OOPS - 7: Overriding var-arg method.

Overriding w.r to var-arg methods:

```
class P {
P v mi(int... x) {
 soplm("Parent method");
}
class C extends P {
  P v mi(int x) {
  soplm("child");
  }
}
```

```
class Test {
P s v m(String[] args) {
  P p = new P();
   p.mi(10); → Parent
C c = new C();
 c.mi(10); → child
P P1 = new C();
 P1.mi(10); → Parent
```

→ Above scenario is not overriding & it is overloading.

Ex:

```
class P {
 int x = 888;
}
class C extends P {
  int x = 999;
}
```

```
class Test {
 P s V m(String[] args){
 P p = new P();
  soplm(p.x); → 888
C c = new C();
  soplm(c.x); → 999
P P1 = new C();
  soplm(P1.x); → 888
}
}
```

→ overriding concept applicable only for methods but not for variables.

→ variable resolution always takes care by compiler based on reference type.

P → Non-static C → Non-static	P → static C → Non-static	P → Non-static C → static	P → static C → static
888 999 888	888 999 888	888 999 888	888 999 888

Difference b/w overloading & overriding

Property	overloading	overriding
① method names	must be same	must be same
② Argument types	must be different [Atleast order]	must be same [including order]
③ method signatures	must be different	must be same
④ Return types	No restrictions	must be same until 1.4V From 1.5V onwards co-varient return types allowed.
⑤ private, static, final methods	can be overloaded	cannot be overridden.
⑥ Access modifiers	No restrictions	The scope of access modifiers cannot be reduced but we can increase.
⑦ throws clause	No restrictions	→ if child class method throws any checked exception compulsory parent class method should throw the same checked exception or its parent.

⑧ method resolution	Always takes care by compiler based on reference type	→ But not restrictions ② for unchecked Always takes care by JVM based on Runtime object.
⑨ it is also known as	compile time polymorphism or static " (or) Early binding	Runtime Polymorphism (or) Dynamic " (or) late binding

* static & abstract modifiers are illegal combination. if any method contains a combination of these two modifiers then it will throw CE.

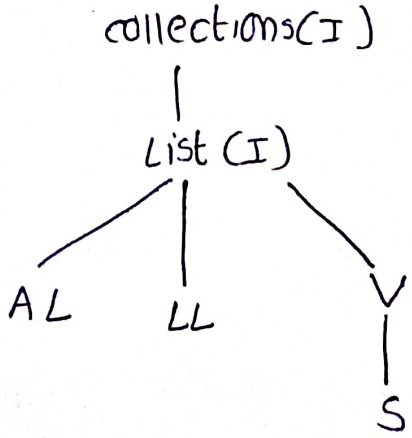
* Polymorphism *

overloading + overriding ⟶ polymorphism.

→ one name but multiple forms is the concept of polymorphism.

→ usage of parent reference to hold child object is the concept of polymorphism

```
List L = new ALC();
         new LLC();
         new stack();
         new Vector();
```



