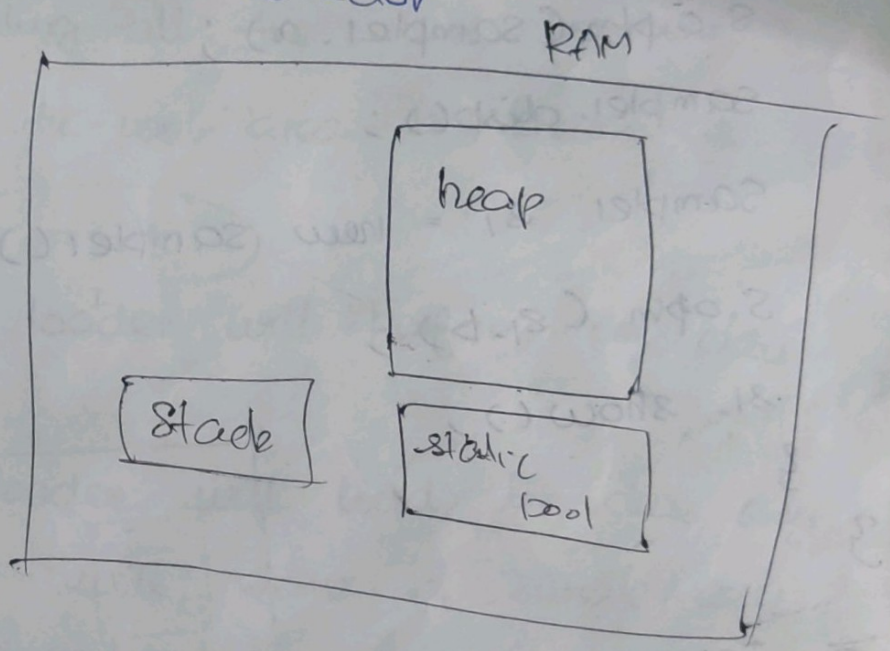
↳ an object is created and all nonstatic members are loaded

```
S.O.ftn (var1.nonstaticmember)
```

* The default value for the reference variable is null.

Task of JVM:

→ It allocates memory
→ It calls class loader
→ It calls main for execution
→ It calls garbage collector



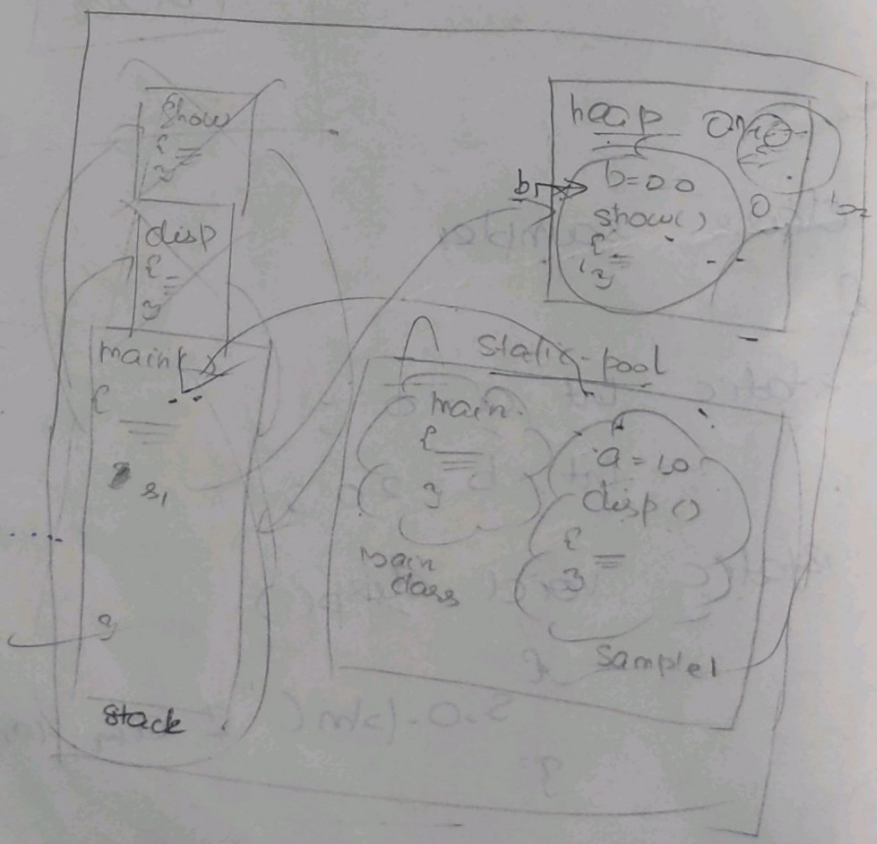

```
class sample
{
    static int a = 0;
           int b = 20;
    static void disp()
    {
        S.O.println("I am in disp() ..");
    }
```

```
void show()
{
 S.o.ph ("I am in show()....");
}
}

class Mainclass
{
  public static void main (String[] args)
  {
    S.o.p(m (Sample1.a);
    Sample1.disp();
    Sample1 s1 = new Sample1();
    S.o.p(m (s1.b);
    s1.show();
  }
}
```

a=10

⇒ Java Mainclass

```
10
I am in disp()...
20
I am in show()...
```



⇒ It allocates memory.

The JVM will divide the entire ram into three parts :

* heap area (It stores objects)
* static-pool area (It stores static-members)
* stack (use for execution & local variables will be stored)

* JVM calls the class loader

→ loading all the static members of a class, inside static-pool area is known as class loading.

→ class loader will perform class loading.

→ class loader will load the class only one time. Hence, we will have a single copy of static members.

* JVM calls main-method for execution.

* The JVM calls the garbage collector to clean the memory.

* It is a process of removing dereferenced objects (or) unreachable object from the heap area.

=> object present in the heap area without any reference variable is known as dereferenced object (or) unreachable object, There is no use of this, they will consume heap memory.

* The garbage collector can be called implicitly by the JVM at the end of the program.

* As a programmer, we can call the garbage collector explicitly by using System.gc();

* System is an built-in class in java and gc is a static method of System class

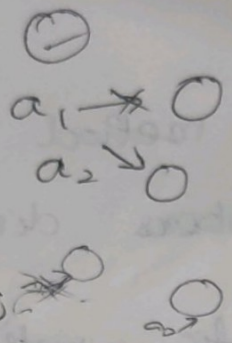
* When we call the garbage collector explicitly, It will internally calls finalize() method..

