3.

Core Java

* OOPs (51-67)

#51. OOPs (Object Oriented Programming)

15|Apr|23

① Data hiding
② Abstraction
③ Encapsulation
④ Tightly encapsulated class
⑤ Is-A Relationship
⑥ Has-A Relationship
⑦ Method Signature
⑧ Overloading
⑨ Overriding
⑩ Static control flow
⑪ Instance control flow
⑫ Constructors
⑬ Coupling
⑭ Cohesion
⑮ Type-casting

} ⇒ Security

* Data hiding :-

→ Outside person can't access our internal data directly (or) internal data should not go out directly, this oop feature called data hiding.

→ After validation (or) Authentation outside person can access our internal data. eg - Mail Service.

→ By declaring data member (var.) as private we can achieve data hiding.

```
public class Account {
    private double balance;
    public double getBalance() {
        // validation
        return balance; }
}
```

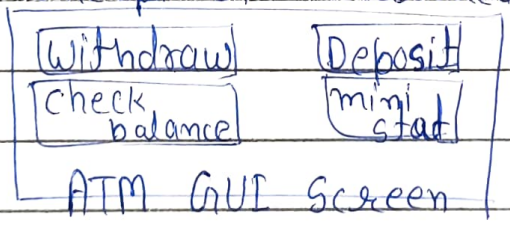
→ Adv. ⇒ Security.

Note:- It is highly recommended to declare data member (var.) as private.

* Abstraction :-

→ Hiding internal implementation and just heilight the set of services what we are offering is the concept of abstraction.

eg



→ Adv.

→ Security bcz we are not heilighting internal implementation.

→ Enhancement will become easy.

→ It improves maintainability of the application.

→ It improves easyness to use our system.

→ By using interfaces & abstract classes we can implement abstraction.

* Encapsulation :-

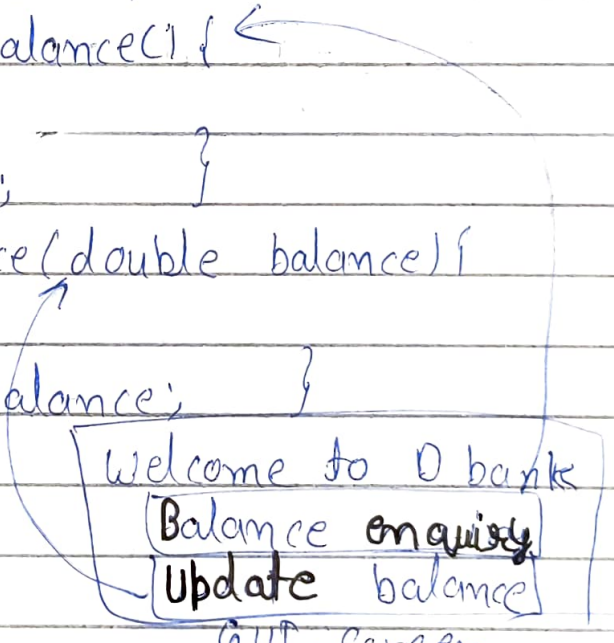

capsule

eg:-
```
class student {
    data + members
    behaviours (methods)
}
```

→ The process of binding data & corresponding methods into single unit is nothing but encaps

→ **Encapsulation = Data hiding + Abstraction.**

→ public class Account {
```
    private double balance;
    public double getBalance() {
        // validation
        return balance;
    }
    public void setBalance(double balance) {
        // validation
        this.balance = balance;
    }
}
```



⇒ Advantages
→ Security
→ Enhancement will become easy.
→ It improves maintainability of application.

⇒ Dis. advantage
→ It increases length of the code & slows down execution.

* Tightly encapsulated class :-

```
> public class Account {
      private double balance;
      public double getBalance(){
          return balance;        }
  }
```

→ A class is said to be tightly encapsulated iff each & every variable declared as private.

→ Whether class contains corresponding getter & setter methods or not & whether these methods are declared as public or not, these things we not require to check.

Q:-

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

✓ ✗ ✓

```
> class A {
      int x=10; }
> class B extends A {
      private int y = 20;
  }
```

✗ ✗

```
> class A {
      int x=10; }
> class C extends B {
      private int z = 30;
  }
```

✗

→ If parent is not tightly encapsulated then no child class is tightly encapsulated

#52. OOPs Inheritance

15/Apr/23

* IS-A Relationship (Inheritance)

→ Also know as inheritance.
→ Adv. is code re-usability.
→ by using extends key-word we can implement Is-A relationship.

```
> class P {
      public void m1() {
          Sopln("Parent"); }
  }
> class C extends P {
      public void m2() {
          Sopln("child"); }
  }
```

```
class Test {
  p s v m(S[] args) {
① P p = new P();
   p.m1();
   p.m2(); CE:
② C c = new C();
   c.m1();
   c.m2();
```

Dang. ③ parent reference to hold child obj.

```
P p1 = new C();
p1.m1();
p1.m2(); XCE:
```

④ `C c1 = new P(); XCE;`

⇒ Conclusions :-

I → whatever method parent has by-default availab to the child & hence on child reference we can call both parent & child class method.

II → whatever method child has by default not avail to the parent & hence on parent reference we can't call child specific methods.

III → Parent reference can be used to hold child object but by using the reference we can't call child specific method but we can call parent metho

IV) Parent reference can be used to hold child object but child reference can't be used to hold parent object.

* Without inheritance

> class VLoan {	> class HLoan {	> class PLoan {
300 methods	300 methods	300 methods
}	}	}

900 methods
90 hrs

* With inheritance

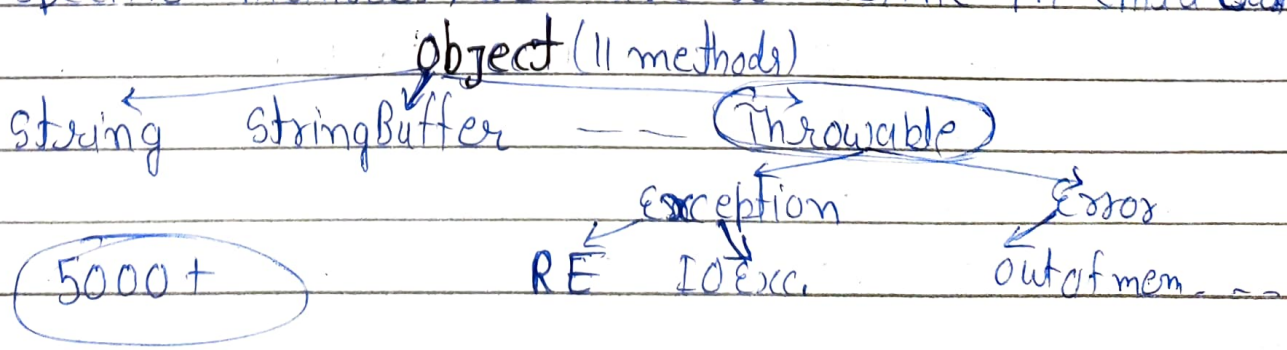
> class Loan {	> class VLoan extends Loan {
250 common methods	50 methods
}	}
> class HLoan extends Loan {	> class PLoan extends Loan {
50 specific methods	50 specific methods
}	}

400 methods
40 hrs

Note :- The most common methods, we have to define in parent class.