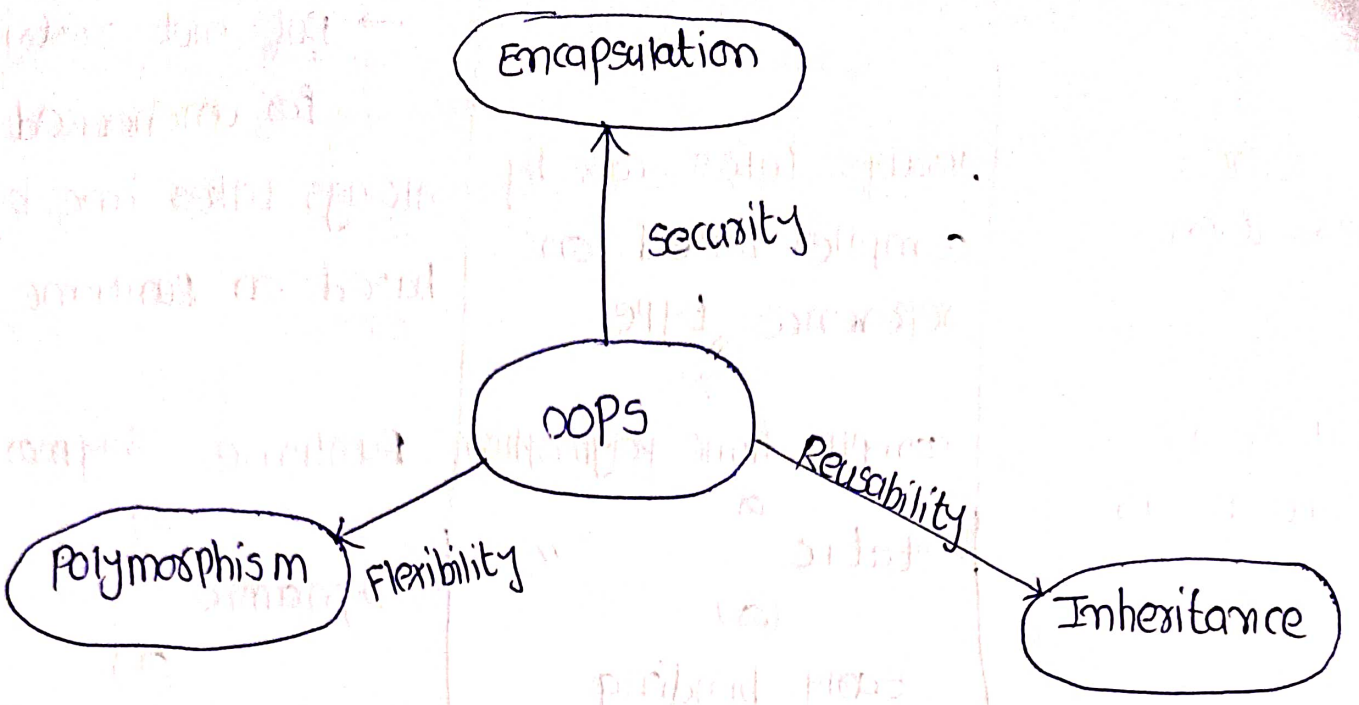


Fig: 3 pillars of oops

oops - 8 - coupling & object type casting.

*

```
class A                  | class B                  | class C
{                        | {                        | {
static int i = B.j;      | static int j = C.k;      | static int k = D.m1();
}                        | }                        | }
```

=

```
class D
{
public static int m1() {
return 10;
}
}
```

→ Above scenario is known as tightly coupled.

* object type casting *

Ex: 1

```
object o = new String("mk");
StringBuffer sb = (StringBuffer)o;
```

syntax:

```
A b = (c) d;
```

A = class/interface name

b = Name of reference variable

c = class/interface name.

d = Name of reference variable.

* condition-1 for object type casting: (@ compile time)

```
A b = (c) d;
```

→ 'd' and 'c' must have some relation.

→ Ex-1 satisfies condition-1 because object & stringBuffer have Parent - child relation.

* condition-2: (CE) * derived type = child.

```
A b = (c) d;
```

→ 'c' must be either same (or) derived type of 'A'.

→ Ex-1 satisfies condition-2 because 'c' & 'A' are of same type.

* condition-3: (Run time Error)

```
A b = (c) d;
```

→ underlying object type of 'd' must be either same or derived type of 'c'.

→ Ex-1: doesn't satisfies condition-3 because underlying object type of d (object) is string which is not same as 'c' (string buffer) (or) derived type of c (stringBuffer).

→ If condition-1 fails then it will report CE as

CE : inconvertible types.

found : d

required : C.

→ Ex: String s = new String("mk");

StringBuffer sb = (StringBuffer) s;

CE : inconvertible types.

Found : j.l.String.

Required : java.lang.StringBuffer.

→ if condition-2 fails then it will report CE as

CE : incompatible types

found : C

required : A.

Ex: Object o = new String("mk");

StringBuffer sb = (String) o;

CE : incompatible types

Found : java.lang.String

Required : java.lang.StringBuffer.

→ if condition-3 fails then it will report RE as

RE: class cast Exception.

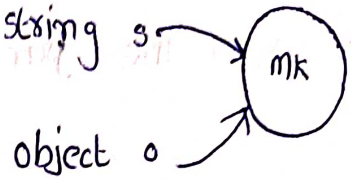
Ex: object o = new String("mk");

StringBuffer sb = (StringBuffer) o;

RE: class cast Exception : j.l. string cannot be cast to j.l. string Buffer.

OOPS - 9 - Type Casting

* Ex.: String s = new String ("mk");
object o = (object) s;



→ In the above example of type casting it is not creating any new objects. just we are providing another type of reference variable to the existing object.

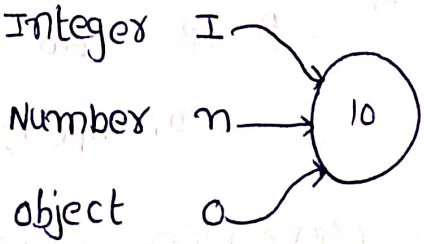
→ Above point can be written in short as below.

∴ object o = new String ("mk");

* Ex.: Integer I = new Integer (10); — ①
Number n = (Number) I; — ②
object o = (object) n; — ③

Verification:

soPln (I == n); → true
soPln (n == o); → true



→ ① & ② can be written as.

Number n = new Integer (10); — ④

→ ①, ② & ③ can be written as

Object o = new Integer(10); - ⑤

* → B is the child of A
→ C is " " " B
→ A is the grand parent of C.

