OOPS

module-1
1) Data hiding
2) Abstraction
3) Encapsulation
4) Tightely 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 (Singleton classes)
13) coupling
14) Cohesion
15) Type Casting

Data hiding

↳ Our data should not go outside without validation. (directly)

↳ After validation & identification user can access data.

```
Public class Account {

    Private double balance;
    .
    .
    .

    Public double getBalance()
    {
      // validation
      return balance;
    }

}
```

we achieve Security.

Abstraction: → An abstract idea.

→ Hiding Internal representation. A set of service what we are offering.

Hiding → Security.

Enhancement become easy.

Easy to System maintainability

→ By using Interfaces & Abstract classes we can implement abstraction.

Encapsulation

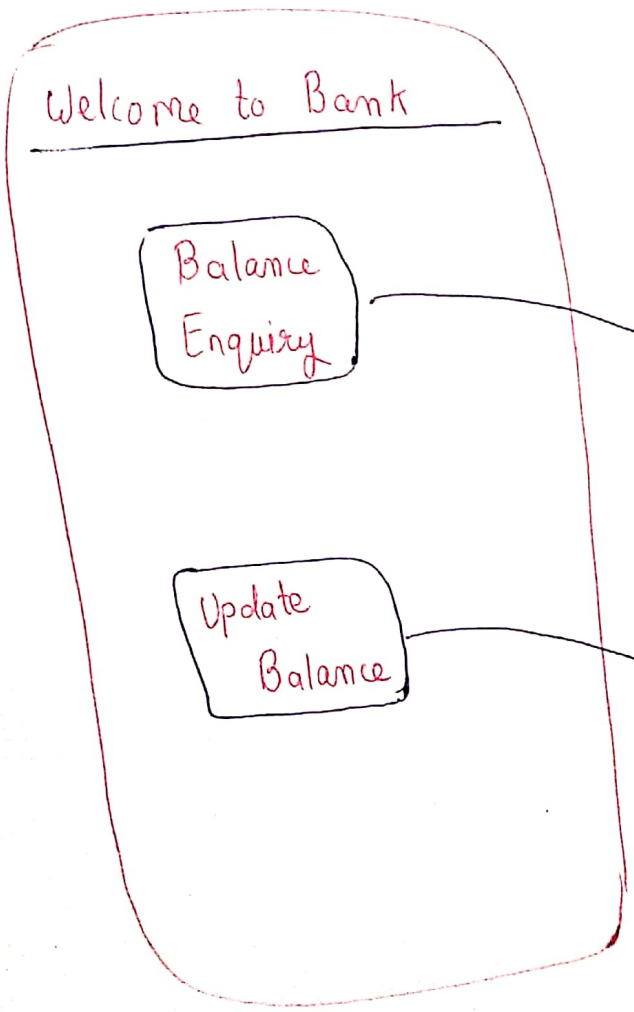
→ class {

data member;
+
behavior (methods)

}

→ The process of binding data member & corresponding method is into a single unit is called an Encapsulation.

→ Data hiding & Abstraction is called Encapsulation.




```
Public class Account
{
  private double Balance;

  Public double getBalance()
  {
    // validation;
    return Balance;
  }

  Public void setBalance(double
  {
    // validation;
    this.Balance = Balance
  }
}
```

A class is said to be tightly encapsulated if each & every variable declared as private, whether class contain corresponding getter & setter method. These method public or not, not require to check.

If a parent class is has non-private variable.

Then its child class als called as non-encapsulated.

Module - 1

Security

- Data hiding
- Abstraction
- Encapsulation
- Tightly Encapsulation

(Is-A relationship) Inheritance

extends

Re-usability

```
class P
{
   Public void m1()
   {
     Sout("parent");
   }
}

class C extends P
{
   Public void m2()
   {
     Sopen("child");
   }
}
```

```
class Test
{
  PSV main(String[] args)
  {
    P p = new P();
```

① { p.m1() ✓
  { p.m2() ×

(2) { C c = new C();
  { c.m1() ✓
  { c.m2() ✓

(3) P p1 = new C();
  p1.m1(); ✓
  p1.m2(); ×

Child object C
Polymorphism

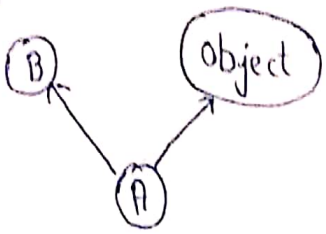
(4) C c1 = new P();

Incompatible type found P
require C.

Java not support multiple Inheritance. in classes.

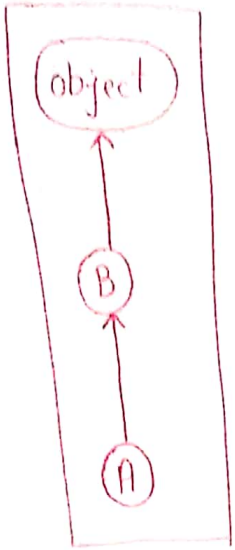
A java class can't extend more than one class at a time.

```
class A extends B
{

}
```