→ Specific methods, we have to define in child class

→



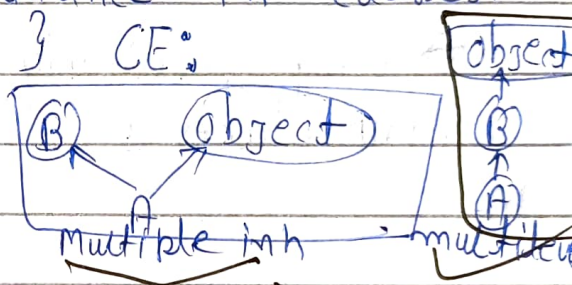
→ Total Java API is implemented based on inheritance
→ The most common methods, are defined in object cl
→ Object class acts as root cl for all Java classes.
→ Throwable acts as root for Java Exception hierarc

* Multiple inheritance :-

→ A Java class can't extend more than one class at a time hence Java won't provide support for multiple inheritance in classes.

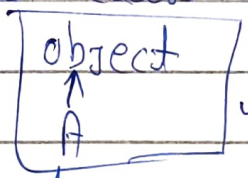
→ ~~class~~ class A extends B, C { } CE:

→ class A extends B { }



Note:→ If our class doesn't extend any other class then only our class is direct child class of object.

→ clas A { }

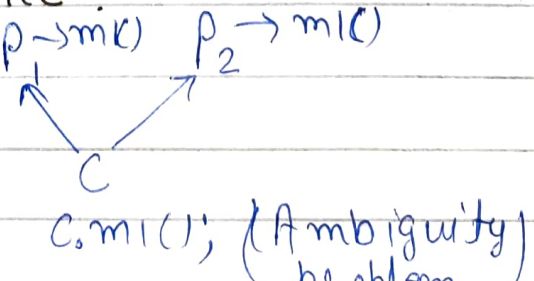


→ If our class extends any other class then our class is in-direct child class of our object.

⇒ Either directly or indirectly Java won't provid Support for multiple inheritance with wrt. classes.

Q Why Java won't provide support for multiple inheritance?

Ans:-



→ There maybe a chance of ambiguity problem hence java won't provide support for multiple inheritance.

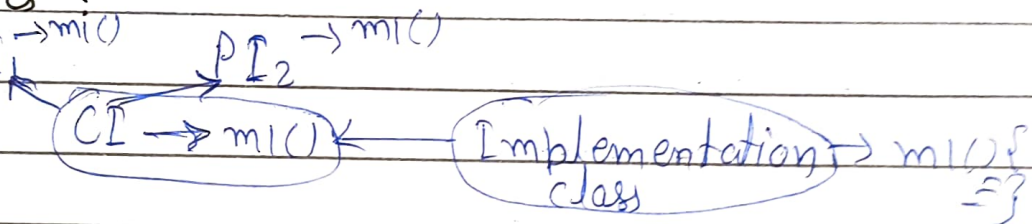
> interface A{} | interface B{}

> interface C extends A,B {}

→ But interface can extend any no. of interface simaltiniously. hence java provide support for multiple inheritance w.r.t Interfaces.

⇒ Why ambiguity problem won't be there in interfaces ?

Ans:- →



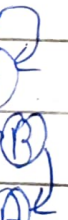
→ Eventhough multiple method declarations are available but implementation is unique & hence there is no chance of ambiguity problem in interfaces.

Note:- Strictly speaking, through interfaces we won't get any inheritance.

> class A extends A{} CE:

> class A extends B{} CE:
class B extends A{}

} cyclic inh. ✗



✗ ⇒ Cyclic inheritance

→ It is not allowed in java.

#53. OOPs Inheritance.

15/Apr/23

* Has-A Relationship.

```
> class Engine {
      // Engine specific functionality
  }
> class Car {
      Engine e = new Engine();
  }
```

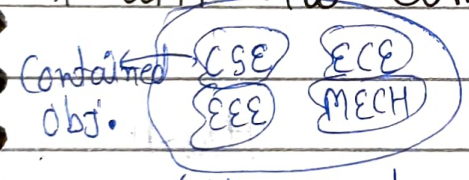
> Car Has-A Engine Reference

→ Has-A Relationship also ⇒ Composition / Aggregation

→ There is no specific keyword to implement has-A relation, but most of the time we are depending on new keyword.

→ Adv. is reusability of the code.

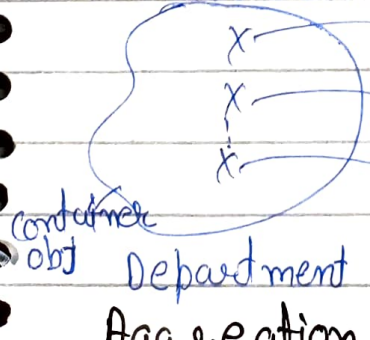
* diff b/w Composition & Aggregation



Composition

⇒ without existing container obj if there is no chance of existing contained object then container & contained objects are strongly associated & this strong association is called composition.

→ University & Department ~~relatio~~ association.



Aggreation

→ without existing container obj. if there is a chance of existing contained obj. then container & contained obj. are weakly associated & this weak association is aggreation

→ Department & Professor association

Note:- Composition ⇒ Strong association
Aggregation ⇒ Weak association

→ Composition ⇒ container obj. holds directly contained objects.
Aggregation ⇒ container obj. holds just references of contained objects.

→ Person class
IS-A relation
Student class

Test
100 methods
Has-A
Demo {
Test t = new Test();
t.m1();
t.m2();

Has-A rel.

* IS-A Vs Has-A

⇒ If we want total functionality of a class automatically then ⇒ IS-A relationship.
⇒ If we want part of functionality then ⇒ Has-A rel.

* Method Signature

public static int m1(int i, float f)

⇓

m1(int, float)

→ In java, method signature consist of method names followed by argument types.
→ return type is not part of it in java.

> class Test {
    public void m1(int i) {}
    public void m2(String s) {}
}

Test t = new Test();
t.m1(10); ✓
t.m2("durga"); ✓
t.m3(10.5); ✗ CE:

Class Test
m1(int)
m2(String)

Method table

→ Compiler will use method signature to resolve method calls.

> class Test {
    public void m1(int i) {} ⇒ m1(int)
    public int m1(int x) { ⇒ m1(int)
        return 10;
    }
}

Test t = new Test();
t.m1(10); CE:

→ Within a class two method with same signature are not allowed.

Overloading

* Overloading :-

→ Two methods are said to be overloaded iff both methods having same name but different argument types.

→ In C method overloading is not available, which increases complexity of programming.
abs(int i), labs(long l), fabs(float f).