* Garbage collector uses mark and sweep algorithm

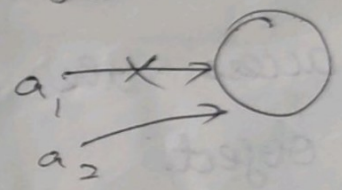
How to make our objects eligible for garbage collection?

* By removing the reference variables
* By assigning null to the reference variable.

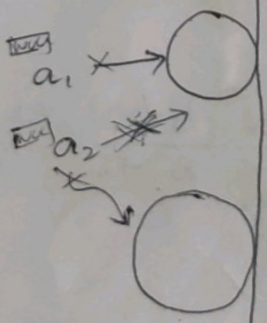
```
new A();
A a1 = new A();
A a2 = new A();
a1 = null;
System.gc
```



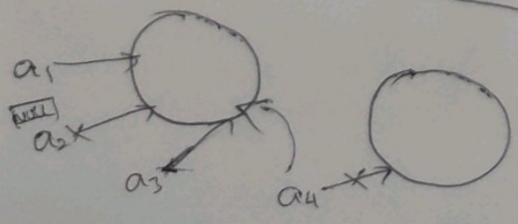

```
A a1 = new A();
A a2 = a1;
a1 = null;
System.gc
```

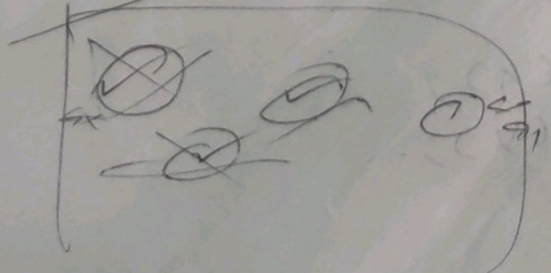


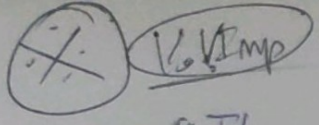

```
A1 a1 = new A();
A a2 = a1;
a2 = new A();
a1 = null
a2 = a1;
System.gc();
```




```
A a1 = new A();
A a2 = a1;
A a3 = a2;
A a4 = new A();
a4 = a2;
a2 = null;
a1 = a3;
```







*If you want to access the members outside the class, we must follow the syntax.

* If you want to access the members within the class, syntax is not mandatory.

(1.) A static method can access static members directly.

(2.) A non-static method can access both static & non-static members directly.

*From the static method, if you want to access the non-static members, we must create an object.

```
class A
{
 static int x = 10;
        int y = 20;
 static void m1()
      {
        S.o.pln(x);
      }
      void m2()
     {
      S.o.pln(x);
      S.o.pln(y);
      }
```

```
p.s. v.m()
{
    s.o.pln(x);
    m1();
    A a1 = new A();
s.o.pln(a1.y);
    a1.m2();
  }
}
```

Note:

The data members and function members can be static (or) non static not the local variables.

If it is local varial then neither static nor non-static.

It is in main.

Member ← class B { static int x=10; }

inside the class

```
p.s.v.m()
{
  static int x=10;
}
```

15.03.23

BLOCKS

```
class sample1
{
    static int a;
           int b;

    static
    {
        System.out.println("I am in static block... [SIB]");
        a=10;
    }
    {
        System.out.println("I am in notstatic block... [IIB]");
        b=20;
    }
}
class mainclass1
{
    Public static void main(String[] args)
    {
        System.out.println(sample1.a);
        System.out.println(sample1.a);
        - - - - -
        Sample1 s1 = new Sample1();
        System.out.println(s1.b);
        sample1 s2 = new Sample1();
        System.out.println(s2.b);
    }
}
```

Static Main called for static

This will execute automatically.

blocks are also called as INITIALIZERS

O/P

I am in static block -- [SIB]

10

10

I am in non static block [IIB]

20

I am in nonstatic block [IIB]

20

```
Class Sample2
{
   static int x = 10;
   int y = 20;

   static
     {
       System.out.println("I am in static block..1");
     }

   static
     {
       System.out.println("I am in static block..2")
     }

   {
     System.out.println("I am in non-static block...1");
   }

   {
     System.out.println("I am in non-static block...2");
   }
}
```

```
Class mainclass2
{
 static
 {
  S.O.pln("I am in mainclass static block");
 }
 {
  S.O.pln("I am in mainclass non-static block");
 }

public static void main(String[] args)
{
   S:O.pln(sample2.x);
   sample2 s2 = new sample2();
   sample2 s22 = new sample2();

mainclass2 m2 = new mainclass2();
 }
}
```

O/P: I am in mainclass static block

I am in static block 1
I am in static block 2
10
I am in not static block 1
I am in not static block 2
I am in non static block 1
I am in nom static block 2
I am in mainclass nonstatic block

block

The set of statements enclosed within the braces is known as blocks.

↓ at class level

```
class className
{
    Block { {
            ======
            } }

}
```

There are two types of blocks

- static block
- non-static (or) Instance block

STATIC Block:

* It must be defined with static key word.
* It will executes at the time of class loading.
* It will executes only one time.
* It is used to initialize static members of a class, Hence it is called as static initialization block. [SIB]

NON-STATIC(or)INSTANCE BLOCK:

* It should not be defined with static keyword.

* It will executes at the time of object creation.

* It will execute multiple times

* It is used to initialize instance members of the class, hence it is called as Instance Initialization block [IIB].

Note:

* blocks are called as initializers.

* In a class we can define 'n' no of block and they will execute in sequential order.

* In a class, if we have static block, non static blocks and main-method, the order of execution will be static block, main-method, non-static (when we create object).

drawbacks:

We can't call the blocks explicitly

We can't pass any arguments.

CONSTRUCTOR

* Constructor is a special member of a class, which is used to create and initialize the object.

* Its name must be same as class name, It should not have any return type and it can take arguments.

* For every java class, constructor is mandatory, If we are not writing the constructor, the compiler will defines the default constructor. [It initializes the data members to a default value].