→ C c = new c();

(B)c ⇒ B b = new c();

(A)(B)c ⇒ A a = new c();

* Ex:

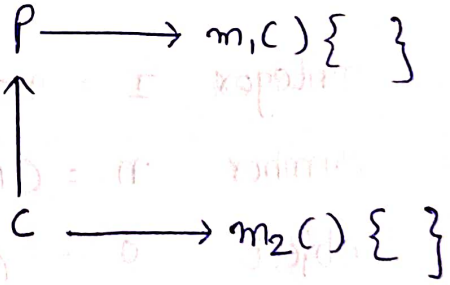
C c = new c();

* c.m1(); ✓
* c.m2(); ✓
* ((P)c).m1; ✓

⇒ P p = new c();
p.m1();

* ((P)c).m2; ✗

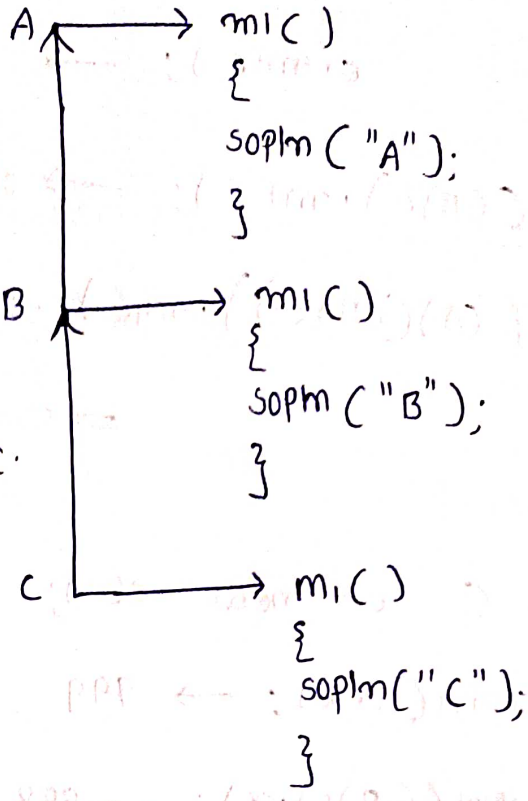
⇒ P p = new c();
p.m2();



* Note: Using parent reference can be used to hold child objects but by using that reference we can't call child specific method.

Ex:

```
C c = new C();
c.m1();  ⟶ o/p: C
((B)c).m1(); ⟶ o/p: C
(((A)(CB)c)).m1(); ⟶ o/p: C
```

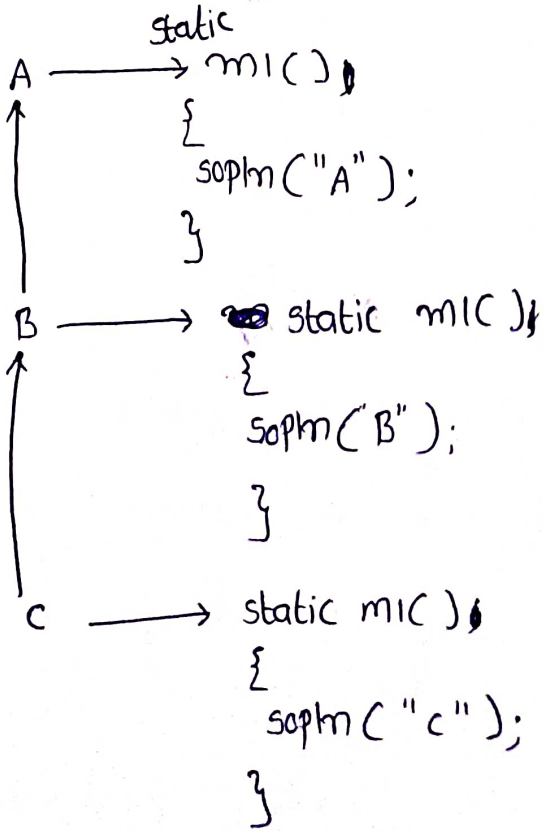


→ Above scenario is overriding.

* Note: In overriding, method resolution is always based on runtime object type. In the above example runtime object type is C.

Ex:

→ if all methods are static then it is method hiding. In method hiding, method resolution is always based on reference type. ~~In the below example reference types are different~~



C c = new cC);

c.m1C); → o/p - C [reference is 'c']

((B)c).m1 C); → o/p - B [reference is B]

((A)((B)c)).m1C); → o/p - A [reference is A]

=.=

Ex:

C c = new cC);

sopln (c.x); → 999

sopln (((B)c).x); → 888

sopln ((A((B)c)).x); → 777.

A → int x = 777;

↑

B → int x = 888;

↑

C → int x = 999;

* Note: variable resolution is always based on reference type.