→ In java we can declare multiple methods with same but different argument types, these methods are called overloaded methods.
abs(int i), abs(long l), abs(float f)

→ It reduces complexity of programming.

```
class Test {
    public void m1() {
        Sopln("no-arg"); }
    public void m1(int i) {
        Sopln("int-arg"); }
    public void m1(double d){
        Sopln("double-arg"); }
    p s v m(S[] args) {
        S Test t = new Test();
        t.m1();     no-arg
        t.m1(10);   int-arg
        t.m1(10.5); double-arg }
}
```

overloaded methods

C.T.P
S.P
Early Binding

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

→ It is Compile Time Polymorphism (or) Static Polymorphism (or) Early Binding

#54. OOPs Overloading

15/04/23

C-I: class Test {

overloaded methods

```
    public void m1(int i) {
        Sop("int-arg"); }
    public void m1(float f) {
        Sop("float-arg"); }
}
p s v m (S[] args) {
    Test t = new Test();
    t.m1(10);     int-arg
    t.m1(10.5f);  float-arg
    t.m1('a');    int-arg
    t.m1(10L);    float-arg
    t.m1(10.5);   CE:
}
```

byte → short
int → long → float → double
char → int

C-I → Automatic promotion in overloading :-

While resolving overloaded methods if exact match method is not available then we won't get any compile time error. immediately

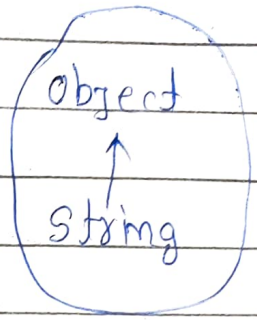
→ First it will promote argument to the next level & check, if match is there then it will be considered else it will go on promoting & if not found then give CE:,

byte → short → int → long → float → double
char → int

C-2 :->

```
class Test {
    public void m1(String s) {
        Sopln("String version");}
    public void m1(Object o) {
        Sopln("Object version");}
    p s v m(S[] args) {
        Test t = new Test();
        t.m1(new Object()); object version
        t.m1("durga"); ; String version
        t.m1(null); String version }
}
```

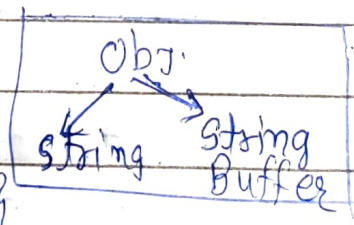
overloaded method



Note:- While resolving overloaded methods compiler will always gives precedence for child type argument when compared with parent type argument.

C-3

```
class Test{
    public void m1(String s){
        Sopln("String version");}
    public void m1(StringBuffer sb){
        Sopln("String Buffer version");}
    p s v m(S[] args){
        Test t = new Test();
        t.m1("durga"); String version
        t.m1(new StringBuffer("durga")); SB version
        t.m1(null); CE: ambiguous. }
}
```



C-4:-

```
> class Test {
    public void m1(int i, float f) {
        Sopln("int-float version"); }
    public void m1(float f, int i) {
        Sopln("float-int version"); }
    p s v m(S[] args) {
        Test t = new Test();
        t.m1(10, 10.5f); → int-float v
        t.m1(10.5f, 10); → float-int v
        t.m1(10, 10); CE: ambiguous
        t.m1(10.5f, 10.5f); CE: not find symbol. }
}
```

C-5:-

```
> class Test {
    public void m1(int x) {
        Sopln("General method"); }
    public void m1(int... x) {
        Sopln("var-arg method"); }
    p s v m(S[] args) {
        Test t = new Test();
        t.m1(); var-arg method
        t.m1(10,20); var-arg method
        t.m1(10); General method }
}
```

→ var-arg method will get least priority.

C-6 :-

```
> class Animal { }
> class Monkey extends Animal {
> class Test
overloaded method { public void m1(Animal a) {
                        Sopln("Animal version"); }
                    public void m1(Monkey m) {
                        Sopln("Monkey version"); }
    p s v m(S[] args) {
        Test t = new Test();
① Animal a = new Animal();
    t.m1(a);    Animal version
② Monkey m = new Monkey();
    t.m1(m);    Monkey version
③ Animal a1 = new Monkey();
    t.m1(a1);   Animal version
    }
```

→ In overloading method resolution always takes care by compiler based on reference type

→ In overloading run-time object wont play any role.

#55 OOPs Overriding

15/Apr/23

```
> class P {
      public void property() {
          Sop("Cash+Land+Gold"); }
      public void marry() {
          Sop("Subbalaxmi"); }
> Class C extends P {
      public void marry() {
          Sop("gtara14mel"); }
  }
```

overridden method

overriding method

→ Whatever method parent have bydefault available to the child through inheritance. If child class not satisfied with parent class implementation then child is allowed to re-define that method based on it's requirement. This process is called overriding.

→ The parent method which is overridden is called overridden method.

→ The child method which is overridding is called overriding method.

```
> class Test {
      p s v m(S[] args) {
      ① P p = new P();
          p.marry();   → parent method.
      ② C c = new C();
          c.marry();   → child method
      ③ P p1 = new C();
          p1.marry();  → child method
  }
```

*

→ In overriding method resolution is always takes care by JVM based on runtime object & hence it is also considered as Runtime polymorphism, Dynamic polymorphism or Late Binding.

* Rules for overriding :-

> class P {
    public void m1(){ }
  }

> class C extends C {
    p v m1(){ }
  }

I ⇒ In overriding method names & argument types must be matched i.e method signature must be same.

⇒ class P {
    public Object m1(){
      return null; }
  }

> class C extends P {
    public String m1(){
      return null; }
  }

→ not valid till 1.4V but valid 1.5V afterward.

II ⇒ In overriding return type must be same (till 1.4 V), after 1.5 V we can take co-variant return type.

→ A/c to this child class method return type need not be same as parent method return type, it's child type also allowed.

> parent class method Return type

	Object	Number	String	double
child class method return type	Object / String / StringBuffer ✓	Number / Integer ✓	Object ✗	int ✗

→ Co-varient return type concept applicable only for object type but not for primitive type.

III →
```
class P {
    private void m1() {}
}
class C extends P {
    private void m1() {}
}
```
→ valid but not overriding

→ parent class private method is not available to the child ∴ hence overriding is not applicable for private methods.

→ based on our requirement we can define exactly same private method in child class. It is valid but not overriding.

III →
> class P {
    final public void m1() { }
  }
> class C extends P {
    public void m1() { } CE: m1() in C can't override m1() in P, as it is final.
  }

→ We can't override parent class final method in child classess.

IV →
abstract class P {
    public abstract void m1();
  }
> class C extends P {
    public void m1() { }
  }

→ Parent class abstract method we should override in child class to provide implementation.

V →
class P {
    public void m1() { }
  }
> abstract class C extends P {
    public abstract void m1();

Small child example

→ We can override non-abstract method as abstract.

→ The main adv. of this approach is we can stop the availability of parent method implementation

to the next level child classes.

VII ⇒

Parent: final — non-final — abstract

Child: non-final/final ✗ — final ✓ — non-abstract ✓

Parent: Synchronized — native — strictfp

Child: non-Synchronized — non-native — non-strictfp

→ In overriding following above modifier don't keep any restriction.

[Static → seperate discussion]

VIII ⇒

```
> class P {
      public void m1() { }
> class C extends P {
      void m1() { }  CE: access previlage issue.
  }
```

→ While overriding we can't reduce scope of access modifier but we can increase the scope.