* As a programmer, we can define two types of constructor: * Zero param (no-argument)
* Parameterized constructor

Syntax:

```
Class classname
{
  Classname (arguments)
  {
    == }. Initialization logic
  }
}
```

* In a class we can define "n" no. of constructors, but they should different arguments that is known as "CONSTRUCTOR OVERLOADING".

```
Class Mobile
{
   double price;
   int storage;
   String brand;

   Mobile (double arg1, int arg2, String arg3)
   {
    S.o.pln ("creating and Initializing Mobile object...");
    price = arg1;
    storage = arg2;
    brand = arg3;
   }
}
Class Mainclass1
{
   public static void main(String[] args)
   {
    Mobile m1 = new Mobile (40000.0, 8, "samsung");
    S.o.pln ("price of the Mobile=" + m1.price);
    S.o.pln ("storage of the mobile =" + m2.storge);
    S.o.pln (" brang of the mobile =" + m3.brand);
   }
}
```

O/P: Creating and Initialing mobile object.

Price of the mobile = 24000

Storage of the mobile = 8

brand of the mobile = samsung

Constructor overloading

```
class sample1
{
   int x;
   double y;

 sample1()
  {
    S.o.pln("I am in sample1...");
  }

 sample1(int arg1, double arg2)
  {
    x = arg1;
    y = arg2;
  }

 sample1 (int arg1)
  {
    x = arg1;
  }
}
```

```
class mainclass2
{
  public static void main (String[] args)
  {
    sample1 s1 = new sample1 ();
    S.O.pln ("x value =" + s1.x);
    S.O.pln ("y value =" + s1.y);

    sample1 s2 = new sample1(10);
      S.O.pln ("x value =" + s2.x);
      S.O.pln ("y value =" + s2.y);

    sample1 s3 = new sample1 (10, 22.22)
      S.O.pln ("x value =" + s3.x);
      S.O.pln ("y value =" + s3.y);
  }
}
```

I am in sample1
x value = 0
y value = 0.0

O/P

I am in sample 1
x value = 0
y value = 0.0
x value = 10
y value = 0.0
x value = 10
y value = 22.22 //

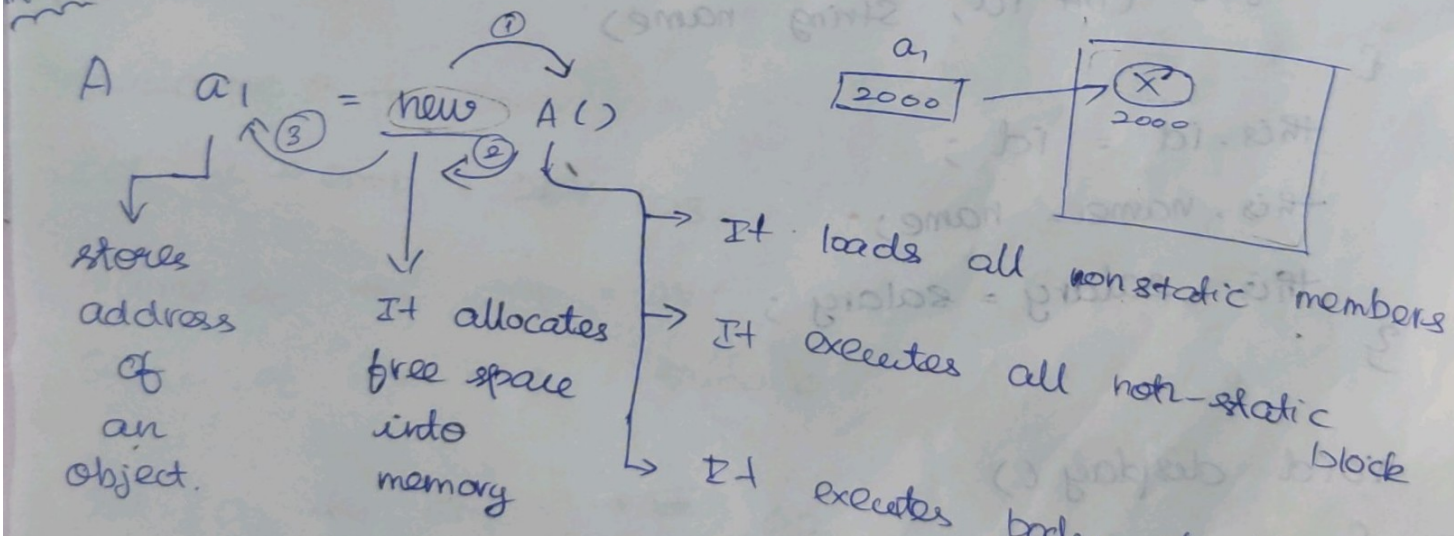
16.03.23

Constructor should not have any return type, If you write a return time, It becomes a method.

```
Class A
{
    void A()
    {
    }
}
```

It is never possible to define the constructor as static.

WHY?



Constructor will executes at the time of object creation, neither before nor after.

```
Class Employee
{
final static string companyName = "IOTIO";
final int id;
final String name;
  double salary;

  Employee (int id, String name, double salary)
  {
    this.id = id;
    this.name = name;
    this.salary = salary;
  }

  Employee (int id, String name)
  {
    this.id = id;
    this.name = name;
    this.salary = salary;
  }

  void display()
  {
    System.out.println("Id of the emp = " + id);
    System.out.println("Name of the emp = " + name);
    System.out.println("Salary of the emp = " + salary);
  }
}
```

```
Class EmployeeManagement
{
  public static void main (String [] args)
  {
    System.out.println ("Welcome to " + Employee.companyname);

    Employee Emp1 = new Employee (123, "DEV", 5000.0);
        Emp1.display ();

    Employee Emp2 = new Employee (02, "XAVI", 4000.0);
        Emp2.display ();
  }
}
```

O/P

Welcome to ZOHO

Id of the emp = 123

Name of the emp = DEV

Salary of the emp = 5000.0

Id of the emp = 02

Name of the Emp = XAVI

Salary of the emp = 4000.0

this is used to say accas one more time in local

7-03-

Constants in Java:

We can define constants in java by using "final" keyword. If we define any variable is final. It is never possible to reassign them.

There are three types of constants in java.