Multiple Inheritance not possible

✗



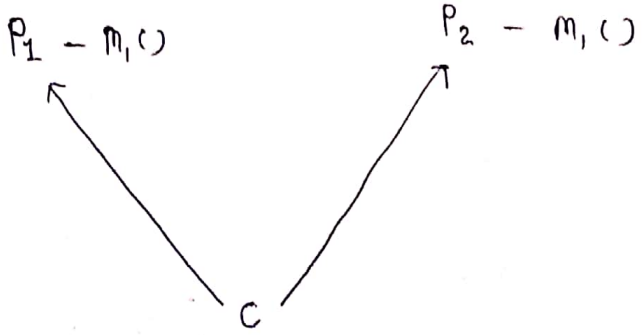
multilevel Inheritance

✓

```
class A
{

}
```

✓

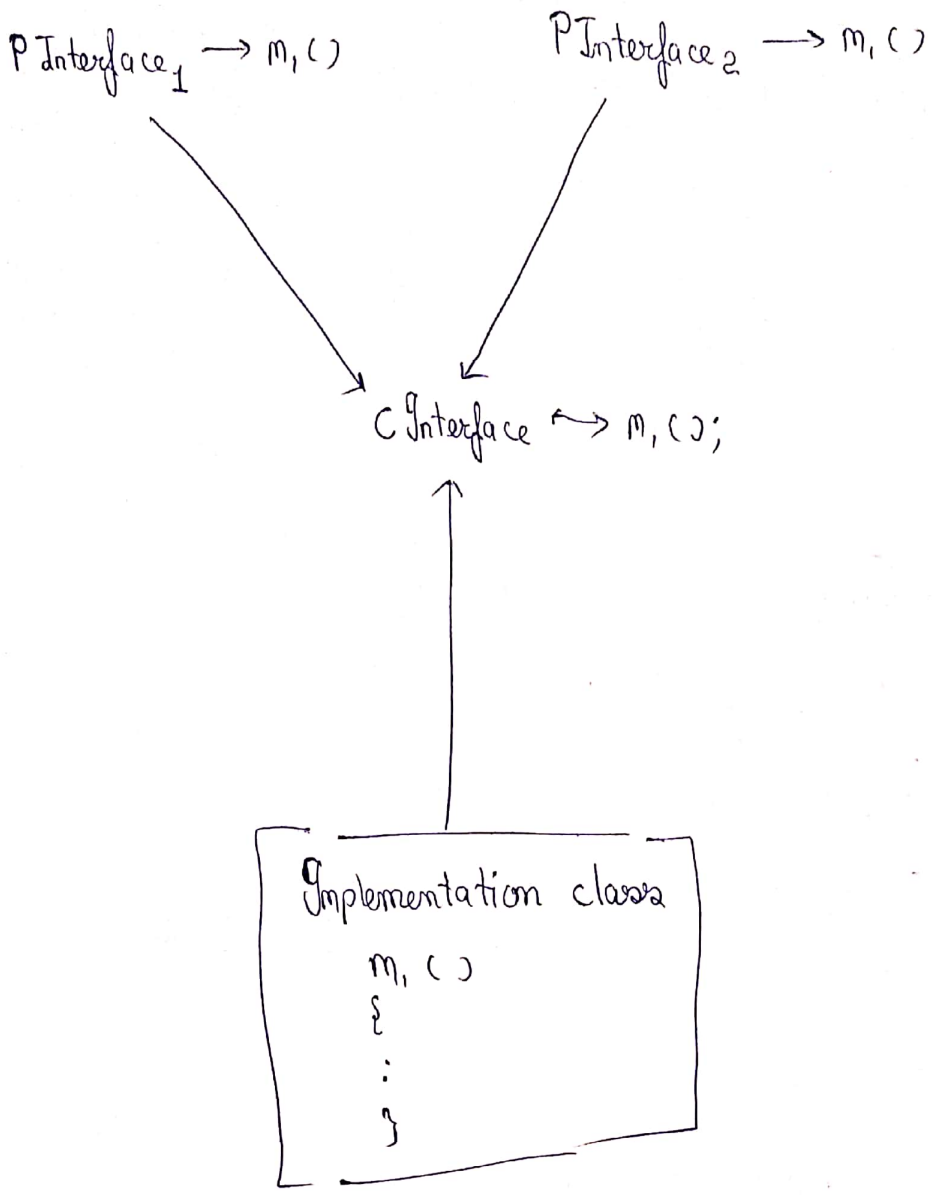
Q] Why java won't provide support for multiple Inheritance?



→ C.m1();

→ Ambiguity problem.

Interface extends multiple inheritance (Any no. of interface simultaneously).



Even though multiple declaration of [m1()] available but implementation is unique. Hence there is no chance of ambiguity problem in interfaces.

Note :- Through interface we don't get any inheritance

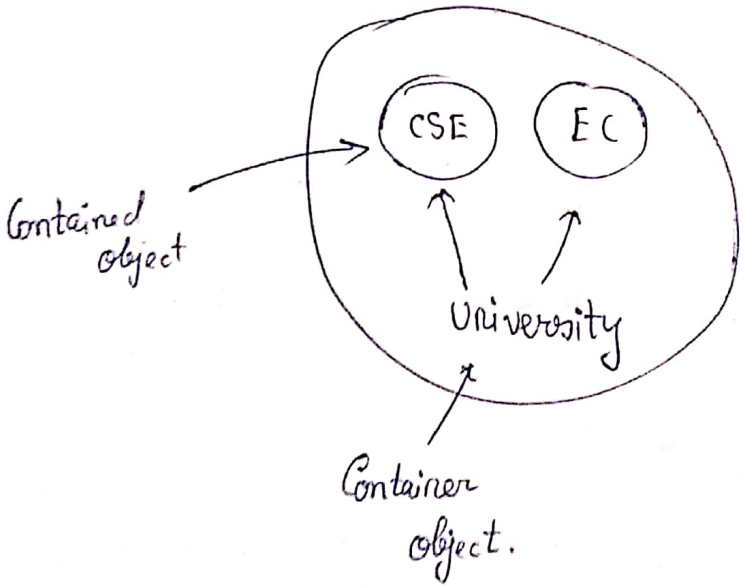
No multiple inheritance as such only declarations are there.

→ Cyclic Inheritance is not allowed in java.

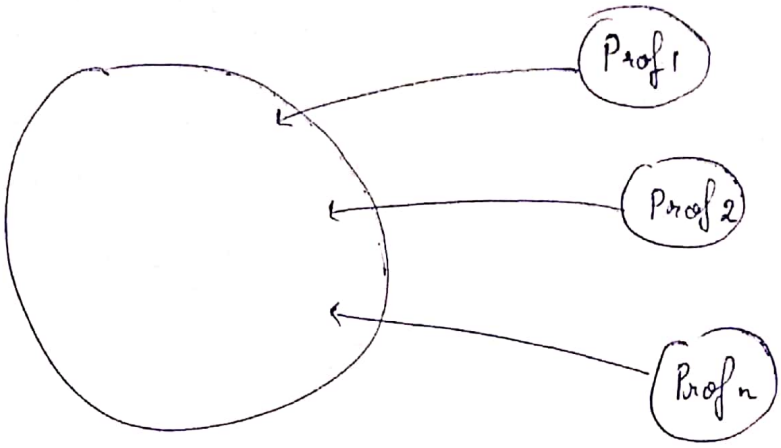
Has-a relationship - Common relationship usage.

1] ->
```
class Student
{
    string name;      // Student has-a name

}                     // Student Has a name reference
```

→ Composition or Aggregation.
→ No specific keyword (i.e) Extend
→ Depend on 'new' operator
→ Write once used anywhere. Re-usability



Strong relationship associate.
↓
Composition



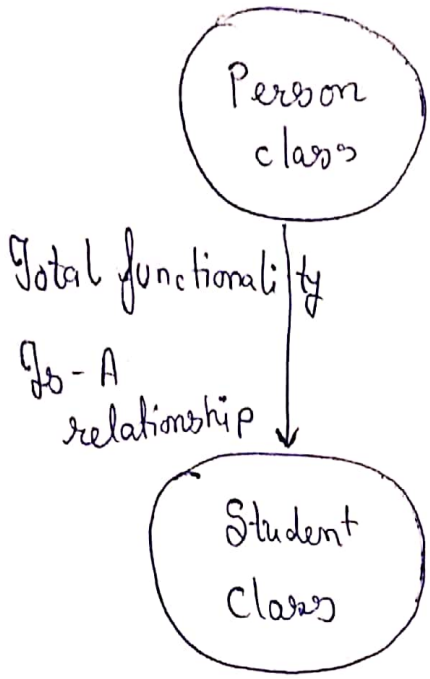
→ If dept is closed then professor should not be closed.

→ If the container object is weakly associated with contained object.