[private < default < protected < public]

Parent:	public	protected	default	private.
	↓	↓	↓	↓ overriding not allowed
Child:	public	protected / public	default / protected / public	

#56. Overriding OOPs

15-Apr-23

Internal reason.

```
> class P {
      public void m1(){ }
> class C extends P {
      void m1(){ }  }
  P p = new C();
  p.m1();
```

So many were accessing parent m1() suddenly u changed, will cause issue

IX

✓ ① P: public void m1() throws Exception
C: public void m1()

✗ ② P: public void m1()
C: public void m1() throws Exception

✓ ③ P: public void m1() throws Exception
C: public void m1() throws IOException

✗ ④ P: public void m1() throws IOException → child
C: public void m1() throws Exception

✓ ⑤ P: public void m1() throws IOException
C: public void m1() throws FileNotFoundExcp, EOFException

✗ ⑥ P: public void m1() throws IOException
C: public void m1() throws EOFException, InterruptedException

✓ ⑦ P: public void m1() throws IOException
C: public void m1() throws throws AE, NPE, CCE.

→ If child class method throws any checked exception compulsary parent class method should throw the same checked exception

or its parent otherwise CE:

→ but there are no restrictions for un-checked exceptions.

⇒ Internal reason:

```
class P{
    public void m1() throws IOException { }
}
class C extends P{
    public void m1() throws Exception{}
}
```

→ caller is responsible

→ 1000's of outputs

who will handle

* Overriding w.r.t Static modifier.

> class P{
    public static void m1(){ }
}

> class C extends P{
    public void m1(){ }; CE:
}

① → we can't override a static method as non-static.

> class P{
    public void m1(){}
}

> class C extends P{
    public static void m1(){}; CE:
}

② → we can't override a non-static method as static.

```
> class P {
      public static void m1(){}
  }
> class C extends P{
      public static void m1(){}
  }
```

→ method hiding but not overriding.

→ If both parent & child class method is static then we won't get any CE error.

→ It seems overriding concept is applicable for static methods but it's not overriding & it is method hiding.

* Method hiding :-

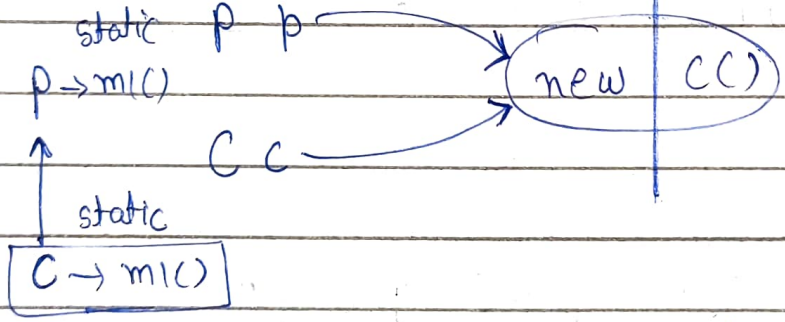
```
> class P {
      public static void m1(){
          Sop("Parent"); }
  }
> class C extends P{
      public static void m1(){
          Sop("Child"); }
  }
> class Test{
      p s v m (S[] args){
          P p = new P();
          p.m1(); → Parent
```

```
C c = new C();
c.m1();  ⟶ child
P p1 = new C();
p1.m1();  ⟶ ~~child~~ Parent.
```

→ All rules of method hiding is exactly same as overriding except the following diff.

Hiding	Overriding
① Both static	① Non-static
② Compiler Reference	② JVM Runtime object
③ C.T.P ⓪ static Polymorphism early binding	③ RTP ① Dynamic Poly. late binding.

→



→ In overriding only child reference is there as parent reference is removed. (Board example)

→ In Hiding both reference is available we can get both (chart example).

#57 OOPs Overriding

16-Apr-23

* Overriding w.r.t var-arg method

```
> class P {
      public void m1(int... x) {
          Soln("Parent"); }
  }
> class C extends P {
      public void m1(int x) {
          Soln("child"); }
  }
> class Test {
      p s v m(S[] args) {
          P p = new P();
          p.m1(10); ⟶ Parent
          C c = new C();
          c.m1(10); → child
          P p1 = new C();
          p1.m1(10); → ~~child~~ parent
      }
```

If it is overloading but not overriding

→ We can override var-arg method with another var-arg method only.

→ If we are trying to override with normal method then it will become overloading but not overriding.

→ If var-arg then

Parent
child
child

* Overriding w.r.t Variables

```
> class P {
      int x = 888; }
> class C extends P {
      int x = 999; }
> class Test {
    p s v m(S[] args){
      P p = new P();
       Sopln(p.x); → 888
      C c = new C();
       Sopln(c.x); → 999
      P p1 = new C();
       Sopln(p1.x) = 888 }
}
```

P: n-static	static	non-static	static
C: n-static	n-static	static	static
888			
999			
888			

→ Variable resolution always takes care by compiler based on reference type irrespective of wether var. is static (or) non-static

→ Overriding concept is applicable only for methods but not for variables.

Property	Overloading	Vs	Overriding
* diff. b/w.			
① Method Names	Must be Same		Must be same
② Argument types	Must be diff. (Atleast order)		Must be same (including order)
③ Method Signatures	Must be different		Must be same
④ Return Types	No restrictions		Must be same (1.4V) Co-varient allowed (1.5V)
⑤ private, static, final methods	can be overloaded		Cannot be overridden
⑥ Access modifier	No restrictions		Can-not be reduced but can increase
⑦ throws clause	No restriction		Checked Excp. rest. Un-che. Excp. No. re
⑧ Method Resolution	Compiler based on Reference type		JVM based on runtime object
⑨ Also known as	C.T.P | S.P | early binding		R.T.P | D.P | late binding

Note:- In overloading we have to chek only method names (must be same) & arguments (must be diff.) we are not require to check remaining like return type, access modifier etc.

→ But in overriding everything we have to take like names, arg. type, return type, access modifier throws class etc.

> public void m1(int x) throws IOException

overriding ① public void m1(int i)

overloading ② public static int m1(long l)

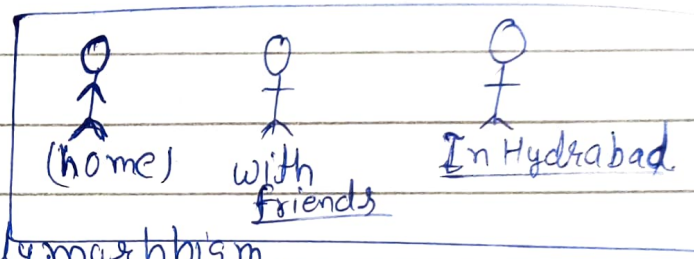
overriding ③ public static void m1(int i)

overriding ④ public static m1(int i) throws exception

⑤ public static abstract void m1(double d);

CE: illegal comb.