1. Global constant: If we define, any static variable with final keyword, then it is called global constant.

2. Object constant: If we define not static variable with final keyword, then it is object constant

3. Local constant: If we define a local variable with final keyword, then it is local constant.

Eg:

```
Class A
{
    final static int x = 10 ; / Global constant
    final int y = 20 ; / object "
    void m()
    {
        final int z = 30 ; // local "
    }
}
```

TYPES OF VARIABLES

1. STATIC -
2. NON-STATIC -
3. LOCAL -

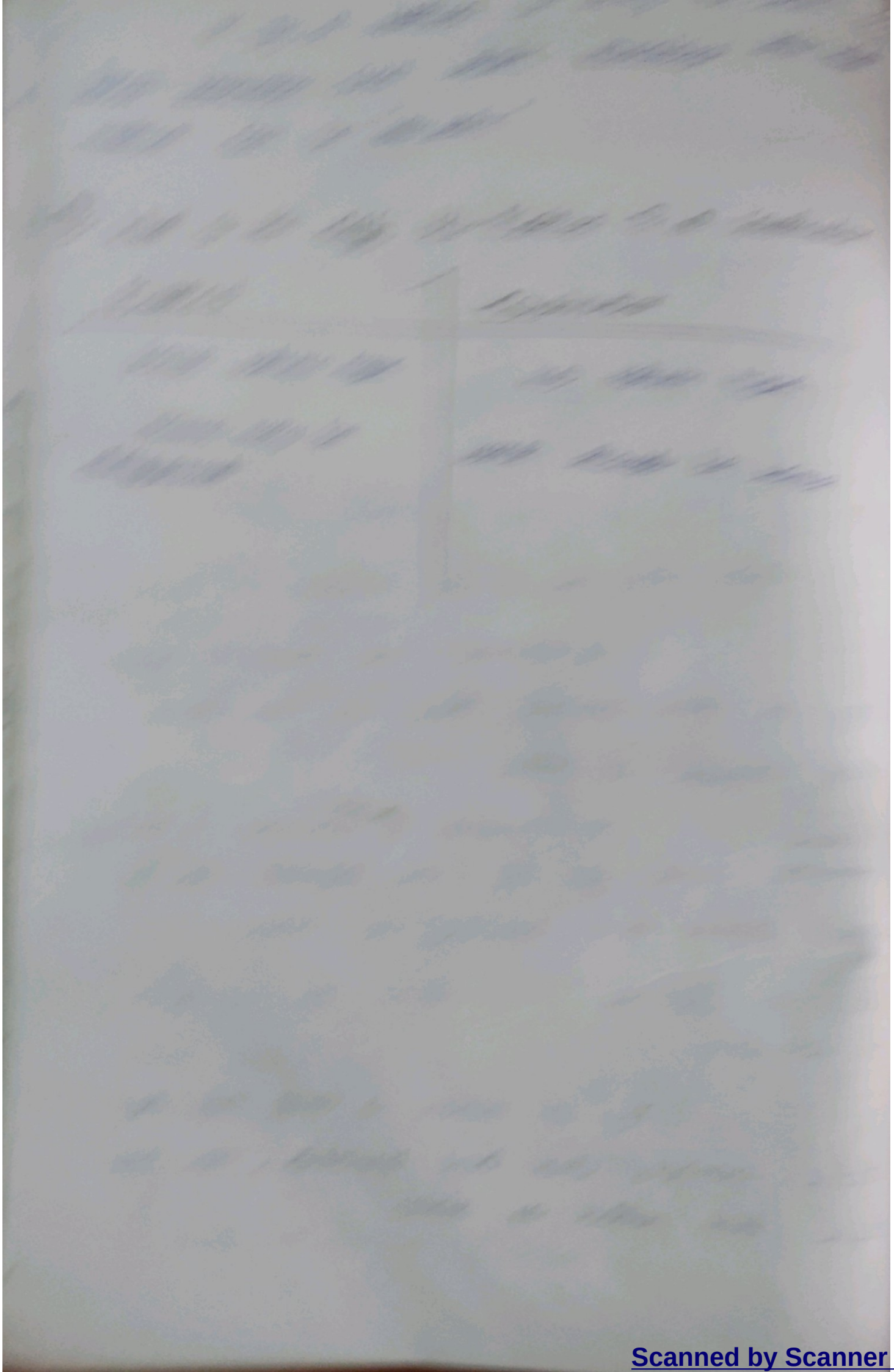
When to define a variable as static?

* Whenever we want a single copy of that variable, then we can define that variable as static.

* Whenever a variable is having same ~~common~~ data for all the objects, then we can define that variable as static.

How to decide a method can be static or non-static

If a method is using only the static members and local variables, then we define that method as static



* If a method is using non-static and local variable and static variables then that method can be "non-static".

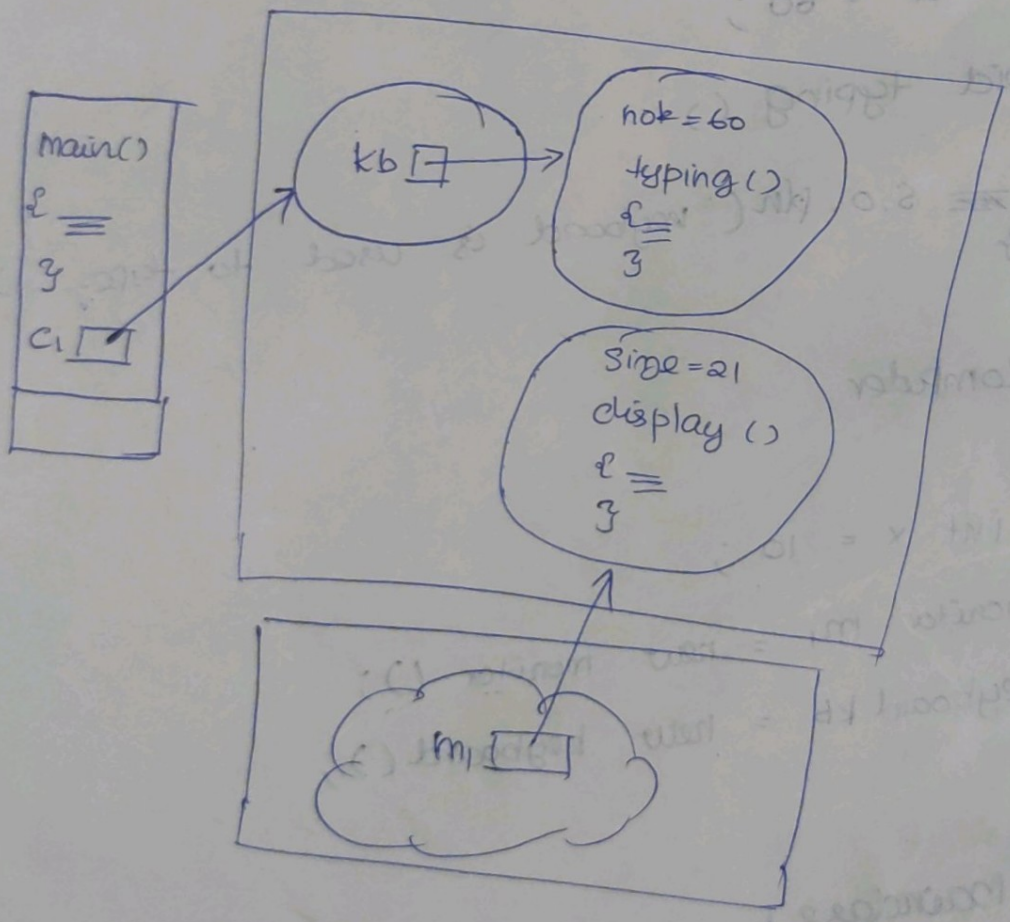
(3) What is the diff b/w Method & a constructor