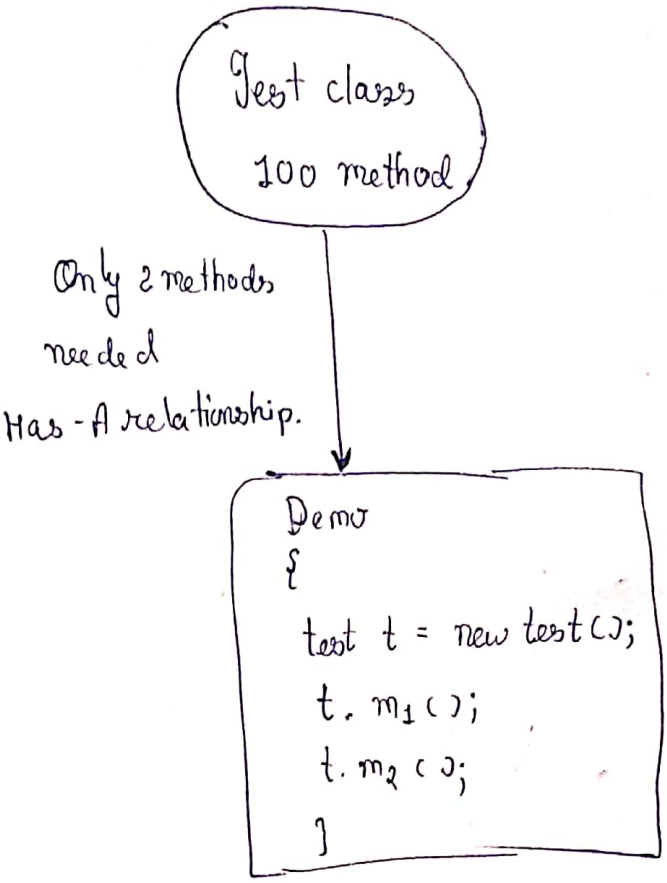
→ Weakly associated

→ Aggregation.

Is-A relation



Has-A relation



Method Signature

```
Public static int m1 (int i, float f);
```

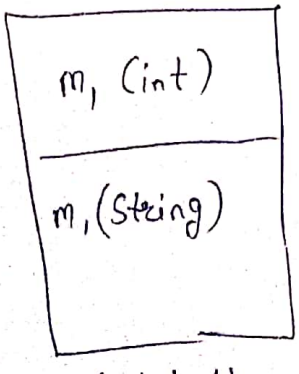
⇓

method name – arguments type.

m1 (int, float)

// return type is not a part of method signature in Java.

Jvm



method table

```
class test{
    Public void m1(int i){ }
    Public void m1(String s){ }
}
```

Compiler will use method signature to resolve method calls.

Case-1]

```
class Test {
    Public void m1(int i){ }
    Public int m1(int x){ }
}
```

→ m1(int) is already defined in Test.
→ return type is not play a major role

Overloading - Same name
- But different argument

→ 2 methods are said to be overloaded iff both methods having same name but different arguments types.
→ Changing in argument type - new method name -> Increase complexity.

Overloading - compile time polymorphism
- Early binding
- method resolution is taken by compiler
  ↳ In overriding is taken care at run-time.

Overloading Concepts [5-6 case]

Case-1

```
class Test
{
    Public void m1(int i)
    { Sopen("int"); }

    Public void m1(float f)
    { Sopen("float"); }
}
```

Overloading

```
psvm(String[] args)
{
    Test t = new test();
    t.m1(10); // int
    t.m1(10.5f); // float
    t.m1('a'); // int
}
```

If exact match will not get

↓

It will not raise compile time error

↓

It will give chance to extend **automatic promotion** in overloading

char → int

byte → short → int → long → float → double

char → int

Case-2

```
class Test{
    Public void m1(int i)
    Public void m1(String s)
        Sopen("String");
    Public void m1(Object o)
        Sopen("object");
}
```

```
psvm(String[] args)
{
    test t = new test();
    t.m1("face"); //String
    t.m1(new object); // object
    t.m1(null); //String.
```

String → Object

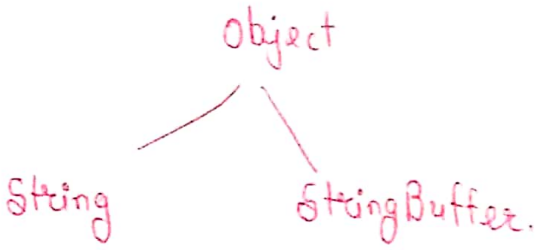
Case - 3

```
class test
{
   Public void m1(String s)
   {
      Sop ("String");
   }
   Public void m1(StringBuffer sb)
   {
      Sop ("String Buffer");
   }
}
```

Overloaded methods

```
test t = new test();
t.m1("Durga");  // String
t.m1(new StringBuffer()); // String Buffer
t.m1(null); //
```

C.E. reference to m1() is ambigious.



Case - 4

```
Public void m1(int, float)
{
   Sop ("int - float");
}

Public void m2(float, int)
{
   Sop ("float - int");
}
```

```
t.m1(10, 10.5f); // int - float
t.m1(10.5f, 10); // float - int
t.m1(10, 10); C.E ambigious
t.m1(10.5f, 10.5f);
   // Cannot find symbol
```

Case - 5

```
class test {
    Public void m1(int)
      {
        Sop(int);
      }
    Public void (int... x)
      {
        Sop("arg");
      }
}
```

```
t.m1(1);     //int
t.m1( );     // arg
t.m1(10,20); // int
```

→ If no other method matched then only var-arg method matched.

→ JVM - will use Switch-case to check method signature.

Case - 6

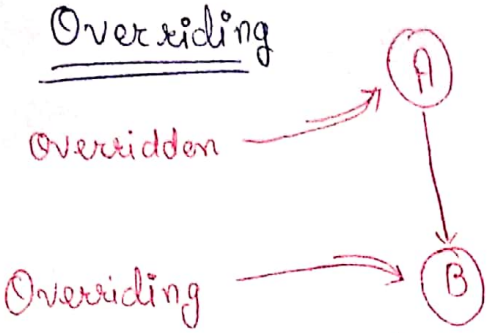
```
class Animal
{
}
class monkey extends animal
{
}
```

```
Public void m1(Animal a);
Public void m1(monkey m);
```

```
Animal a = new Animal();
t.m1(a);   //animal

Monkey m = new Monkey();
t.m1(m);   // monkey

Animal a = new monkey();
t.m1(a);   // animal
```

In overloading - only compile time reference will be considered.

Overriding



Inheritance
need to redefine method.
in child scope.

→ Runtime polymorphism, | dynamic polymorphism, | late binding.

→ In overriding method resolution will check at run-time.

→ In overloading method resolution will check at compile-time

---

```
class P {
    marry()
    {
      sop("parent");
    }
}

class C extends P {
    marry(){
      sop("child");
    }
}
```

```
class Test {
  psvm(String[] args) {
    P p = new P();
    P.marry();   // parent

    C c = new C();
    c.marry();   // child

    P p = new C();
    P.marry();   // child
```

1] At ~~first~~ first JVM will check if the P has marry method. then continue

2] At run time child reference will taken into consideration and ~~of~~ called child marry method.

Method binding will done lately at run-time

Rules for Overriding

1] Method name must be same
2] And argument type must be matched.
} method signature must be same.