* Polymorphism :-



→ One name but multiple forms is concept of polymorphism.

eg:- ① Method name is same but we can apply for diff. types of arguments (overloading)

abs(int i), abs(long l), abs(float f)

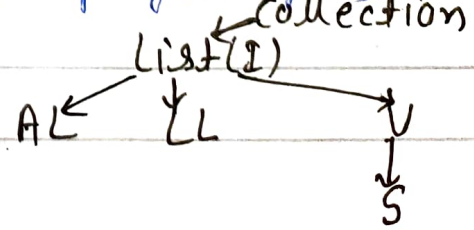
② Method signature is same but in parent class one type of implementation & in child class another type of implementation. (Overriding)

```
class P {
    marry() { Sop("Subalaxmi"); }
}
class C extend P {
    marry() { Sop("Shal, Gtara14me"); }
}
```

overriding

③ Usage of parent reference to hold child object is the concept of polymorphism

```
List l = new AL();
         new LL();
         new Stack();
         new Vector();
```



P p = new C();

p.m1(); ✓

p.m2(); XCE:

P → m1()

↓

C → m2()

→ Parent class reference can be used to hold child object but by using that reference we can call only the method available in parent class & we can't call child specific methods.

→ C c = new C();

c.m1(); ✓

c.m2(); ✓

→ But by using child reference we can call both parent & child class method.

* When we should go for Parent reference to hold child object?

→ If we don't know exact run time type of object then we should go for parent reference.

→ Object o = l.get(0); → | o | o | o | o |

Heterogeneous object

eg: The first element in arraylist can be any type, It may be Student o, Customer o, or String o or StringBuffer o, hence return type of get method is object, which can hold any object.

→ List l = m1();

public List m1() {

ArrayList/Stack

}

C c = new C();

eg: AL l = new AL();

① if we know exact runtime type of object.

② By using child ref. we can call both child & parent method

Adv.

③ We can use child ref. to hold only particular child object

dis.

P p = new C();

eg List l = new AL();

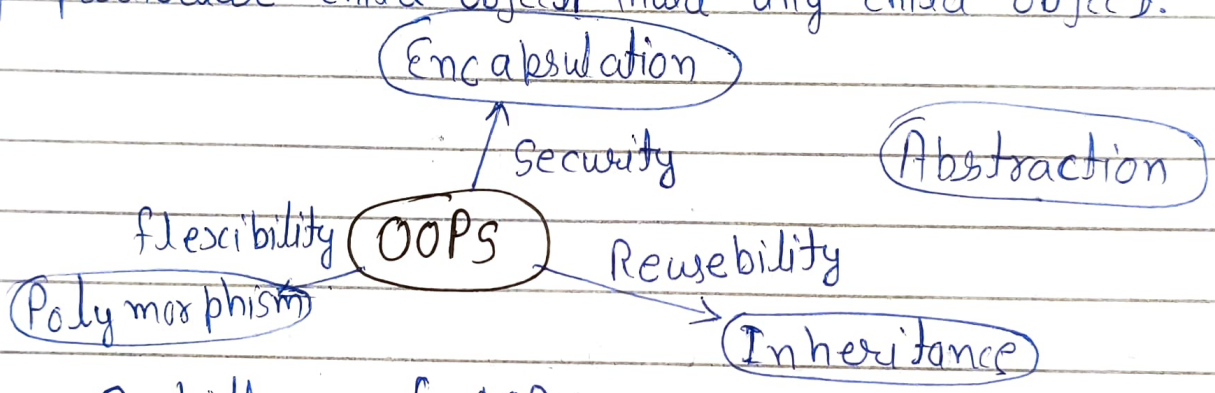
① If we don't know exact runtime type of object.

② By using Parent ref. we can call only Parent method.

dis

③ We can use Parent ref. to hold any child object.

Adv



3 pillars of OOPs

Polymorphism

S.P / C.T.P / Early binding

D.P / R.T.P / Late binding

overloading

Method hiding

Overriding

⇒ **Beautiful defination of Polymorphism:-**

A BOY starts LOVE with the word FRIENDSHIP, but GIRL ends LOVE with the same word FRIENDSHIP. word is same but attitude is different. This beautiful concept of OOPs is nothing but polymorphism.

#58. OOPs Coupling

16-Apr-23

* Coupling :-

→ The degree of dependency between the component is called coupling.

→ If dependency is more then it is tightly coupled.

→ If dependency is less then it is loosely coupled.

```
class A {
    static int i = B.J; }
class C {
    static int K = D.m1();}
```

```
class B {
    static int J = C.K; }
class D {
    public static int m1() {
        return 10; }
}
```

Problems Tightly coupled

① Enhancement difficult.

② If supress Reusebility of code.

③ Maintainability is difficult.

→ We have to maintain dependency b/w component as less as possible i.e loose coupling is good programing practice.

* Cohesion :-

→ For every component a clear well defined functionality is defined then that component is said to be follow high cohesion.

→ 70 L.of lines

```
login page
validation
inbox page
reply page
compose page
error pag-
```

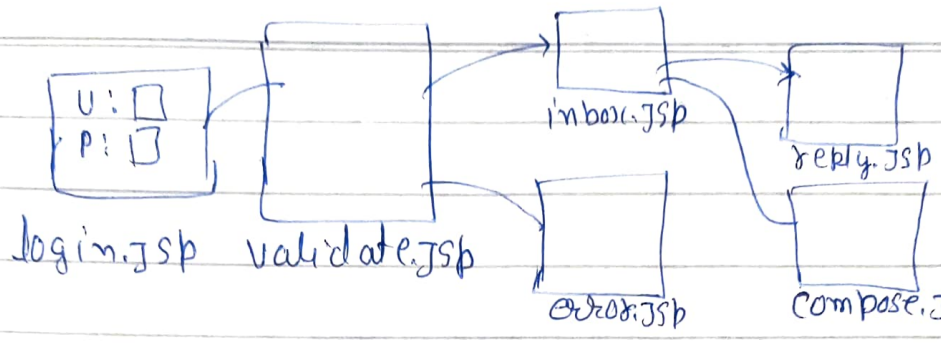
TotalServlet

→ Low Cohesion

① Enhancement is difficult

② Re-usebility is down

③ Maintainability goes down.



High-Cohesion

→ High-cohesion is always a good programing practic
→ It has several advantages
① without effecting remaining component we can modify any component. {Enhancement become Easy}
② It promotes re-usebility of the code.
③ It improves maintainability of application.

Note :- Loose coupling & High cohesion are good programming practice

* Object type-casting :-

*→ we can use parent reference to hold child object. eg: Object o = new String("durga");
→ we can use interface reference to hold implem-ented class object.
eg :- Runnable r = new Thread();
> Object o = new String("durga");
StringBuffer sb = (StringBuffer) o; ✗
RE:

A b = (C) d;

- A → class/interface name
- b → name of ref. var.
- C → (class/interface) name
- d → reference var. name

Mantra 1 :- Compile time checking one

→ The type of 'd' & 'C' must have some relation either Parent to child, child to parent or same type otherwise CE: inconvertible type.