Method	Constructor
have return type	no return type
name may be different	name should be same

Composition :

```
Class Monitor
{
    int size = 21;
    void display()
    {
        S.O.pln("monitor is used to display");
    }
}

Class keyboard
{
    int key = 60;
    void typing()
    {
        S.O.pln("keyboard is used to type...");
    }
}

class computer
{
    int x = 10;
static monitor m1 = new monitor();
    keyboard kb = new keyboard();
}

class Mainclass1
{
    public static void main(String[] args)
    {
        System.out.println("*****");
```

```
System.out.println("monitor size = " + computer.m1.size);
computer.m1.display();

Computer c1 = new Computer();
System.out.println("number of keys = " + c1.kb.nok);
c1.kb.typing();
System.out.println("*******");
}
```



O/P:

```
Monitor size = 21
monitor is used to display
Number of keys = 60
keyboard is used to type..
```

⇒ An object consisting of some other object is known as composition. (or)

⇒ A class which is defined with reference variable of some other class as its data members is known as composition

Java supports two types of composition

* static composition
* non-static (or) Instance composition

Static composition :

* In static composition, the reference variable must be defined with static keyword.

* We can access static composition by static reference (i.e classname. static reference variable. non static members)

* static reference variable will be loaded in the static pool area but the object is created in the heap area

* We will have a single copy of object creation for static composition, because it will happens at the time of class loading.

Non-static composition:

* In non-static composition, the reference variable should not be defined with static keyword.

* We can access non-static composition by creating the object.

* The non-static reference variable will be loaded in the heap area as a part of object.

* For each object creation, we will have a separate copy of non-static composition.

Composition.m1.display()

System.out.println()

- System: It is a built in class in java
- out: static reference of PointStream class present in System class.
- println(): It is a non-static overloaded method of PointStream class

Eg: Static composition & Method overloading

20/03/21 H.W

```
public class address
{
    int drNo;
    String street;
    String city;
    long pin;
    Address (int drNo, String street, String city,
                                        long pin)
    {
        this.drNo = drNo;
        this.street = street;
        this.city = city;
        this.pin = pin;
    }
}

public class Employee
{
    Static String companyName = "zoho";
    Static Address office = new Address (34, "Rajaji street",
                                "Chennai", 600044);
    int id;
    String ename;
    double salary;
    Address residance;
```

```
Employee (int id, String name, double salary,
                                    Address residence)
{
    this.id = id;
    this.name = name;
    this.salary = salary;
    this.residence = residance;
}
}

public class mainclass1
{
    public static void main (String [] args)
    {
     S.O.pln ("Welcome to" + Employee.companyName);
     S.O.pln ("_ _ _ _ _ Company address _ _ _ _ _");
     s.o.pln ("DrNo =" + Employee.office.drNo);
     S.O.pln ("Street =" + Employee.office.street);
     S.opln ("city =" + Employee.office.city);
     S.opln ("pincode =" + Employee.office.pin);
     S.opln ("_ _ _ _ _ _ _ _ _");
```

```
Employee empl = new Employee(123, 'jeffrin", 100000.0,
                new address(80,"venugopal", "dubai", 564541);
S.o.pln("--------");
S.o.pln(" Employee Id = ' + empl.Id);
S.o.pln(" Employee Name = "+emp.name);
S.o.pln(" Employee salary = " + empl.salary));
S.o.pln(" Employee address = drno : " + empl.residence
                                        .drno
  + ", " + empl.residence.street + "street,"
        + empl.residence.city + ", "
        + " PIN = " + empl.residence.pin);
}
```

O/P

```
---- compang address ------
DrNo = 34
street = Rajaji street
city = Chennai
Pincode = 600044
- - - - - - - - -
- - - - - - - - - -
Employee id =123
Employee Name = jefrin
Employee salary = 100000.0
Employee address = drno: 80, Venugopal street, dubai,
                   PIN = 56
```

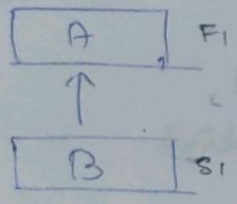
Inheritance :

* An object [class] is acquiring the properties of some other object [class] is known as Inheritance.

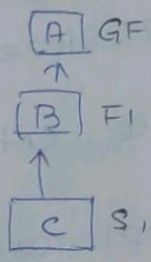
* We use inheritance to achieve code-reusability and extinsibility.

* There are five types of inheitance ; but java won't support multiple inheritance.

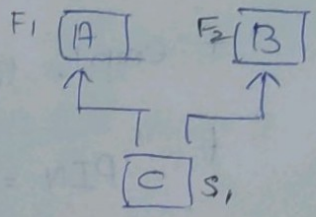
1. SINGLE LEVEL



2. MULTI-LEVEL

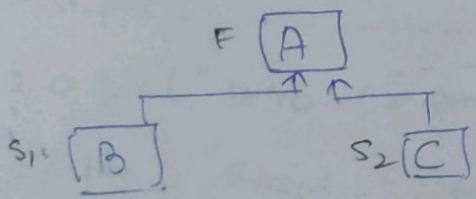


3. MULTIPLE



× java won't support

4. HIERARCHICAL



5. HYBRID [combination of other]

We can inherit from class to a class by "extends" keyword.

```
Class A
{
  int x = 10;
}
Class B extends A
{
  int y = 20;
}
public class Mainclass1
{
  public static void main (String [] args)
  {
    B b1 = new B();
    S.o.pln (b1.x);
    S.o.pln (b1.y);
  }
}
```

o/p
10
20

Super statement :

```
Class A
{
  A (int x)
  {
    S.o.pln ("I am in A (int)..." + x);
  }
Class B extends A
  {
    B (double d)
```

```
{
    super(10);
    S.O.pln("I am in B(double)..... "+d);
}

class C extends B
{
  C()
  {
     super(10.5)
     S.O.pln("I am in C()....");
  }
}

public class Mainclass1
{
   public static void main(String[] args)
   {
     C c1 = new C();
   }
}
```

O/p:

I am in A(int).....
I am in B(double).....
I am in C()....