OOPS - 10 - Static control flow

Ex: 1

```
class Base
{
    static int i = 10;                    (7)
    static
    {
        m1();                             (8)
        soPln("First static Block");      (10)
    }
    P S V main(String[] args)             (3)
    {
        m1();                             (13)
        soPln("main method");             (15)
    }
    P S V m1()                            (4)
    {
        soPln(j);                         (9), (14)
    }
    static                                (5)
    {
        soPln("second static block");     (11)
    }
    static int j = 20;                    (6)  (12)
}
```

i = 0 [RIWO] (1)
j = 0 [RIWO] (2)
i = 10 [R&W]
j = 20 [R&W]

→ This static member is a variable. if it is a variable JVM will assign default value (0) at the time of identification.

* RIWO = read indirectly write only.

java Base ↵

0
First static Block
second static Block.
20
main method.

Explanation: Java Base ↵

① Identification of static members from top to bottom.

② Execution of static variable assignments & static blocks from top to bottom.

③ Execution of main method.

only for static control flow.

→ ① - ⑥ in Ex-1 are identified as a part of Explanation condition - ①

→ ⑦ - ⑫ in Ex-1 are identified as a part of Explanation condition - ②

→ ⑬ - ⑮ in Ex-1 are identified as a part of Explanation condition - ③

* EX-2:

```
class Base {
static int i = 10;
static {
    m1();  ⇒ if we call m1() method then it is calling 'j' i.e indirect read.
    sopln(i); ⇒ Direct Read ⇒ Directly we are reading a value.
}
p s v m1() {
    sopln(j);  ⇒ Indirect read.
}
```

Ex: 3

```
class Test
{
 static int i = 10;
 static
 {
  m1();
  soplm(i); → Direct read.
 }
 p s v m1() {
  soplm(i); → Indirect read.
 }
}
```

i = 0 [RIWO]

CE: illegal forward reference.

→ Inside static block if we're trying to read a variable that read operation is called Direct read.

→ if we're calling a method & within that method if we're trying to read a variable that read operation is called Indirect read.

Note if a variable is just identified by the JVM & original value is not yet assigned then the variable is said to be in Read indirectly & write only state [RIWO].

* if a variable is in RIWO state then ~~that read operati~~ we can't perform direct read. But we can perform indirect read.

* if we're trying to read directly then we'll get CE saying "Illegal forward reference".

OOPS - II - Static Block.

→ static block will be executed at the time of class loading. Hence at the time of class loading if we want to perform any activity we have to define that inside static block.

Ex-:1

```
class Test {
  static {
    System.loadLibrary("native library paths");
  }
}
```

Ex:2

```
class Test {
  static {
    sopln("Hello Iam Print");
    system.exit(0);
  }
}
```

o/p: Hello Iam Print

Ex:3

```
class Test {
  static int x = m1();
  public static int m1()
  {
    System.out.println("Hello Iam Print");
    System.exit(0);
    return 10;
  }
}
```

Ex: 4

```
class Test
{
    static Test t = new Test();
    {
        soplm("Hello I can print");
        System.exit(0);
    }
}
```

* Note: whenever we are creating object, instance block will be executed & constructor will be executed.

Ex: 5

```
class Test {
    static Test t = new Test();
    Test()
    {
        soplm("Hello I can print");
        System.exit(0);
    }
}
```

Ex: 6

```
class Base {
0   static int i=10;                    (12)
②   static {
        m1();                           (13)
        soplm("Base static Block");     (15)
    }
③   P S V m(String[] args) {
        m1();
        soplm("Base main");
    }
④   P S V m1() {
        soplm(j);                       (14)
    }
⑤   static int j=20;                    (16)
}
```

```
class Derived extends Base {
⑥   static int x=100;                   (17)
⑦   static {
        m2();                           (18)
        soplm("Derived First static Block");  (20)
    }
⑧   P S V m(String[] args) {
        m2();                           (23)
        soplm("Derived main");          (25)
    }
⑨   P S V m2() {
        soplm(y);                       (19) (24)
    }
⑩   static {
        soplm("Derived second static Block");  (21)
    }
⑪   static int y=200;                   (22)
}
```

→ javac Base.java

Base.class Derived class

① Identification of static members from Parent to child [① - ⑪]

② Execution of static variable assignments & static blocks from Parent to child [⑫ - ㉒]

③ Execution of only child class main method. [㉓ - ㉕]

Java Derived ↵

i=0 [RIWO]
j=0 [RIWO]
x=0 [RIWO]
y=0 [RIWO]
i=10 [R&W]
j=20 [R&W]
x=100 [R&W]
y=200 [R&W]

O/P :

0
Base static Block
0
Derived First static Block.
" Second " "
200
Derived main

Java Base ↵

0
Base static Block
20
Base main.

OOPS-12- Instance control flow

EX-1:

```
class Test {
   int i = 10;
   {
      m1();
      soPln("First Instance Block");
   }
 Test
 {
 soPln("constructor");
 }
p s v m(String[] args) {
 soPln("main");
}
 Public void m1()
 {
 soPln(j);
 }
 {
 soPln("second Instance block");
 }
 int j = 20;
}
```

Java Test ↵

o/p :- main.