> eg. object o = new String ("durga");
StringBuffer sb = (StringBuffer) o; ✓

> String s = new String ("durga");
StringBuffer sb = (SB) s; C.E: ✗

Mantra 2: (Compile time checking-2)

→ 'C' must be either same or derived type of 'A' otherwise we will get compiletime error: incompatible type.

eg > object o = new String ("durga");
SB sb = (SB) o; ✓

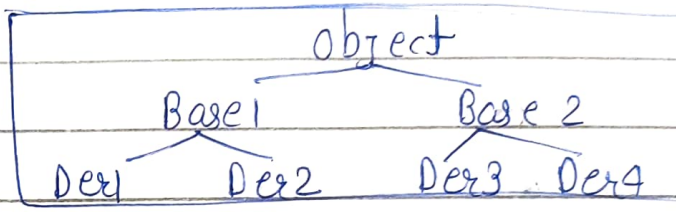
> object o = new String ("durga");
SB sb = (String) o; ✗ CE:

Mantra 3: Runtime checking

→ Runtime object type of 'd' must be either same or derived type of 'C' otherwise RE: classCastException.

> object o = new String("durga");
SB sb = (SB) o; RE:

> Object o = new String("durga");
Object o1 = (String) o;



> Base2 b = new Der4()

① Object o = (Base2) b;

② object o = (Base1) b; CE: incovertible ty.

③ object o = (Base(Der3) b; RE: CCE

④ Base2 b1 = (Base1) b; CE:

⑤ Base1 b1 = (Der4) b; CE: Incompatible type.

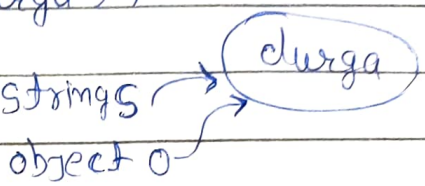
⑥ object o = (Base1

⑦ Base1 b1 = (Der1) b; CE:

#59. OOPs Type Casting

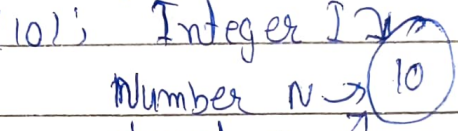
16-Apr-23

> String s = new String("durga");
Object o = (Object)s;



→ Strictly speaking, through typecasting we are not creating any new object for the existing object we are providing another type of reference variable i.e. we are performing type casting but not object casting.

> Integer I = new Integer(10);
Number n = (Number) I;
object o = (object) n;



Sop(n(I == N) true
Sop(n(n == o); true

Note:
> C c = new C();

Bb = new C(); (B) C
(A) (B) C → A a = new C();

eg:-
1 ⇒ C c = new C();

P → m1() {}
C → m2() {}

✓ c.m1();
✓ c.m2();
✓ ((P)c).m1(); → P p = new C(); p.m1();
✗ ((P)c).m2(); → P p = new C(); p.m2();

Reason:- Parent ref. can be used to hold child object but we can't called child specif method. only parent method can be called.

A → m1() {sop('A')}
↑
B → m1() {S('B')}
↑
C → m1() {S("C")};

2) C c = new C();
C.m1(); → C
((B)C).m1; → C
((A)((B)C)).m1(); → C

→ It is overriding & method resolution is always based on runtime object.

3)
A → static m1()
↑
B → static m1()
↑
C → static m1()

> C c = new C();
C.m1(); → C
((B)C).m1(); → B
((A)((B)C)).m1(); → A

→ It is method hiding & method resolution is always based on reference type.

4)
A → int x = 777;
↑
B → int x = 888;
↑
C → int x = 999;

> C c = new C();
Sop1n(c.x); → 999
Sop1n(((B)C)x); → 888
Sop1n(((A((B)C))x); → 777

→ Variable resolution is always based on reference type but not based on runtime object.

#60. OOPs Static Control Flow

16/Apr/23

* static control flow

```
> class Base {                                        (7)
i = 0[RIWO] (1)  Static int i = 10;
J = 0[RIWO] (2)  Static {
i = 10[RSW]          m1();                  (8)
J = 20[RSW]          Sop1n("First Static block");   (10)
                 }
(3)  p s v m(S[] args) {
                     m1();   (13)
                     Sop1n("main method");   (15)
                 }
(4)  p s v m1() {
                     Sop1n(J);   }   (9)  (14)
(5)  Static {                                        (11)
                     Sop1n("second static Block");
(6)  Static int i = 20;   (12)
}
```

> 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.

(1)	[1 to 6]	Java Base ↵
(2)	[7 to 12]	0
(3)	[13 to 15]	First static Block Second static Block 20 main method

⇒ Whenever we are executing a java class following sequense of steps will be executed as a part of static control flow.

① ② ③

* Read indrectly write only :-

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

→ Inside a static block if we try to read a variable that read is Direct read.

→ If we are calling a method & within that method if we are trying to read a variable that read is indirect read.

⇒ If a var. is just identified by JVM & original value not yet assigned then the var. is said to be in "Read indrectly & write only" state(RIWO)

→ If a var. is in RIWO state then ~~that read~~ we ~~operation~~ cant perform direct read but we can perform indirect read.

#61.

→ If we try to read directly then we will get CE: illegal forward reference.

1.6 version

```
> class Test {
① static int x=10;
② static {
      Sop(x); }
}  O/P: 10
   RE: main method.
```

```
> class Test {
① static {
      Sop(x); }      x=0 [R&W]
② static int x=10;
}  CE: illegal
```

```
> class Test {
① static {
      m1(); }        x=0 [R&W]
② p s v m1(){
      Sop(x); }      O/P: 0
③ static int x=10;   RE: main
}
```

#61. OOPs Static block 16/Apr/23

* Static block

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

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

Eg: - We have to define native lib. loading inside static block.

16/Apr/23

Eg 2 :- > class DbDriver{
    static{
        Register this Driver
        with DriverManager}
}

① Load Driver class
② get Connection object
③ prepare statement obj.
④ execute Query
⑤ Use Resultset.

→ After loading every db driver class we have to register driver class with driver manager but inside db driver class there is static block to perform this activity.

Note:- Within a class we can declare any no. of static blocks but all blocks will be executed from top to bottom.

Q:- Without writing main() method is it possible to print some statements to the Console.?

A: Yes, by using static block.

> class Test {
    static{
        Sopln ("Hello I can print");
        System.exit(0);
    }
}

O/P: Hello I can print.

Q:- Without writing main() & static block is it possible to print some stat. to the console?

A:- ① class Test {
    static int x = m1();
    public static int m1() {
        System.out.println("Hello, I can print");
        System.exit(0); return 10; }
}

Yes, {Till 1.6 version}

② > class Test {
    static Test t = new Test();
    { Sop("Hello, I can print..");
      System.exit(0); }
}

③ class Test {
    static Test t = new Test();
    Test() {
      Sop("Hello, I can print");
      System.exit(0); }
}

Note :- from 1.7 version onwards main method is mindatory to start a program execution.
→ From 1.7 version onwards without writing main method it is impossible to prit some statement to the console.

★ Static control flow in parent to child relationship.