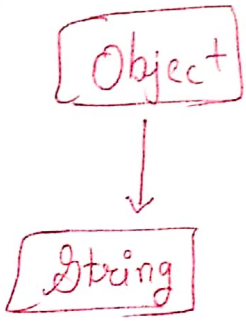
3] Untill 1.4 version return must be same (type)
↳ After 1.5 version return type ~~must~~ can be different of [Co-variant type]

Case-1

```
class P
{
    Public object m1()
    {
      return null;
    }
}
```

```
class C extends P
{
    Public String m1()
    {
      return null;
    }
}
```

Co-variant return type



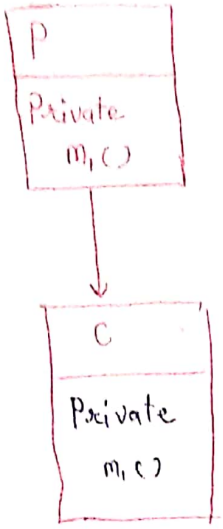
Parent class method return type	Object	Number	String	int
Child class method return type	Object / String / String Buffer ✓	Number / Integer ✓	object ✗	Double ✗

Co-variant is only applicable to object only. But not for Primitive type.

Return type - done ✓

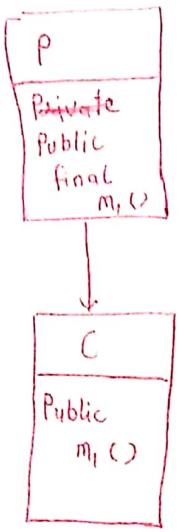
Access modifier

Case - 1



→ Parent class private method not available to child.
& hence overriding concept not applicable for private methods.
It is valid, but not overriding.

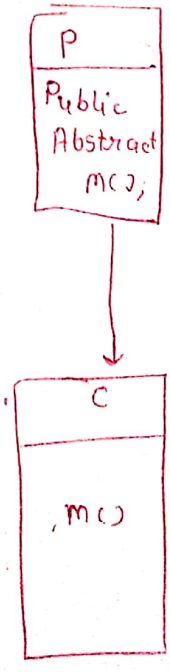
Case - 2



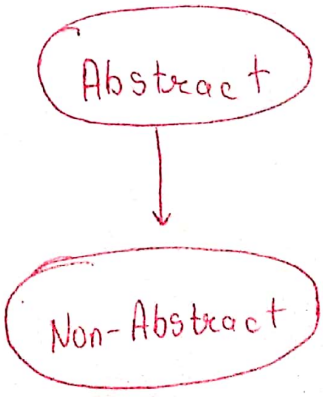
→ If it is Parent is public, easily can be override

→ If parent is final, then it can not overide
Bcoz overridden method is final

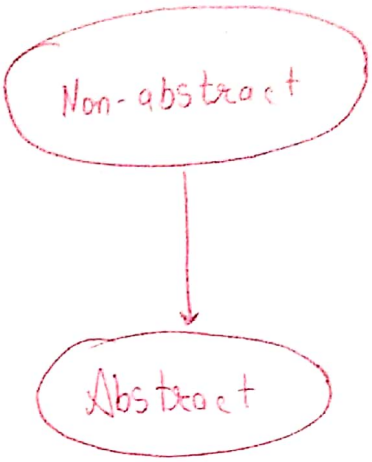
Case - 3



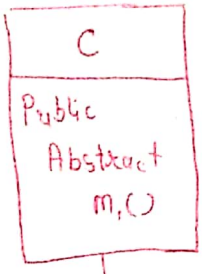
→ parent class abstract method we should override to provide implementation.



It is valid.

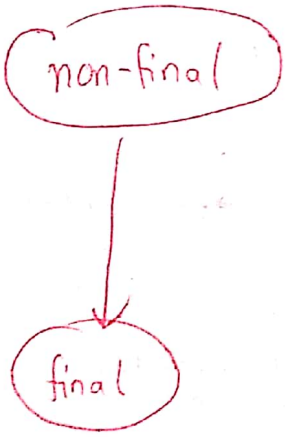


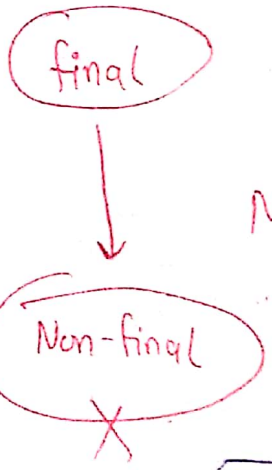
Abstract



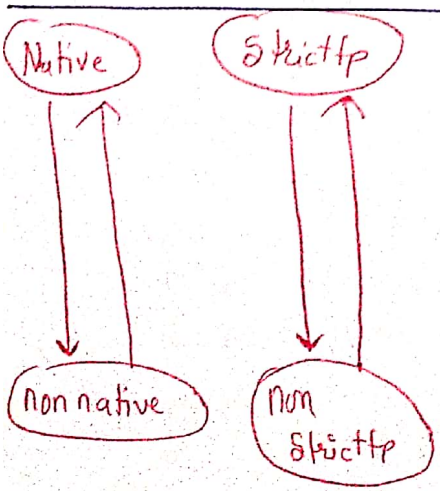
← Compulsory need to implemention the abstract method.