* Note: whenever we execute java class first ~~thing~~ static control flow will be verified.

* Instance members will be verified whenever we are creating an object.

Ex-2:

```
class Test {
③ int i = 10; ⑨
④ {
    m1(); ⑩
    soplm("First Instance block"); ⑫
  }
⑤ Test() { ⑬
  soplm("constructor");
  }
① P S V m(String[] args)
  {
② Test t = new Test();
  soplm("main"); ⑯
  }
⑥ Public void m1()
  {
  soplm(j); ⑪
  }
⑦ {
  soplm("second Instance Block"); ⑬
  }
⑧ int j = 20; ⑭
  }
```

[when object is created]

Java Test ←

① Identification of instance members from top to bottom [③-⑧]

② Execution of instance variable assignments & instance blocks from top to bottom. [⑨-⑭]

③ Execution of constructor [⑤]

i=0 [RIW0]
j=0 "
i=10[R&W]
j=20 [R&W]

o/p:

0
First Instance Block
second " "
constructor
main.

* Note: whenever we are executing a java class, first static control flow will be executed. In the static control flow, if we're creating an object. For every object creation the following sequence (above ①, ② & ③) will be performed.

* Note: → Static control flow is one time activity which will be performed at the time of class loading but instance control flow is not one time activity & it will be performed for every object creation.

→ object creation is the most costly operation. if there is no specific requirement then it is not recomended to create object.

* Instance control flow in parent to child relationship:-

Ex:1

```
class Parent
{
④ int i=10; ⑮
⑤ {
    m1c(); ⑯
    soplm("pIB"); ⑱
  }
⑥ Parent()
  {
   soplm("Parent constructor"); ⑳
  }
① p s v main(String[] args)
  {
    Parent p = new Parent();
    soplm("Parent main");
  }
⑦ Public void m1c()
  {
   soplm(j); ⑰
  }
⑧ int j=20; ⑲
}
```

```
class child extends Parent
{
⑨ int x=100; ㉑
⑩ {
   m2c(); ㉒
   soplm("CFIB"); ㉔
  }
⑪ child()
  {
   soplm("child constructor"); ㉗
  }
② p s v m(String[] args)
  {
③ C c = new CC();
   soplm("child main"); ㉘
  }
⑫ Public void m2c()
  {
   soplm(y); ㉓
  }
⑬ {
   soplm("CSIB"); ㉕
  }
⑭ int y=200; ㉖
}
```

Javac Parent.java

Parent.class child.class

① Identification of instance members from parent to child [④ - ⑭]

② Execution of instance variable assignments & instance blocks only in Parent class. [⑮ - ⑲]

③ Execution of Parent constructor. ⑳

④ Execution of instance variable assignments & instance blocks in child class. [㉑ - ㉖]

⑤ Execution of child constructor.

Java child ↵

i=0 [RIWO]	O
j=0 "	PIB
x=0 "	Parent constructor.
y=0 "	O
i=10 [RIW]	CFIB
j=20 [RIW]	CSIB
x=100 "	child constructor
y=200 "	child main

OOPS - 13 - Instance, static control flow.

Ex:

```
class Test {
 {
 soPln("FIB");
 }
 static
 {
 soPln("FSB");
 }
 Test()
 {
 soPln("constructor");
 }
 P s v m(String[] args)
 {
   Test t1 = new Test();
    soPln("main");
   Test t2 = new Test();
 }
 static
 {
 soPln("SSB");
 }
 {
 soPln("SIB");
 }
}
```

o/p:

```
FSB
SSB
FIB
SIB
constructor
main
FIB
SIB
constructor
```

Ex-2

```
Public class Initialization
{
 private static String m1( String msg )
 {
  sopln( "msg" );
  return msg;
 }
 Public Initialization( )
 {
  m = m1 ( " 1 " );
 }
 {
  m = m1 ("2");
 }
 String m = m1("3");
 P S V m ( string[] args )
 {
  object o = new Initialization();
 }
}
```

o/p:

2
3
1

m = null
2
3
1

Ex:3

```
Public class Initialization2
{
 Private static String m1( String msg )
 {
  sopln (msg);
  return msg;
 }
 Static String m = m1 ("1");
 {
   m = m1 ("2");
 }
 static
  {
   m = m1 ("3");
  }
```

o/p:

1
3
2

m = null
1
3
2

```
p s v m ( String [] args)
{
  object obj = new Initialization ();
 }
}
```

EX-4:

```
class Test
{
  int x = 10;
  p s v m ( String [] args )
  {
   soPln (x);  — ①
  }
}
```

o/p :-

Non-static variable x cannot be referenced from a static content.

*Note: From static area we can't access instance members.

```
Test t = new Test ();
soPln ( t.x);
```

— ②

if we replace ② in place of ① then o/p will be '10'

Notes

→ In how many we can create an object in Java (or) In how many ways we can get an object in java.