21/03/23

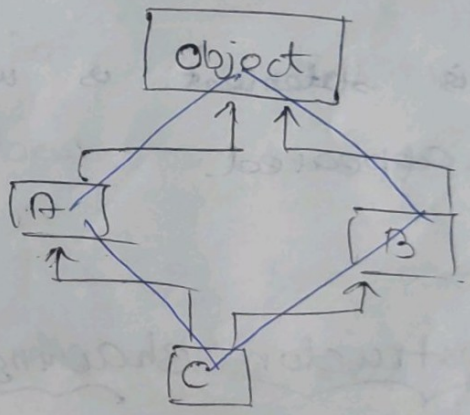
4. Why java won't support multiple inheritance?

1.) Syntax is not supported (extends only one class)

2.) Ambiguity of diamond problem

3. Multiple super() statement not allowed.

```
Class A extends object
{

}
Class B extends object
{

}
Class C extends A, B
{


}
```




```
Class D
{
  D()
  {
   this(6);
   System.out.println("I am in D()....");
  }

  D(int x)
  {
   this(10.5)
   System.out.println("I am in D(int)....");
  }
```

```
D( double x)
{
  System.out.println("I am in D(double)...");
}
}
public class Mainclass
{ public static void main(String[] args)
{
  D d1 = new D();
}
}
```

O/P:

I am in D(double)...

I am in D(int)....

I am in D()....

* This statement is used when constructor overloading is appeared.

Constructor chaining:

* calling a constructor from any other constructor is known as constructor chaining.

* We can achieve constructor chaining either by super statement (or) by this() statement.

* Both must be written inside the constructor body as their first line.

Super statement:

* It is used to call super class constructor from the sub class constructor.

* It must be written inside the constructor as their first line.

* If you are not writing the super statemond, the compiler will define the super statemond implicitly.

* if complier is defining the super statemond, we must have zero param constructor (or) default constructor in the super class

* if there is no zero param (or) default constructor in the supor class, we must write the super statement explicitly by passing arguments.

* TO write a super statement, inheritance is mandatory.

* For inheritance, super statement is mandatory.

This statement:

* This statement is used to call current class (or) same class constructor.

* It should be written inside the constructor as a first line.

* This statement should be written always explicitly.

* To write this statement constructor overloading is mandatory.

* In a class if we have, n constructors, we can write a maximum of (n-1) this statement (or else) we will get recursive constructor invocation.

Super & this keyword:

* Super keyword is used to access immediate super class members.

* This keyword is used to access current class (or) same class members.

* super and this keyword can be used anywhere except static context.

```
class A
{
  int x = 10;
}

class B extends A
{
   int x = 100;
   void test()
   {
     int x = 1000;
     System.out.println("local x value = "+x);
     System.out.println("current class x value = "
                                    + this.x);
     System.out.println("super class x value = "
                                    + super.x);
   }
}

Public class mainclass1
{
  public static void main(String[] args)
  {
    B b1 = new B();
    b1.test();
  }
}
```

o/p:

local x value = 1000

current class x value = 100

super class x value = 10

NOTE: * It is not possible to use super and this keyword inside main method.

Method Overriding:

```
class shoppingAppV1
{
   void payment()
   {
     System.out.println("payment is based on COD");
   }
}
class shoppingAppV2 extends ShoppingAppV1
{
   void payment()
   {
     System.out.println("payment is based on
                                        card");
   }
}
Public class Mainclass2
{
   public static void main(String[]args)
   {
     ShoppingAppV1 V1 = new shoppingAppV1();
     V1.payment();
```

```
ShoppingApp V2   V2 = new shoppingAppV2();
V2.payment();
}
}
```

o/p:

payment based on COD
payment based on card

* Inheriting the method and changing its implementation in the subclass according to the subclass specifications is known as method overriding.

* For method overriding inheritance is mandatory.

* While performing method overriding, we should not change the method signature (return type, method name, arguments), But we can change the body of the method.

* method overriding is used to achieve run-time polymorphism.

* When we want to perform a same type of task in a different way in the sub class, we can use method over riding.

The following method can't be overrided.

* private methods
* static methods
* final methods

Modifier	Inheritance	Overriding	Overloading	
			Inside	Subclass
private	×	×	✓	×
static	×	×	✓	×
final	✓	×	✓	✓

23/02/23

TYPE CASTING

PRIMITIVE TYPECASTING

WIDENING

NARROWING

CLASS/REFERENCE

UPCASTING

DOWN CASTING

Widening (Imp/Exp)

boolean	char	byte	short	int	long	float	double
True/false	16	8	16	32	64	32	64

Narrowing (Exp)

↓ It is never possible to convert

```
public class mainclass1
{
  public static void main(String[] args)
  {
    byte b = 28;
    // int x = (int)b // Explicit Widening
    // int x = b      // Implicit
    char ch = 'A';
    // int x = (int)ch // Explicit Widening
    // int x = ch;
    int i = 32;
    // double x = i;
    double d = 22.9999;
    int x = (int)d; // narrowing
    System.out.println(x);
  }
}
```

22

TYPE - CASTING.

Converting one type of data to any other datatype of data is known as type-casting.

There are two types of type casting.

1. PRIMITIVE TYPE CASTING
2. CLASS / REFERENCE CASTING.

1. PRIMITIVE:

Converting one primitive datatype to any other primitive data-type. is known as primitive type casting.

There are two types of primitive type casting.

Converting smaller datatype to any other larger-type is known as widening.

Widening can be done implicitly by the compiler or explicitly by the programmer, by using casting operator - (data type)

NARROWING.

Converting larger data-type to smaller data type is known as narrowing.

narrowing must be done always explicitly because it leads to precession loss.

NOTE
→

It is never possible to cast boolean datatype.

If datatypes are having same size, if the conversion is increasing the precession, then it is widening. (or else) narrowing.

The conversion from char to short and short to char is always explicit.