> class Base {
① static int i = 10;
② static {
    m1();
    Sop("Base static Block");
}

(12) (13) (15)

```
③ p s v main(S[] args){
      m1();
      Sop1n("Base main");
  }
④ p s v m1(){
      Sop1n(J); ⑭
  }
⑤ static int J = 20; ⑱
}
```

i = 0 [RIWO] ①
J = 0 [RIWO] ⑤
x = 0 [RIWO] ⑥
y = 0 [RIWO] ⑪
i = 10 [R&W] ⑫
J = 20 [R&W] ⑯
x = 100 [R&W] ⑰
y = 200 [R&W] ㉒

```
> class Derived extends Base {
⑥ static int x = 100; 17
⑦ static {
      m2(); ⑱
      Sop1n("Derived first static Block"); ⑳
  }
⑧ p s v main(S[] args){
      m2(); ㉓
      Sop1n("Derived main"); } ㉕
⑨ p s v m2(){
      Sop1n(y); } ⑨ ㉔
⑩ static {
      Sop1n("Derived Second Static Block"); } ㉑
⑪ static int y = 200; ㉒
}
```

Javac Base.java

Base.class → Derived.class

Java Derived ↵

* Sequence of Events. [① to ⑪]

① Identification of static members from parent to child

② Execution of static var. assignment & static blocks from parent to child. [12 to 22]

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

Java Derived ↵

o/p:

0 (14)
Base static block (15)
0 (19)
Derived first static block (20)
Derived second static block (21)
200 (24)
Derived main (25)

Java Base ↵

o/p:

0
Base static block
20
Base main

→ Whenever we are loading child class automatically parent class will be loaded. but in parent loading child will not be loaded.

#62. OOPs Instance control flow

16/Apr/23

* instance control flow :-

```
> class Test {
③   int i=0;                    ⑨
④   {
        m1();                   ⑩
        Sopln("First instance block");  ⑫
    }
①   psvm(String[] args){
        Sopln("main");          ⑯
    }
⑥   public void m1(){
        Sopln(j);               ⑪
    }
⑦   {
        Sopln("second instance block");  ⑬
    }
⑧   int j=20;                   ⑭
}
②> Test t = new Test();
⑤   Test(){
        Sopln("constructor");   ⑮
    }
```

i = 0 [RIWO] ③
J = 0 [RIWO] ⑧
i = 10 [R&W] ⑨
J = 20 [R&W] ⑭

Java Test ↵

0 ⑪
first instance block ⑫
Second instance block ⑬
constructor ⑮
main ⑯

⇒ First static control flow will be executed.
① Identification of instance member from T to B [3 to 8]
② Execution of instance var. assignment & instance blocks from T to B. [9 to 14]
③ Execution of constructor (5)

Note:- → Static control flow is one time activity
→ Instance control flow is not one time activity it will be perfored for every object creation.
→ Object creation is costly operation if no specific requirement then it's not recomended.

* Instance control flow in Parent to child relationship.

```
> class Parent {
(4)   int i=10; (15)
(5)   {
        m1();  (16)
        Sop1n("Parent instance Block"); (18)
      }
(6)   Parent(){
        Sop1n("constructor"); (20)
      }
(1)   p s v main(S[] args){
        Parent p = new Parent();
        Sop1n("Parent main");
      }
```

```
⑦ public void m1(){
      Sop1n(j); (17) }
⑧ int (J) = 20; (19)
}
> class child extends Parent{
⑨ int (x) = 100; (21)
⑩ { m2(); (22)  (24)
   Sop1n("CFIB"); }
⑪ child(){
      Sop1n("child const"); (27)
   }
② p s v m(S[] args){
③ Childc = new Ch();
   sop1n("child main"); } (28)
⑫ public void m2(){
      Sop1n(y); (23) }
⑬ { Sop1n("CSIB"); (25)
   }
⑭ int (y) = 200;
}
```

i = 0 [RIWO] ④
J = 0 [RIWO] ⑧
x = 0 [RIWO] ⑨
y = 0 [RIWO] ⑭
i = 10 [R&W] ⑮
J = 20 [R&W] ⑲
x = 100 [R&W] (21)
y = 200 [R&W] (23)

Javac Parent.Java
Parent.class child.class
Java child

0 (17)
Parent instance block (18)
constructor (20)
0 (23)
CFIB (24)
CSIB (25)
child constructor (27)
child main (28)

*

⇒ Instance control flow

① Identification of instance members from Parent to child. [4 to 14]

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

③ Execution of parent constructor. (20)

④ Execution of instance var. assignments & instance block in child class. [21 to 26]

⑤ Execution of child constructor. [27]

#63. OOPs instance & static control flow

16-Apr-23

```
class Test {
    {
      Sopln("FIB");
    }
  static {
      Sopln("FSB");
  }
  Test() {
      Sopln("Constructor");
  }
  p s v main(S[] 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

> public Initialization{

① private static String m1(String msg){
Sopln(msg);
return msg; }

④ public Initialization () { m = null
m = m1 ("1");
}

⑤ {
m = m1 ("2");
}

⑥ String m = m1 ("3");

② p s v main (S [] args) {

③ Object o = new Initialization();
}

}

O/p: 2
3
1

*> class Test {
int x = 10;

① p s v main (S [] args) {
Sopln (x); → CE;
}

}. Test t = new Test();
Sopln (t.x);

⇒ From static area we can't access instance members directly bcz while executing

Static area JVM may not identify instance members.

* Q:- In how many ways we can create an object in Java.? (or)
In how many ways we can get object in Java ?

Ans:-

① By using new operator
```
Test t = new Test();
```
② By using newInstance() method
```
Test t = (Test) Class.forName("Test").newInstance();
```
③ By using Factory method
```
Runtime r = Runtime.getRuntime();
DateFormat df = DateFormat.getInstance();
```
④ By using clone() method
```
Test t1 = new Test();
Test t2 = (Test) t1.clone();
```
⑤ By using Deserialization
```
FileInputStream fis = new FIS("abc.ser");
ObjectInputStream ois = new OIS(fis);
Dog d2 = (Dog) ois.readObject();
```

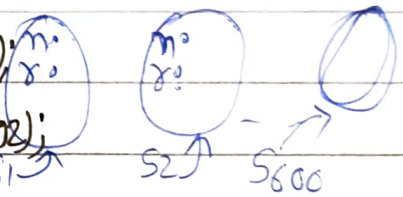
#64. OOPs Constructors

16-Apr-23

* Constructor

```
> class Student {
      String name;
      int rollno;
      Student(String name, int rollno){
          this.name = name;
          this.rollno = rollno;
      }
  p s v m(S[] args){
      Student S1 = new Student('d', 10);
      Student S2 = new Student('R', 102);
      }
  }
```

Constructor.



→ Once we create an object compulsary we should perform initilization then only object is meaningful.

→ Whenever we are creating an object some piece of the code will be executed automatically, to perform initilization of the object, this piece of the code is Constructor.

→ The main purpose of constructor is to perform initilization of an object.

Note:- The main purpose is initilazation not to create object.

* diff. b/w Constructor & Instance block.

→ The main purpose of constructor is to perform initilization of an object.

→ But other than initilization if we want to perform any activity for every object creation then we should go for Instance block.

→ Update an entry in DB for every obj. creation.

→ or incrementing count value etc.

→ Each have their own work & replacing with each other may not work always.

→ First instance block will be executed then constructor will be executed.