(i) By using new operator :- Test t = new Test ();

(ii) By using newInstance( ) method :- Test t = (Test)Class.forName("Test").newInstance();

(iii) By using factory method :- Runtime r = Runtime.getRuntime();

DateFormat df = DateFormat.getInstance ();

(iv) By using clone() method :-

Test t_1 = new Test (); | Test t_2 = (Test) t_1.clone ();

(v) By using deserialization : ObjectInputStream ois = ne

(v) By using De serialization.

```
FileInputStream fis = new FIS("abc.ser");
ObjectInputStream ois = new OIS(fis);
Dog d2 = (Dog)ois.readObject();
```

OOPS - 14 - Constructor

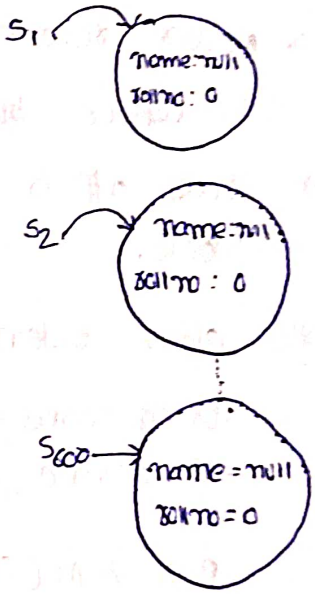
→ Constructor job is to initialize an object but not to create an object.

→ For Instance variables, for every object a seperate copy will be created.

→ Default values of string & int instance variables are null & 0.

Ex:
```
class Student {
    String name;   — ①
    int rollno;    — ②

    P S V M(String[] args)
    {
     Student s1 = new Student();
     Student s2 = new Student();
       :
     Student s600 = new Student();
    }
}
```



***

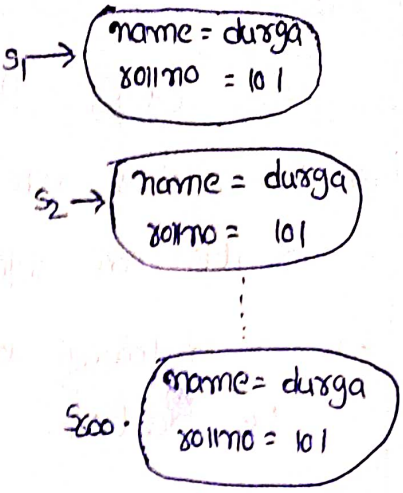
→ once we create an object compulsory we should perform initialization then only that object in a position to respond properly.

→ In the above Ex: all values are null & 0 which is meaningless.

→ Never recommended to perform initialization at the time of declaration i.e string name = "durga"; } in place of ① & int rollno = 101; ②

Ex:

```
class student {
    string name;
    int rollno;
    {
      name = "durga";
      rollno = 101;
    }
    p s v m(String[] args)
    {
     Student s1 = new student();
     Student s2 = new student();
        :
    }
}
```



→ Never recommended to perform initialization for object inside instance block.

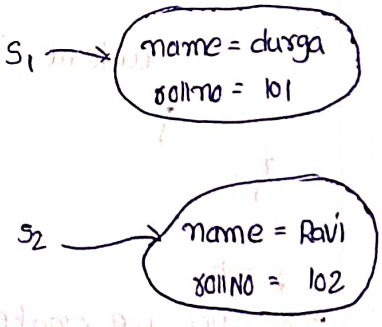
→ Above o/p is useless because we initiated objects inside instance block.

Ex:

```
Class student {
    string name;
    int rollno;

    p s v m (String[] args)
    {
     Student s1 = new student();
      s1.name = "durga";
      s1.rollno = 101;
     Student s2 = new student();
      s2.name = "Ravi";
      s2.rollno = 102;
    }
}
```

o/p



→ Above process is not recommended even it will maintain different values. Disadvantage is length of the code increases.

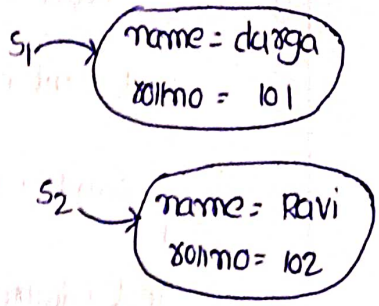
constructor:

Ex:

```
class student {
    String name;
    int rollno;
    Student( String name, int rollno )
    {
        this.name = name;
        this.rollno = rollno;
    }
    P s v m(String[] args)
    {
    Student s1 = new student ("durga", 101);
    student s2 = new student ("Ravi", 102);
        :
    }
}
```

(constructor)

O/P:



→ constructor is the best choice to initialize an object.

→ whenever we are creating an object some piece of the code will be executed automatically to perform initialization of the object. This piece of the code is nothing but constructor. Hence the main purpose of the constructor is to perform initialization of an object.

* Diff b/w constructor & Instance block *

→ Other than initialization if we want to perform any activity for every object creation then we should go for Instance block.