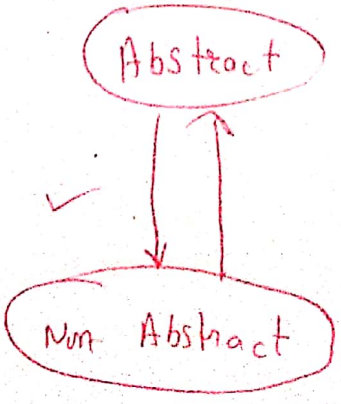
← Stop parent method logic availability

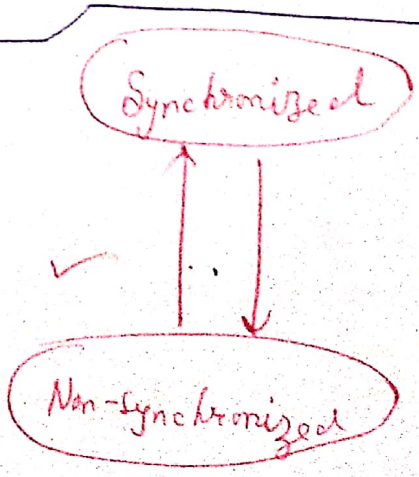




Not override



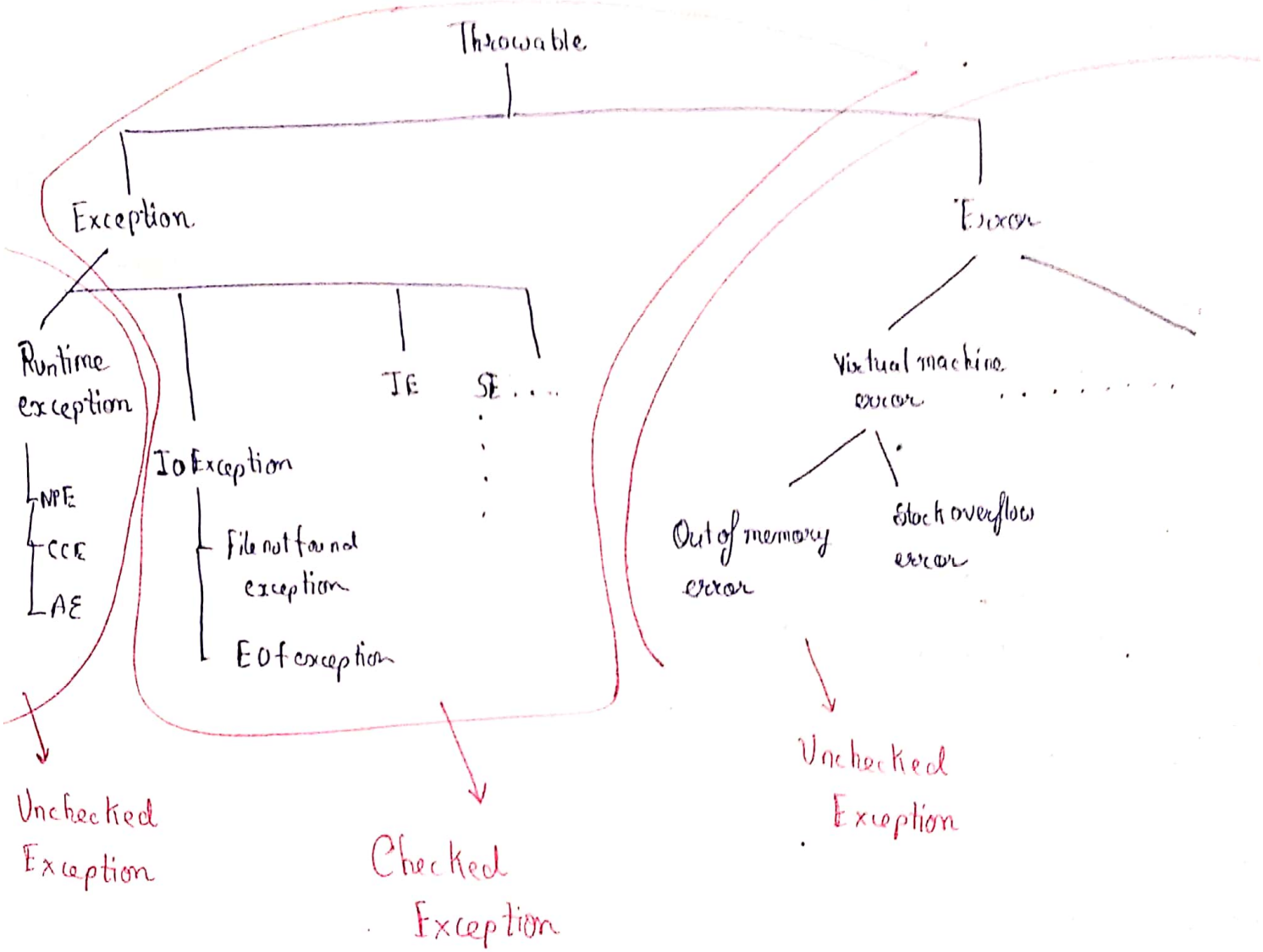




While overriding the scope of access modifier we can't reduce.

If u want increase the scope.

Private < default < protected < public

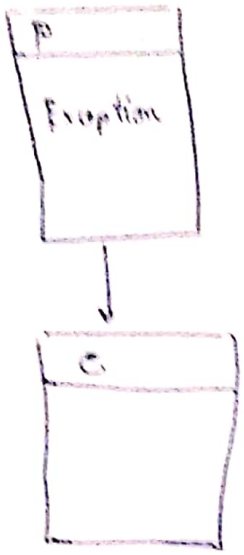


1) If ~~child class should through~~ checked exception then compulsory ~~its~~ parent

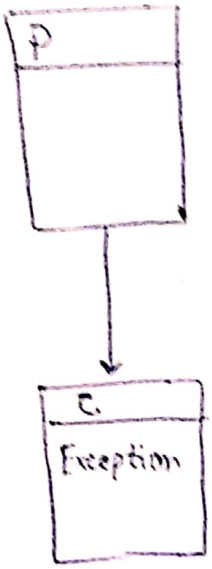
If child class method through any checked Exception Compulsory the parent class should through the (same checked exception or its parent.)

2) No restriction for unchecked exception

(1)



(3)



Overiding rules

1] method name must be same
argument type must be same
} method signature must be same.

2] Return type must be same
↳ OR Co-variant can possible.

3] parent class private method not visible to child class.
→ No overiding concept applicable

4] final method we can't override.

5] Abstract method we can override

6] Synchronized, Strictfp, native modifier won't give any restriction in overiding

7] While overiding the scope of access modifier we can't reduce, if it can be increase

8] If any child method throws any checked exception then its parent method throws same checked exception or its parent exception.

9] No restriction for unchecked exception.

→ Overriding with respect to static methods

①

```
class P {
    psvm1()
    {
    }
}
class C Extends P
{
    Public static void m1()
    {
    }
}
```

Class level static method

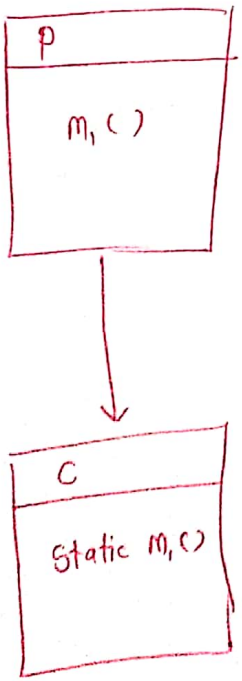
object level static method.

Compile time error : Bcoz we cant overide class level method with object level method

: Static with non-static method.

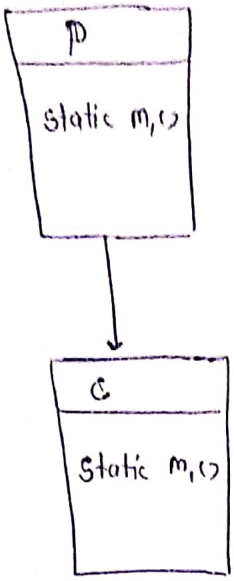
---

②



Bcoz we can't override non-static method as static.

③



→ No compile time error.

→ It's not overriding but it is method data-hiding.

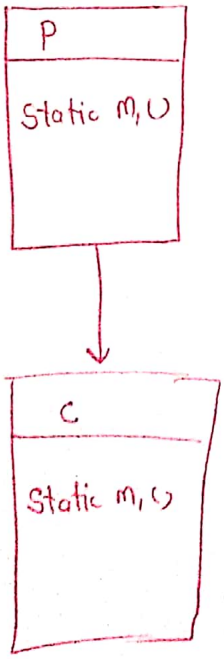
→ If both parent and child method are static then we won't get any compile time error.

→ It seems overriding concept applicable for static methods but it is not overriding

→ It is method-hiding

Method Hiding – All rules Same as overriding except

↳ Method resolution always takes by compiler based on reference type.
↳ Overriding resolution always takes by JVM based on runtime object.