ASCII ⇒ American standard code for information interchange.

Java follows unicode standards,

A - 65 a - 97 ' ' (space) - 32
Z - 90 z - 122 '0' - 48

D - 68

```
Public class mainclass2
{
  public static void main (String [] args)
  {
    byte b = (byte) 257 ; // narrowing convarting
                          into to byte.
① S.o.pln (b);
② S.o.pln ((long)(Math.pow(2,8)/2-1));
   int x = 025 ; // octal
③ S.o.pln (x);
   int y = 0xAA // hexa decimal
④ S.o.pln (y);

   Char ch = 'A';
   S.o.pln (ch); ①
   S.o.pln (ch+1); ②
   S.o.pln (ch+ch); ③
   S.o.pln (ch++);
   S.o.pln (ch);
   S.o.pln (++ch);

   for (char c = 'A'; c<='z'; c++)
   {
     S.o.pln (c+" ");
   }
}
}
```

1
127
21
176
A ②
65 ②
130 ③
A
B
C
A B C .... Z

ch++ // A
ch // B
++ch // C

```
Char ch = 'A';
for (i=1; i<5 ; i++)
 {
  for (j=1; j≤ i ; j++ )
   {
     System.out.println (ch);
     ch++ ;
   }
}
```

```
class program6
{
public static void main (String [] args)
{
  int n=5 )
  for (int i=0 ; i<n ; i++ )
  {
    char ch = 'A';
    for ( int j=0 ; j<n ; j++)
    {
      if (i >=j )
      {
       System.out.print( ch++ + " ")
      }
      else {
         system.out.print (" ");
      }
    }
    System.out.println();
  }
}
```

A
A B
A B C
A B C D
A B C D E

① class programp

```
{
 public static void main (String[] args)
 {
   int n=5;
   for(int i=0; i<n; i++)
   {
     char ch='A';
     for(int j=0; j<n; j++)
     {
      if (i+j >= n-1)
      {
       System.out.println(ch++ +" ");
      }
      else
      {
       system.out.println(" ");
      }
     }
    System.out.println ();
 }
}
}
```

```
        A
      A B
    A B C
  A B C D
A B C D E
```

② class programp

```
{
 public static void main (String[] args)
 {
   int n=5;
   for (int i=0; i<n; i++)
   {
    char ch='A';
    for (int j=0; j<n; j++)
   {
    if (i+j >= n-1)
```

```
{
 System.out.println(ch++ +" ");
}
else {
     system.out.print(" ");
}
}
for (int j=1; j<n; j++)
{
  if (i>=j)
  {
   system.out.print(char)((char)ch-2)+" ");
    ch--;
  }
  else{
      System.out.print(" ");
  }
}
System.out.println();
ch++;
}
}
}
```

```
        A
      A B A
    A B C B A
  A B C D B A
A B C D E D C B A
```

③ class programp

```
{
 public static void main (String[] args)
 {
    int n=5;
    for (int i=0; i<n; i++)
    {
     char ch='A';
     for(int j=0; j<n; j++)
     {
      if (j+j >= n-1)
```

```
{
  System.out.print(ch+" ");
}
else
{
  System.out.print(" ");
}
ch++;
}
System.out.println();
}
}
}
```

```
    E
  D E
C D F
B C D E
A B C D E
```

② class program4

```
{
public static void main (String [] args)
{
  int n=5; ch k='A';
  for (int i=0; i<n; i++)
  {
    for (int j=0; j<n; j++)
    {
      if (i>=j)
      {
        System.out.print(k++ +" ");
      }
      else{
        System.out.print(" ");
      }
    }
    System.out.println();
    k++ j;
  }
}
}
```

```
A
B B
C C C
D D D D
E E E E E
```

⑤ class program5

```
{
public static void main (String[] args)
{
  int n=5; char ch='A';
  for (int i=0; i<n; i++)
  {
    for (int j=0; j<n; j++)
    {
      if (i>=j)
      {
        System.out.print(ch++ +" ");
      }
      else
      {
        System.out.print(" ");
      }
    }
    System.out.println();
  }
}
```

```
A
B C
D E F
G H I J
K L M N O
```

24
04.03.23

Class / Reference Casting

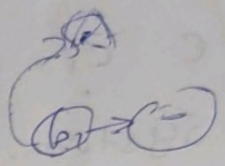
A a1 = (A) new B() // Explicit upcasting

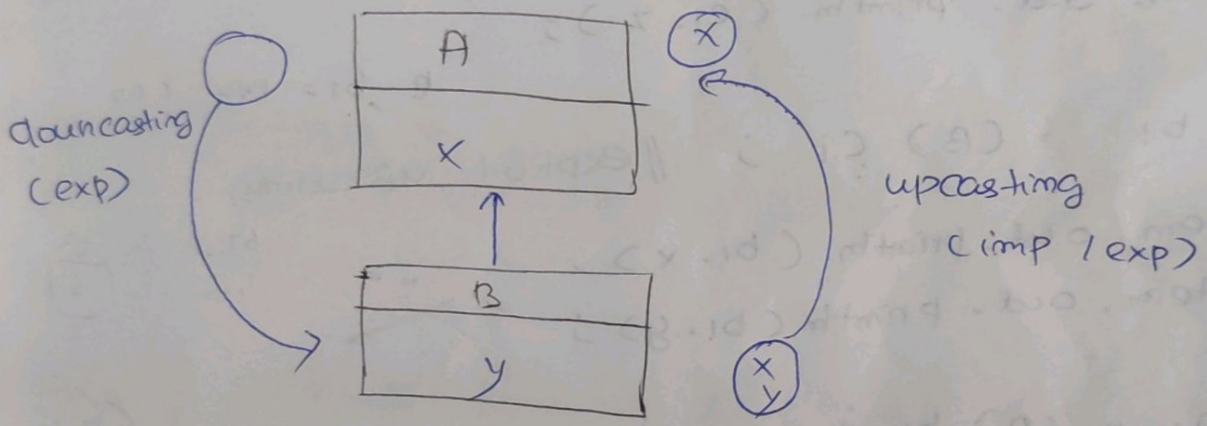
A a1 = new B() // implicit upcasting

B b1 = new B()

A a1 = (A) b1 // explicit upcasting

A a1 = b1 // implicit upcasting



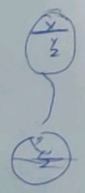


UPCASTING

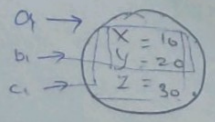
```
Class A
{
    int x = 10;
}
Class B extends A
{
    int y = 20;
}
Class C extends B
{
    int z = 30;
}
```