→ Both constructor & I. Block will be executed for every object creation but instance block first followed by constructor.

Ex: class Test {

static int count = 0;

Test ( )
{ this(10);
count ++;
}
Test (int i)
{
count ++;
}
Test (double d)
{
count ++;
}
p s v m (String[] args)
{
Test t1 = new Test ( );
" t2 = " " (10);
" t3 = " " (10.5);
}
}

→ To get the value for no. of objects got executed, if we write this code without this(10) then it will give count value. But by mistake if this(10) is available then it should increment twice & won't give exact count.

→ In order to resolve the issue we will use Instance Block.

Ex: class Test {

static int count=0;
{
count ++;
}
Test ( )
{
}
Test (int i)
{
}
Test (double d)
{
}

p s v m (String[] args)
{
Test t1 = new Test ( );
Test t2 = new Test(10);
" t3 = " " (10.5);

soplnC" The no. of objects created"
+ count);
}
}

O/P :- 3

* Rules of writing constructor *

1. Name of the class & name of the constructor must be matched.
2. Return type concept not applicable for constructor even void also.

Ex:
```
class Test {
    void Test()
    {
    }
}
```
it is a method but not constructor

→ it is possible to declare a method with same name as class but it is not recommended.

***

3. The only applicable modifiers for constructor are:

(a) public (b) private (c) protected (d) default.

Ex:
```
class Test {
    static Test()
    {
    }
}
```
CE: modifier static not allowed here.

→ compiler is responsible to generate default constructor iff and only if we are not writing any constructor.

→ Every class in java will contain constructor. it may be default construct generated by compiler or customized constructor explicitly provided by programmer but not both simultaneously.

OOPS - 15 - Default constructor.

prototype of Default constructor:

```
class Test {
Test() {
} super();
}
```

① Default constructor is no-argument constructor

② Access modifier of the default constructor is same as class modifier (applicable only for public & default)

→ private & protected access modifier we cannot apply for top-level class.

③ Default constructor contains only one line 'super();' ∴ it is a no-argument call to super class constructor.

programmer's code (level-1)	compiler generated code (level-1)

class Test
{
}
	class Test
	{
	Test()
	{
	super();
	}
	}

public class Test
{
}
	public class Test
	{
	public Test()
	{
	super();
	}
	}

public class Test
{
void Test()
{
}
}
	public class Test {
	public Test() {
	super();
	}
	void Test() {
	}
	}

programmer's code(level-2)	compiler generated code (level-

class Test {
Test()
{
}
}
* First line inside every constructor should be either super() or this(). if we didn't provide anything then compiler will provide super by default.
	class Test {
	Test()
	{
	super();
	}
	}

class Test {
Test(int i)
{
super();
}
}
	class Test {
	Test(int i)
	{
	super();
	}
	}

class Test {
Test() {
this(10);
}
Test(int i) {
}
}
	class Test {
	Test() {
	this(10);
	}
	Test(int i) {
	super();
	}
	}

* case-1 *

```
class Test {
Test()
 {
  sopln("constructor");
   super();
 }
}
```

CE: call to super must be first statement in constructor

* case-2 *

```
class Test{
  Test()
 {
   super();
   this();
 }
}
```

CE: call to this must be first statement in constructor.

→ we can't take super() & this() in constructor simultaneously.

* case-3 *

```
class Test {
 Public void m1()
 {
   super();
   sopln("Hello");
 }
}
```

CE: call to super must be first statement in constructor.

→ we can take super() & this() only inside a constructor. if we are taking outside of constructor it will give CE:

→ we can call a constructor directly from another constructor only.

* super() & this() *

→ we can use only in constructors.
→ only in first line
→ only one but not both simultaneously.

super(), this()	super, this

* constructor calls to call super class & current class constructors.	class P {
	int x = 100;
	}
	class C extends P
	{
	int x = 200;
	Public void m1() {
	sopln(this.x); 200
	sopln(super.x); 100
	};
	}
	* These are keywords to refer super class & current class instance members.

② * we can use only in constructor as first line	② class Test {
	p s v m (String[] args)
	{
	sopln(super.hashcode());
	}
	}
	CE: non-static variable super cannot be referenced from a static context.
	* we can use anywhere except static area

③ we can use only once in constructor	③ we can use any no. of times

OOPS - 16 : Overloaded constructor

Ex:

```
class Test {
Test () {
 this(10);
 sopln("no-arg");
}
Test (int i) {
 this (10.5);
 sopln(" Int - arg");
}
 Test (double d) {
  sopln ("double-arg");
}
```

} Overloaded constructors.

```
p s v m( string [] args) {
Test t1 = new Test ();
Test t2 = new Test (10);
Test t3 = new Test (10.5);
Test t4 = new Test (10L)
```

o/p: double-arg
int - arg
no - arg

o/p: double-arg
int - arg

o/p: double- arg

o/p: double-arg.