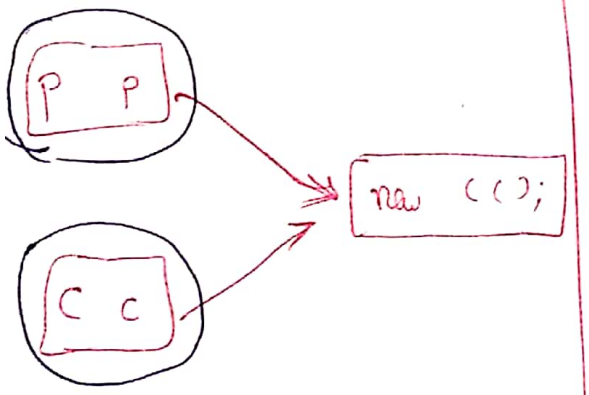



```
P p = new P();
p.m,();   ⟶ parent

C c = new C();
c.m,();   ⟶ child

P p = new C();
p.m,()    ⟶ parent
```

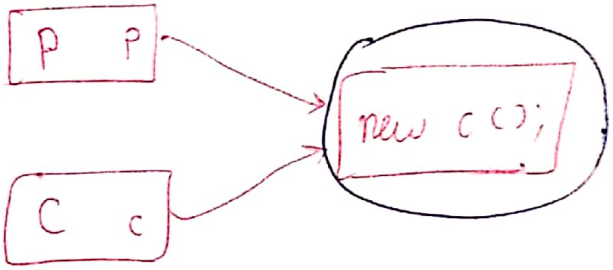
~~Not~~ ∴ Only for static modifier.

Method hiding	Overriding
① Both methods static	Non-static
② Compiler time reference	JVM Run time reference
③ Static polymorphism	Dynamic polymorphism

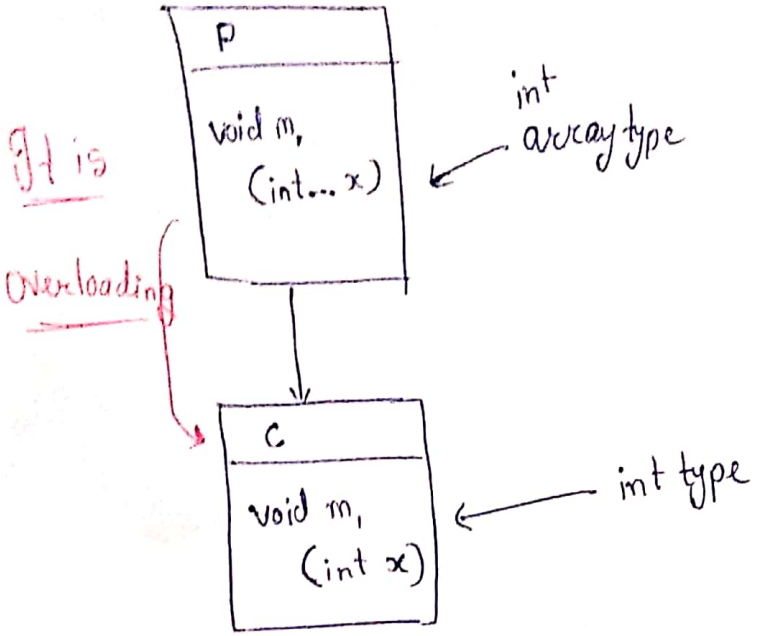
→ Both method copy are available



→ Old copy is not available only new copy is available.



Overriding with Respect to var-args method. [1.5 version]



P p = new P();
p.m1(); → parent

C c = new C();
c.m1() → child

P p = new C();
p.m1(); → parent

→ Method name same

→ Argument no. is different

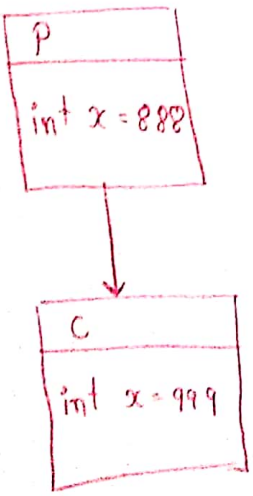
If replace child method with var-arg method then it is overriding

---

Overriding w.r.t to variables.

→ Overriding concept only applicable for method only

→ In overriding - variable resolution takes care by compiler based on reference type.



P p = new P();
p.x; → 888;

C c = new C();
c.x; → 999

P p = new C();
p.x; → 888

No changes for static variables.

P → Non-static
C → Non-static

P - static
C - static

P - static
C - Non-static

P - non-static
C - static

→ Difference between Overloading & Overriding

Property	Overloading	Overriding
1] Method name	must be same	must be same
2] Argument types	must be different [Atleast order]	must be same [including order]
3] Method signature	must be different	must be same
4] Return type	No restriction	must be same until 1.4 V From 1.5 v co-variant return types allowed
5] Private, static final methods	can be overloaded	Cannot be overridden
6] Access modifier	No Restriction	The scope of Access modifier cannot be reduced but we can increase
7] throws class	No restriction	Checked Exception child class throws Checked Ex... then parent class must throws same checked Ex or its parent. Unchecked Exception - No restriction
8] Method resolution	Reference type, Compiler	Runtime object JVM
9] known	Compile time polymorphism Early binding Dynamic Static	Run time polymorphism Late binding Dynamic

Polymorphism

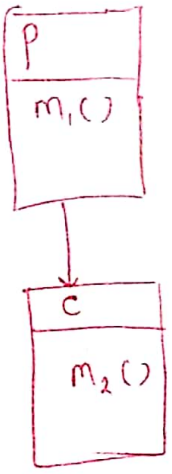
One name with different form [multiple]

Same method name with different behavior

- Overloading
- Overriding

List l = { new AL();
new LL();
new Stack();
new vector();

→ parent class can be used to hold child object, But using that reference we can call, only methods available in parent class. We can't call child specific methods.



P p = new C();

p.m1() ✓

p.m2() ✗

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

C c = new C();

If we know run time object

Eg:- AL l = new AL();