```
class Test {
    static int count = 0; { count++; }
    Test() { }
    Test (int i) { }
    Test (double d) { } { }
  p s v m (S[] args) {
    Test t1 = new Test();
    Test t2 = new Test(10);
    Test t3 = new Test(10.5);
    Sop("no. of obj" + count); }
}
```

* Rules :-

1) Name of the class & name of constructor must be matched.

2) Return type concept not applicable for const. even void also.

→ by mistake if we declare return type for constructor it will become method (No CE).

> class Test {
    void Test() { }
}

⇒ method but not constructor.

→ It is legal (but stupid) to have a method of same name of class.

3) The only applicable modifiers are public, private, protected, default. (other CE)

> class Test{
    static Test() { } CE: not allowed here.
}

* Default Constructor

→ Compiler is responsible to generate default constructor (but not JVM)

→ If we are not writing any constructor then only compile will generate default constructor.

→ Default & customize both will not be generated.

#6 OOPs Constructor

16/Apr/23

* Prototype of default Constructor

→ It is always no-arg Constructor
→ The access modifier of default cons. is exactly same as class (public & default)
→ It contains only one line. It is no-arg call to super class constructor.

P's code

> class Test { }

> public class Test { }

> public class Test {
  void Test() { } }

> class Test {
  Test() { } }.

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

C's code

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

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

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

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

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

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

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

→ The first line inside every constructor should be either super (or) this.

C-I :-

```
class Test{
  Test(){
    Sopln("const.");
    super();}
}
```

⇒ super() (or) this() should be first line.

CE:

C-II :-

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

⇒ we can take either one at first line. (super (or) this

⇒ CE:

C-III :-

```
class Test{
  public void m1(){
    super();
    Sopln("Hello");}
}
```

⇒ we can use super() (or) this() inside const. only.

→ i.e we can call another constr. from constr. only.

CE:

#66

* super(); this(); → only in const.
→ only in first line.
→ only one at a time.

super(), this()	super, this
① These are const. call to call super class or current class const.	① These are keyword to refer super or current class members.
② Only inside constructor. as first line. (can use)	② can use anywhere except static area.
③ can use only once in const.	③ can use any no. of time.

#66. OOPs Constructor

16/Apr/23

* Overloaded constructor:-

```
> class Test {
    Test () {
      this(10);
      Sopln("no-arg"); }
    Test( int i) {
      this (10.5) ;
      Sopln ("int-arg"); }
    Test (double d) {
      this Sopln ("double-arg"); }
  }
  p s v m (S[] args) {
    Test t1 = new Test(); double / int / no-arg
    Test t2 = new Test(10); double int-arg
    Test t3 = new Test(10.5); double arg
    Test t4 = new Test(10.1); double arg.
```

(overloaded const.)

→ Within a class we can declare multiple constructors & all this const. having same name but diff. type of arguments. hence all these const. are considered as overloaded constructors.

→ Overloading concept is applicable for constructor

→ Inheritance & overriding are not applicable for const. but overloading is applicable.

→ Every class in java including ~~java~~ abstract class can contain constructor but interface can not contain constructor.

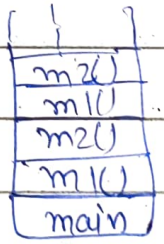
> class Test {
  Test(){ }
}
✓

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

> interface Test {
  Test(){ }
}
✗

C-I

> class Test {
  p s v m1(){
    m2(); }
  p s v m2(){
    m1(); }
  p s v main(S[] args){
    m1();
    Sop.ln("Hello"); }
}



RE: StackOverflow err.

> class Test {
  Test(){
    this(10);
  }

```
Test (int i){
    this();}
p s v m (S[] args){
    Sop.ln("Hello");
    }
}
```

CE: Recursive const. invocation

→ Recursive method call is a Runtime Exception
→ Recursive const. invocation ⇒ even a chance then code won't compile.

C-II

```
class P{ }
class C extends P{ }
```
✓

```
class P{
    P(){ } }
class C extends P{ }
```

```
class P{
    P(int i){ } }
class C extends P{
    }
```
✗ CE:

Note → If parent contains any argument constructor then for child classes we have to take special care w.r.t. constructors.

* → whenever we are writing any-arg const. it is heighly recommended to write no-arg const. also.

C-III > class P{
P() throws IOException{ }
}

> class C extends P{
} ✗ CE: UnReported Exception

→> class C extends P{
C() throws IOException / Exception / Throwable{
super(); }
}

→ If parent class const. throws any checked excepti
then compulsary child const. should throw
the same checked excp. or its parent.

→

Q:- Which are valid.

1. Main purpose of const. is to create an object. ✗
2. Main purpose of const is to perform initi. of object. ✓
3. Name of the const. need not be same as class name ✗
4. Return type applicable for constr but only void. ✗
5. We can apply any modifier for constructor. ✗
6. Default const. generated by JVM ✗
7. Compiler is responsible to generate default const. ✓
8. Compiler will always generate default const. ✗
9. If there is no no-arg const then compiler will ✗
generate default constr.
10. Every no-arg const. is default const. ✗
11. Default const is always no-arg const. ✓

12. The first line should be either super() or this()
    If we are not writing anything then it will
    generate this()
13. both overloading & overriding are applicable
14. for const. inheritance is applicable
15. only concrete classes contain constructor. but
    abstract not.
16. Interface can contain constructor.
17. Recursive const. invocation is Runtime excepti
18. If parent const. throws checked exception then
    complsary child const. should throw the same
    checked or it's childx

#67 OOPs ~~Constructor~~

16|Apr|23

* Singleton class

→ For any java class if we are allow to create only one object. such type of class is called Singleton class.

eg:- Runtime, BusinessDelegate., ServiceLocator etc.

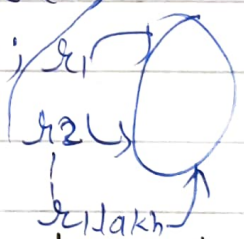
⇒ Advantage.

→ If several people have same requirement then it is not recommended to create seperate a object for every requirement. ∞

→ we have to create only one object & we can re-use same object for every similar requirement. so that performance & memory utilizations will be improved.

→ This is central idea of Singleton classes.

> Runtime r1 = Runtime.getRuntime();
Runtime r2 = Runtime.getRuntime();



* How to create our own Singleton class

```
> class Test {
      private static Test t = new Test();
      private Test() { }
      public static Test getTest() {
          return t;
      }
  }
> Test t1 = Test.getTest();
  Test t2 = Test.getTest();
```

→ We can create our own singleton classes for this we have to use private constructor & private static variable. & public factory method.

II > class Test {
    private static Test t = null;
    private Test { }
    public static Test getTest() {
        if (t == null) {
            t = new Test();
        }
        return t;
    }
}

→ At any point for test class we can create on one object. hence test class is singleton class.

Q :- class is not final but we are not allowed to create child classes how it is possible?

:-> class P {
    private P() { }
}

> class C extends P {
    C() {
        super(); } CE: can't access const.
}

→ By declaring every const. as private we can re child class creation.