Ex:

```
class person {
    :
person ( string name)
{
 this. name =name;
 }
```

Person P1 = new person ("durga");

```
Person ( String name, int age)
{
  this. name = name;
  this. age = age;
}
```

Person P_2 = new Person("Ravi", 30);

```
Person( ..... ) {
    :
    :
    :
}
```

Person P_3 = new Person("Ravi", 30, "Siva", "XYZ");

—o—

Ex:

```
class P {
  m1() {
  }
}
class C extends P {
   m2() {
   }
}
C c = new C();
  c.m1(); ✓
  c.m2(); ✓
```

```
class P {
   P() {
   }
}
class C extends P
   C(int i)
   {
   super();
   }
}
  C  c= new C();
```

* Inheritance concept applicable for methods.
* " " Not " " constructors. if you want you can call parent constructor.
* overriding concept is NA for constructors.
* Every class in java including abstract class contains constructor but interface cannot contain constructor.

```
class Test {
  Test()
  {
  }
}
```
✓

```
abstract class Test {
  Test() {
  }
}
```
✓

```
interface Test {
  Test(){
  }
}
```
✗

→ constructor job is to perform initialization i,e to perform initialization for instance variable. But every variable present in interface is static variable & there is no chance of instance variable in interface.

Ex:
```
class Test {
    p s v m1()
    {
      m2();
    }
    p s v m2()
    {
      m1();
    }
    p s v m(String[] args)
    {
        m1();
        soplm("Hello");
    }
}
```
O/P: RE: stackoverflow error.

Ex:
```
class Test {
    Test() {
    this(10);
    }
    Test(int i) {
     this();
    }
    p s v m(String[] args) {
    soplm("Hello");
}  }
```
O/P: CE: Recursive constructor Invocation.

Ex:
```
class P {
P(){
super();
}
class C extends P
{
C(){
super();
}
}
```
✓

```
class P {
P(){
}
class C extends P
{
}
```
→ super();

```
C(){
super();
}
```
✓

```
class P {
P(int i){
}
}
class C extends P
{
}
```
→ super();

```
C(){
super();
}
```
✗

CE: cannot find symbol
symbol: constructor P()
location: class P

→ if Parent class contains any argument constructors then while writing child classes we have to take special care w.r.t constructors.

**
→ whenever we are writing any argument constructor it is highly recommended to write no-arg constructor also.

Ex:
```
m1()
{
  try
  {
  m2();
  }
  catch (IOException e)
  {
  }
}
```

```
m2() throws IOException
{
}
```
checked exception

(or)

```
m1() throws IOException
{ m2();
}
```

Ex:
```
class P
{
  P() throws IOException
  {
  }
}
class C extends P
{
  .....
}
```

```
C()
{
 super();
}
```

CE: unreported exception java.io.IOException in default constructor.

Ex:
```
class P {
    P() throws IOException
    {
    }
}
class C extends P
    C() {
     try
     {
     super();
     }
     catch(IOException e)
      {
      }
    }
}
```

→ This example is wrong because first line in every constructor is super() or this(). But in this example 1st line is "try".

Ex:
```
class P {
    P() throws IOException {
    }
}
class C extends P
{
  C() throws IOException | Exception | Throwable
  {
    super();
  }
}
```

✓

* if the parent class constructor throws checked exception then compulsory child class constructor should throw the same cheked exception or its parent.

OOPS-17: Singleton class

→ For any java class if we are allowed to create only one object such type of class is singleton class

Ex: Runtime, BusinessDelegate, ServiceLocator.

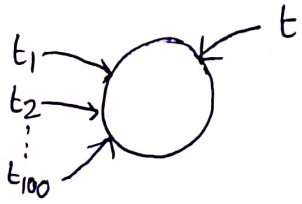
Advantage:

1. To improve performance & memory utilizations will

→ In sigleton class we are not using objects to call but using factory method.

```
Ex: class Test {
private static Test t=new Test();
 private Test()
 {
 }
 public static Test getTest()
 {
  return t;
 }
}
```

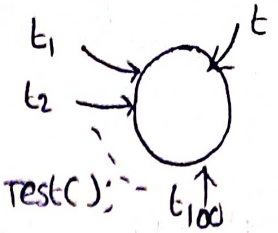
```
Test t1 = Test.getTest();
 "  t2 = "     "
 :
 "  t100 = "   "
```



Ex:

```
class Test
{
  private static Test t=null;
  private Test(),
  {
  }
```

```
Public static Test getTest()
{
  If (t==null)
  {
    t=new Test();
  }
  return t;
 }
}
```




```
Test t1 = Test.getTest();
 "   t2 =  "    "
      :
 "   t100 =  "    "
```

Ex:

```
class P {
   private p()
   {
   }
}
class C extends P {
   c()
   {
   super();
   }
}
```

→ we can't acces p() constructor from child because it is private.

→ By ~~data~~ declaring every constructor as private we can restrict child class creation.