→ We can call both parent & child methods

→ Only child object can hold

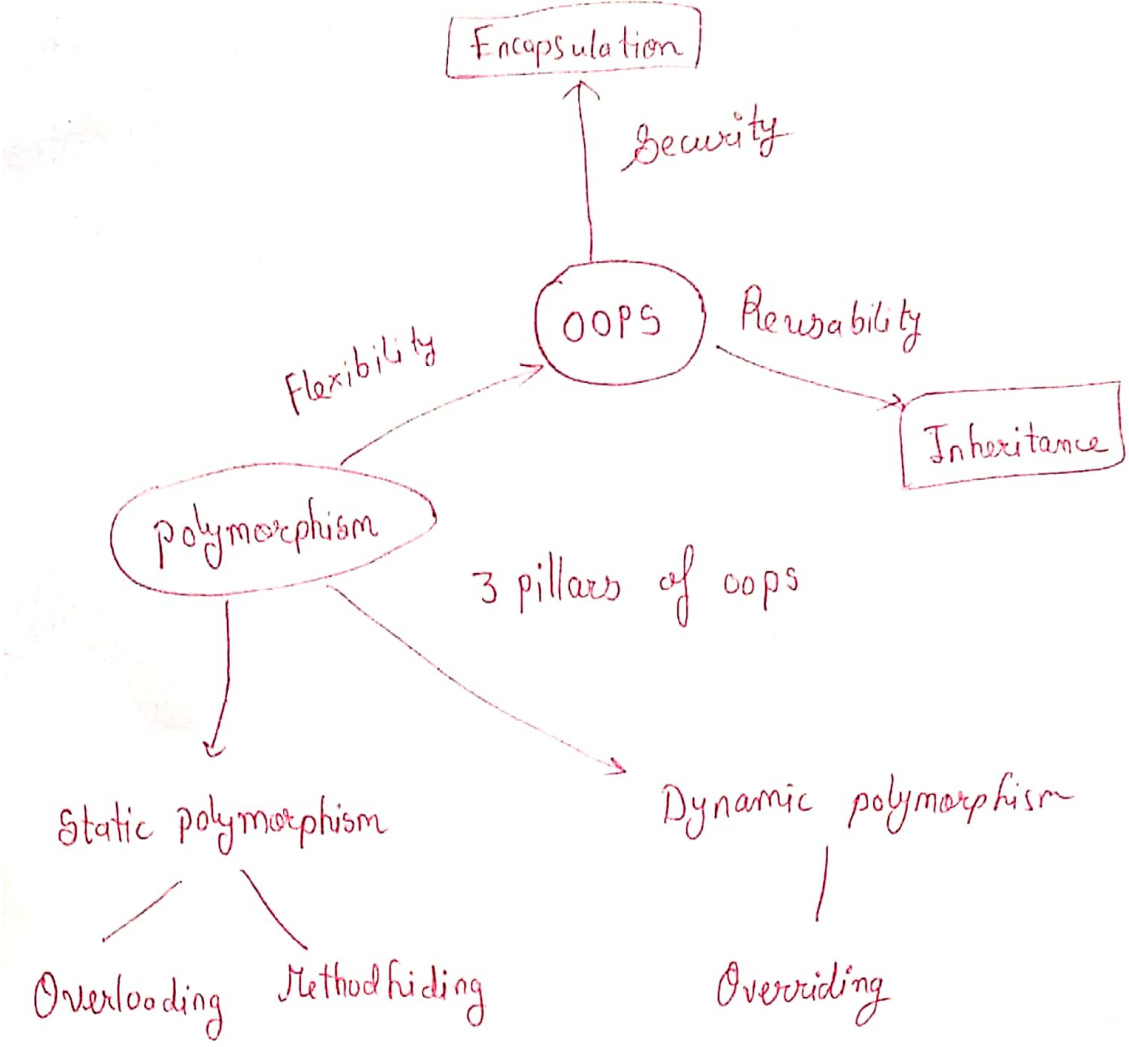
P p = new C();

If we don't know run time object

Eg:- List l = new AL();

→ We can call only child specific methods.

→ Any child object can be hold.



→ Coupling & Cohesion are two OOPS feature only. [28]
→ These two are mainly used at advanced level. (MVC), (framework)

→ Coupling ←

→ The degree of dependency between components is called Coupling.

Less dependency - loosely coupling.
High dependency - Tightly coupling. - worst coding practise

- → Enhancement difficult
- → Re-usability less
- → Maintenance less.

→ Cohesion

A clear well defined functionality are defined for a component, then that component is set to be follow high cohesion.

→ Object type casting ←

→ We can use parent reference to hold child object.

i.e. Object o = new String ("Durga");

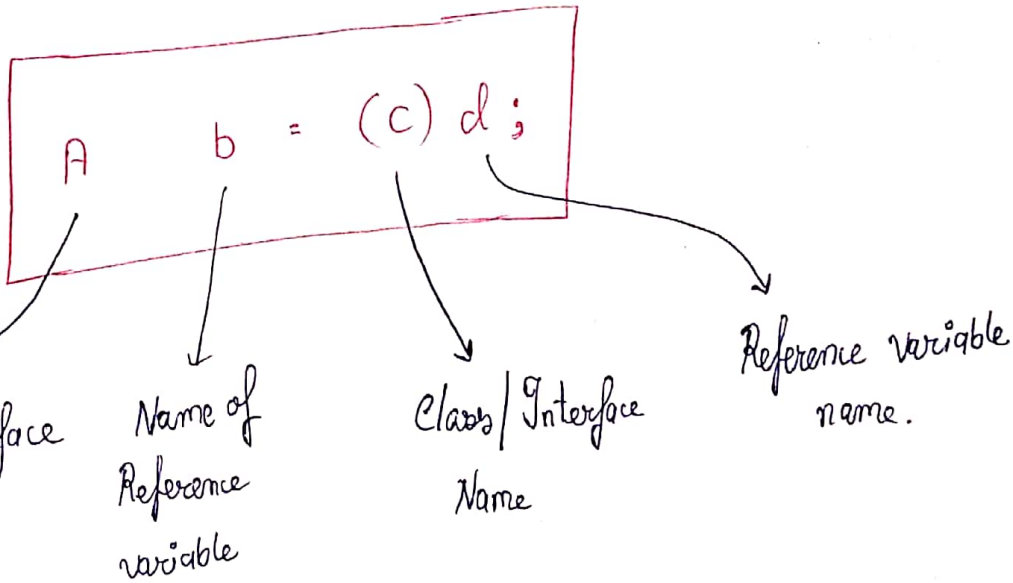
→ We can use interface reference to hold implemented class object.

i.e. Runnable r = new Thread();

~~Object o = new Object ("dur~~

Object o = new String ("durga");

StringBuffer sb = (StringBuffer) o;



Activity
1) Try to convert (d) object to (C) type
2) (C) type object assign to the (A) type reference variable

Total 3 condition will be check
2 condition by compiler
1 condition by JVM.

Object casting

A b = (C) d

Compiler will check 2 condition.

1) Conversion of d to (c) possible.
   If there is a relation bet^n d & (c).

The type of (d) and ('c') must have some relation either child to parent or vice versa or same-type.

- Otherwise : compile type error.
  ↳ Incovertable types found(d) type required (c).

i.e. = String s = new String ("Durga"); Not valid
StringBuffer sb = (StringBuffer) s;

2) C to A type.

C must be either same as A
or derived type of A.

→ 'C' must be either same or derived type of A
otherwise C.E. error
↳ Incompatible type
found c required A.

i.e. Object o = new String ("Durga");
StringBuffer sb = String (o); Not valid

JVM Rule for object casting

```
A  b  =  (C)  d
```

d → Underlying object of 'd' either same or derived one.

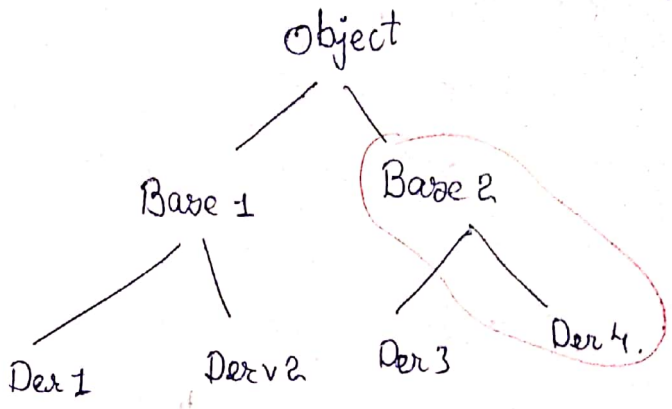
Underlined Object(d) is String.

i.e. Object O = new String("Durga");

StringBuffer sb = (StringBuffer) O;

Runtime type of object 'd' must be either same or derived type of 'C'.