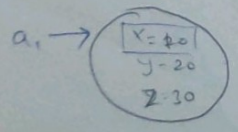
```
public class mainclass1
{
public static void main (String [] args)
{
// A a1 = new B(); (A) new B(); // Ap upcasting
// A a1 = new B() // implicit upcasting

C c1 = new C();
System.out.println (c1.x);
System.out.println (c1.y);
System.out.println (c1.z);

B b1 = new C()

B b1 = (B) c1 ; // explicit upcasting
System.out.println (b1.x);
System.out.println (b1.y);

A a1 = (A) b1 ;
System.out.println(a1.x);
}
}
```

b1.x
b1.y

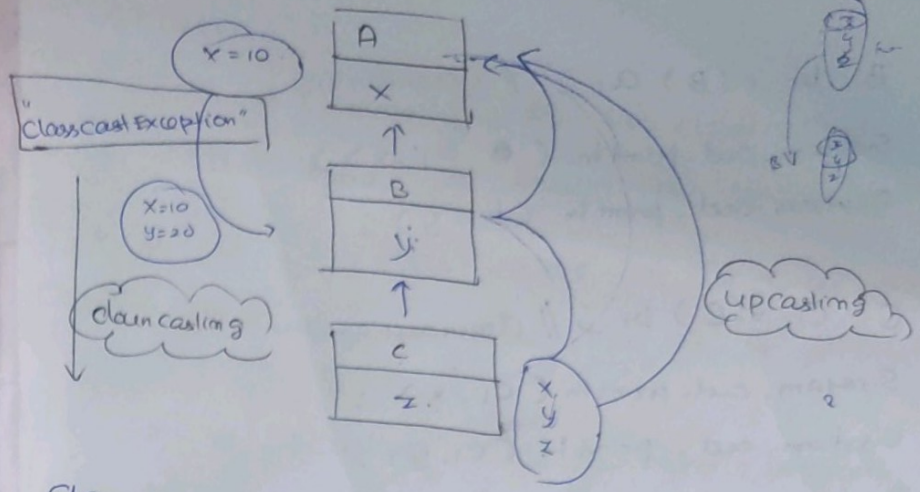


DOWN CASTING






```
C c1 = new C()
B b1 = c1
A a1 = b1
A a1 = new C();
```




```
Class A
{
    int x = 10 ;
}

Class B extends A
{
    int y = 20;
}

Class C extends B
{
    int z = 30;
}

public class mainclass1
{
  public static void main (String [] args)
  {
      A a1 = new C(); // upcasting
      System.out.println(a1.x);
```

```
B b1 = (B) a1 ; // downcasting
System.out.println (b1.x);
System.out.println (b1.y);

C c1 = (C) b1 ; // downcasting
System.out.println (c1.x);
System.out.println (c1.y);
System.out.println (c1.z);
}
}
```

op:

10
10
20
10
20
30

Class / reference casting

* Converting one object type to another object type is known as class / reference casting. /[class]

* TO achieve class casting, inheritance is mandatory.

* There are two types of class casting:

1. upcasting
2. downcasting.

UPCASTING:

* Converting the sub-class object into super class type is known as upcasting

(or)

* If a sub class object accessed by super class reference variable, then it is upcasting.

* Upcasting can be done implicitly by the compiler (or) explicitly by the programmer

* By using casting operator.

* once we upcast the object, it shows only super class properties and behaviours

DOWNCASTING:

* Converting the super class object into subclass object is known as downcasting.

* downcasting must be done always explicitly

* Only, An upcasted object can be downcasted (or else) we will get "ClassCastException".

* If the object is upcasted it shows only super class properties and behaviours, if you want the sub-class properties back, we must perform downcasting.

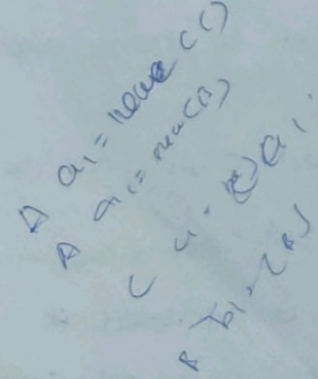
* Downcasting is like restoring of an object.

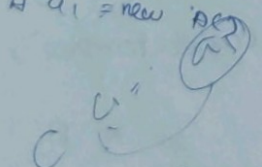
Note:

* If a method is argument is a super class type, we can pass same class objects (or) sub class objects, when we are passing the sub class object, those objects are implicitly upcasted to superclass type.

* Inside the method, these objects shows only super class properties and behaviours, if we want to restore the properties we must perform downcasting.

* Before doing downcasting, it is a better practice to check which object is present in reference variable, by using "instanceof" keyword.






```
class Executor
{
  void execute (A a1)
  {
    if (a1 instanceof C)
    {
      C c1 = (C) a1; // downcasting
      System.out.println (c1.x);
      System.out.println (c1.y);
      System.out.println (c1.z);
    }
    else if (a1 instanceof B)
    {
      B b1 = (B) a1; // downcasting
      System.out.println (b1.x);
      System.out.println (b1.y);
    }
    else
    {
      System.out.println (a1.x);
    }
  }
public class Mainclass1
{
  public static void main (String [] args)
```

A a1 = new C();

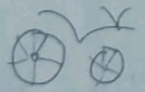
```
{
    Executor e1 = new Executor();
    e1.Execute (new A());
    e1.execute (new B()); // Upcasting
    e1.execute (new C()); // Upcasting
  }
}
```

Op
10
20
30
10
20
10

If a method return type is a super class type, we can return the same class object as well as subclass object, while returning the sub class objects those objects are implicitly upcasted to super class type.

(RUNTIME POLY) (or)

Method overriding with upcasting:



eg: BIKE

```
Class A
{
    void wish()
    {
        System.out.println("Hr...");
    }
}
```

```
Class B extends A
{
    void wish()
    {
        System.out.println("Bye....");
    }
}

Public class Mainclass1
{
    Public static void main (String[] args)
    {
        A a1 = new A();
        a1.wish();

        B b1 = new B();
        b1.wish();

        A a11 = new B();
        a11.wish();
    }
}
```

Op
Hr...
Bye...
Bye...

* Once, we overide the method even if we perform upcasting, overrided implementation will works.