Other wise we will get runtime exception.




```
Base2 b = new Der4();
```

1) `Object o = (Base2) b` → valid

2) `Object o = (Base1) b` → Inconvertible types.

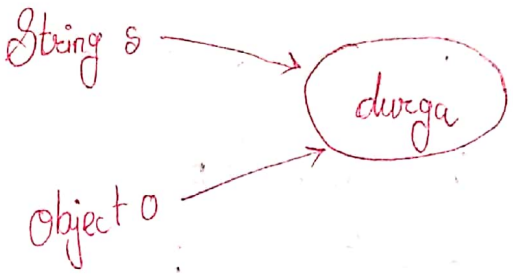
3) `Object o = (Derv1) b` → ×

4) `Base2 b1 = (Base1) b` → ×

5) `Base1 b1 = (Der4) b` → Incompatible types found.

Type Casting

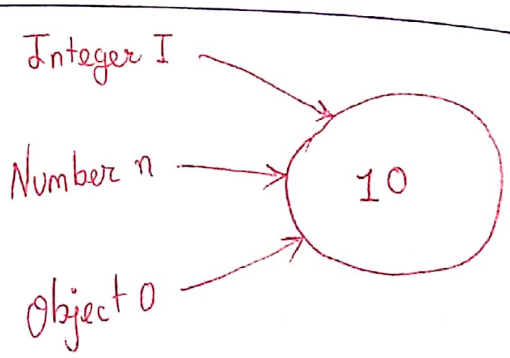
- 2 error compiler — Inconvertible / Incompatible
- 1 error JVM — ClassCastException
- Through type casting we are not creating any new object we are just giving new type for same object.



→ For existing object we are providing another type reference.

this type casting `String s = new String("Durga");`

`Object o = Objects`




```
Integer I = new Integer(10);
Number n = (Number) I;
Object o = (Object) n;
```

Creating only 1 object

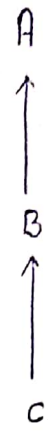
And having 3 reference variable.

Object o = new Integer(10);

(I == n) ✓

(n == o) ✓

```
C c = new C();

(B)c;
```



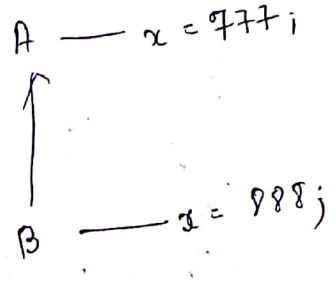
B b = new C();
↳ Run time object of C
↳ But type of B

A a = new C(); → (A)((B)c)
↳ Run time object is of C
↳ But type is of A.

Variable reference is based on reference type not runtime object.

B b = new B()
b.x → 888

A a = new B();
a.x → 777



Static Control flow

```
class Base
{
(1) Static int i = 10;
    Static
(3) {
      m1();
      Sout("First Static Block");
    }
    psvm(String[] args)
    {
      m1();
      Sout("main method");
    }
    ps v m1()
    {
      Sop(j);
    }
    Static
(4) {
      Sop("Second Static Block");
    }
(2) Static int j = 20;
}
```

i = 0 ↓ i = 10;

j = 0 ↓ j = 20

One Base class file will be generated.

① Identification of static member from top to Bottom

② Execution of static variable assignments & static Blocks. from top to Bottom

③ Execution of main method.

→ output

```
0
First Static Block
Second Static Block
20
main method
```

Static Block

→ At the time of class loading to perform any activity then need to go for static Block.

→ I want to load corresponding native library before class loading

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

→ JDBC

① Load Driver class
② get connection object
③ prepare Statement object
④ execute query
⑤ use ResultSet.

```
static
{
  Register this Driver
  with Driver Manger
}
```

→ Any no. of static block can be allowed
  ↳ But its execution will be done from top to Bottom

→ Without writing main method, is possible to write/print to console. with the help of static block.

```
→ class Test
  {
    static int x = m1();
    Public static int m1()
    {
      Sout("Hello I can print");
      System.exit(0);
      return 0;
    }
  }
```

Step-1] Static components assignment

Step-2] Initialization of static components.

1] x = 0
2] x = m1() ← method will be called.

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

Step 1] Static assignment
Step 2] Static components initialization and execution of static blocks

1] Static Test
2] execution of Test();

Whenever creating new object Instance block will be created.

Note: From 1.7 version onwards main method is mandatory to start a program execution, Hence from 1.7 version onwards without writing main method it is impossible to print some statement to the console

→ Untill 1.6v version without main method is possible to print statement in console.

→ Static control flow in parent to child relation

Static Block → parent & Child

```
Class Base
static int i = 10
static
  m1();
  Sout(Base);
psvm()
{ m1()
  Sout(Base main);
psv m1()
{ Sout(j);
static int j = 20;
```

```
Class Derived
static int x = 100;
static
{ m2();
  Sout(Derived);
psvm()
{ m2()
  Sout(Derived main);
```

Instance Control flow

```
class Test
{
  int i = 10;
  {
    m1();
    Sopen("First Instance class");
  }
  Test
  {
    Sout("Constructor");
  }
  psvm(String[] args);
  { Test t = new Test();
    Sout("main");
  }
  Public void m1()
  {
    Sout(j);
  }
  {
    Sout("Second Instance Block");
  }
  int j = 20;
```

Java test ↵

o/p → [main]

If I am creating and object then only instance level component will invoke and execute.

① Static Block will be executed.

② Identification of instance member from top to bottom

③ Execution of instance variable assignments and instance blocks from top to Bottom

④ Execution of constructor.

When Parent & Child is there.

① Identification of Instance member from parent to child.

② Execution of instance variable assignments & instance blocks only in Parent class.

③ Execution of parent constructor

④ Execution of instance variable assignments & instance block only in child class

⑤ Execution of child constructor.

→ Get new Instance object

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");
```

⑤ By using Descrialization method.

```
FileInputStream fis = new FIS ("dbc");
ObjectInputStream ois = new OIS (fis);
Dog dr = (Dog) ois.readObject();
```

Factory method

↳ Integer i = Integer.valueof (10);

Constructors

↳ To initialize an object.

→ Once we create an object compulsory we need to initialize an object.

→ If we are not writing any constructor then compiler will create a default constructor by its own.

Prototype of Default Constructor

① no-arg constructor is not a default constructor vice versa is possible.

② Access modifier of default constructor is same as top level class [public, default]

③
```
Test()
{ super(); }
```
Default constructor ↓ line

this/ Super must be added only in 1st line of constructor

No other statement allowed on 2nd line.

	Super(), this()	Super, this
①	Constructor call to call super class & current class constructor	to refer superclass instance member or current class instance member.
②	We can use only in constructors as first line	We can used anywhere except static area.
③	Within constructor only once.	We can used any no. of times